

Efficient Execution of ATL Model Transformations Using Static Analysis and Parallelism

Jesús Sánchez Cuadrado^{ID}, Loli Burgueño^{ID}, Manuel Wimmer^{ID}, and Antonio Vallecillo^{ID}

Abstract—Although model transformations are considered to be the heart and soul of Model Driven Engineering (MDE), there are still several challenges that need to be addressed to unleash their full potential in industrial settings. Among other shortcomings, their performance and scalability remain unsatisfactory for dealing with large models, making their wide adoption difficult in practice. This paper presents A2L, a compiler for the parallel execution of ATL model transformations, which produces efficient code that can use existing multicore computer architectures, and applies effective optimizations at the transformation level using static analysis. We have evaluated its performance in both sequential and multi-threaded modes obtaining significant speedups with respect to current ATL implementations. In particular, we obtain speedups between 2.32x and 38.28x for the A2L sequential version, and between 2.40x and 245.83x when A2L is executed in parallel, with expected average speedups of 8.59x and 22.42x, respectively.

Index Terms—Model transformation, MDE, ATL, performance, scalability, parallelization

1 INTRODUCTION

THE progressive adoption of Model-Driven Engineering (MDE) [1] approaches for developing better and more efficient software is posing different kinds of challenges to current MDE methods and tools. Despite the potential benefits of MDE technologies to significantly reduce time to market and improve product quality, they still suffer from some limitations that may hinder their full adoption by industry (see, e.g., [2], [3], [4]). In particular, the scalability, usability and performance of model transformations (MT) are crucial issues that need to be tackled if they are to be effectively used to address scenarios such as model-driven modernization of legacy systems and the engineering of large and complex applications in, e.g., the automotive, biology or aerospace domains.

At this moment, ATL [5] and QVT [6] are the most widely-used model transformation languages [7]. Although they provide powerful abstractions to specify and implement transformations between models and to generate model views, their implementations have limited scalability, and thus the execution time of transformations may become

prohibitive with large input models (e.g., in the order of millions of elements), or even medium-size input models if the transformation has complex model navigations. One reason for this lack of scalability is due to the fact that most transformation engines are implemented as simple interpreters and they barely use static analysis information to apply compile time optimizations or to improve their scheduling. Moreover, although multicore computers are widely available, there are very few engines that implement parallel transformation algorithms.

The contribution presented in this paper addresses the engineering of an efficient model transformation engine for the particular case of the ATL model transformation language. We have developed a new compiler for ATL, called A2L, which provides several novel features with respect to state-of-the-art approaches, namely:

- A2L uses static analysis information provided by AnATLyzer [8] to compile ATL transformations to the Java Virtual Machine (JVM), applying optimizations for OCL expressions and for transformation rule handling.
- We present a novel algorithm which enables the parallel execution of the transformation, using data parallelism. This allows A2L to achieve an effective distribution of the parallel jobs, thus outperforming other parallel ATL engines which are based on task parallelism [9], [10].
- A2L is integrated with the ATL/AnATLyzer IDE and Eclipse Java Development Toolkit (JDT), which enables the development of transformations using the facilities provided by AnATLyzer, e.g., quick fixes [11] and visualizations [12]. Moreover, the compiled code can be seamlessly integrated with existing Java code.

A2L has been validated for correctness using the regression tests defined for the ATL virtual machine [13] and supports

- Jesús Sánchez Cuadrado is with the Universidad de Murcia, Department Informática y Sistemas, Campus de Espinardo, Murcia, Spain. E-mail: jesusc@um.es.
- Loli Burgueño is with the Open University of Catalonia, IN3, 08035 Barcelona, Spain, and also with the Institut LIST, CEA, Université Paris-Saclay, 91120 Paris, France. E-mail: lburgueno@uoc.edu.
- Antonio Vallecillo is with the ITIS Software, Universidad de Málaga Bulevar Louis Pasteur, 35, 29071 Malaga, Spain. E-mail: av@lcc.uma.es.
- Manuel Wimmer is with the Johannes Kepler Universität, Business Informatics – Software Engineering, Linz, Austria. E-mail: manuel.wimmer@jku.at.

Manuscript received 25 Feb. 2020; revised 19 June 2020; accepted 19 July 2020. Date of publication 23 July 2020; date of current version 18 Apr. 2022. (Corresponding author: Jesús Sánchez Cuadrado.) Recommended for acceptance by S. Nejati. Digital Object Identifier no. 10.1109/TSE.2020.3011388

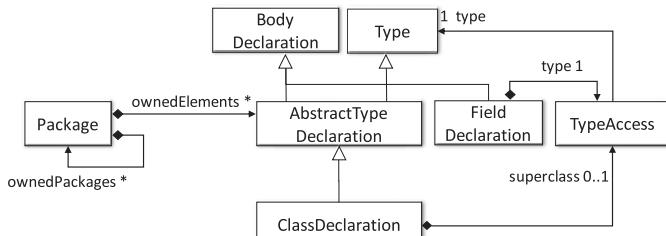


Fig. 1. Excerpt of the Java metamodel (MoDisco).

the majority of the constructs of ATL, including all types of rules (matched, lazy and called), module and context helpers, imperative blocks, all datatypes including collections, maps and tuples, and the standard OCL library. We have run several benchmarks that show significant performance improvements when compared with the existing ATL engines.

This paper is organized as follows. Section 2 introduces the ATL model transformation language and describes the limitations of current transformation engines through a running example. Then, Section 3 describes the architecture of A2L, the compiler we have developed to compile and execute in parallel existing ATL programs. The more prominent features of A2L are described in Sections 4 and 5, which present, respectively, the algorithm used to execute ATL transformations in parallel, and the A2L optimization strategies and mechanisms enabled by the use of AnATLyzer static typing information. Section 6 describes the evaluation that we have conducted to validate our proposal. Finally, Section 7 discusses related work, and Section 8 concludes with an outline on future work.

2 MOTIVATION AND BACKGROUND

Performance and scalability of model transformations is deemed as one of the most important challenges in MDE since it enables the use of model transformation technology to handle large models appearing in scenarios like reverse engineering, model analysis, or data engineering. It is also key to apply MDE to other engineering disciplines, such as construction [14] or automotive engineering [15].

Model transformation languages, notably those with a declarative form, have the potential to tackle this challenge because they provide an abstraction to write transformations which are independent of the execution mechanism. A good compiler should generate efficient code by analysing the structure and relationships of the transformation. However, this possibility has not been exploited in state-of-the-art MT languages, resulting in poor performance. In fact, a recent study [7] has revealed that, although MT users value the advantages of using MT languages, the poor performance and scalability issues of MT engines are hampering their use and forcing them to develop their transformations in general-purpose languages (even though it makes the task more cumbersome and error-prone).

Our working hypothesis is twofold. First, by using static analysis information, it is possible to compile declarative transformations to produce high-performance code; moreover, recurrent transformation idioms can be optimized by the compiler. Second, since a declarative transformation does not prescribe the execution order, it is possible to seamlessly

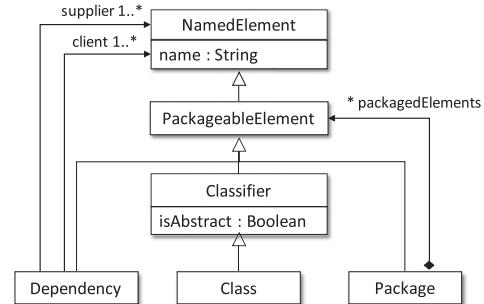


Fig. 2. Excerpt of the UML metamodel.

execute a transformation in parallel, if the adequate transformation algorithm is implemented. More precisely, to achieve efficient parallel execution of ATL programs, such an algorithm should limit the number of dependencies between parallel processes to maximize concurrency, while load balancing between processes should aim at preventing processes from becoming idle if they finish before others. To this end, we propose the use of data-based parallelism, whereby the model is split into chunks of elements that are transformed by the processes, all running the complete transformation in parallel.

This paper describes our proposed parallelization algorithm, its main features and characteristics, the optimizations we have applied, and the performance gains it achieves over existing ATL model transformation engines.

2.1 ATL

ATL [5] is a hybrid model transformation language that allows both declarative and imperative constructs. A transformation consists of a set of rules that specifies which elements of the output model are created from which ones of the input model.

Listing 1 shows an excerpt of an ATL transformation taken from the ARTIST project [16] that generates a UML class diagram (a dependency view) from a Java project. Excerpts from the input and output meta-models of this transformation are depicted in Figs. 1 and 2, respectively. In ATL, the main type of rule is the so-called *matched rule*. It consists of an input pattern that might have a filter condition which is matched on the source model, and an output pattern that produces a set of elements in the target model for each matched input pattern. OCL expressions [17] are used to calculate the values of features of the target elements. In this excerpt, we have included two matched rules, *Package2Package* and *Class2Class*, which take package and class elements respectively from the Java model and convert them to the corresponding counterparts in the UML model, but filtering proxies out (not *s1.proxy*). A rule body consists of *binding* elements. A binding either assigns a primitive value (e.g., *name ← s1.name*) or resolves the source values appearing in its right-hand side (RHS) to target values generated by other rules. For example, the binding in line 15 retrieves and assigns the subpackages mapped by rule *Package2Package*, and the binding in line 16 retrieves and assigns all non-proxy classes mapped by rule *Class2Class*.

The ATL transformation algorithm works in two phases, which are graphically illustrated in Fig. 3. The left-hand side

```

1 -- @nsURI UML=http://www.eclipse.org/uml2/3.0.0/UML
2 -- @nsURI JAVA=http://www.eclipse.org/MoDisco/Java/0.2.incubation/java
3 module java2uml;
4 create OUT : UML from IN : JAVA;
5
6 helper context JAVA!Package def: nonProxyClasses : Sequence(JAVA!ClassDeclaration) =
7   self.ownedElements->select(e | not e.proxy)->select(e | e.oclIsTypeOf(JAVA!ClassDeclaration));
8
9 helper context JMM!ClassDeclaration def : getRefClassFields : Sequence(JMM!FieldDeclaration) = ...
10
11 rule Package2Package {
12   from p : JAVA!Package (not p.proxy)
13   to t : UML!Package(
14     name <- p.name,
15     packagedElement <- p.ownedPackages->select(e | not e.proxy)->select(e | e.oclIsTypeOf(JAVA!Package)),
16     packagedElement <- p.nonProxyClasses,
17     packagedElement <- p.nonProxyClasses
18     ->select(p2 | not p2.getSuperClass.oclIsUndefined())
19     ->collect(p2 | thisModule.createGeneralizationDependency(p2)),
20     packagedElement <- p.nonProxyClasses
21     -> collect(p2 | p2.getRefClassFields)->flatten()->collect(e|thisModule.createUsageDependency(e))
22   )
23 rule Class2Class {
24   from c : JAVA!ClassDeclaration (not c.proxy)
25   to t : UML!Class ( name <- c.name )
26   }
27 lazy rule createGeneralizationDependency {
28   from class : JAVA!ClassDeclaration
29   to d : UML!Dependency (
30     supplier <- Sequence { class.superClass.type },
31     client <- Sequence { class }
32   )
33   }
34 lazy rule createUsageDependency {
35   from field : JAVA!FieldDeclaration
36   to d : UML!Dependency (
37     supplier <- Sequence { field.type.type },
38     client <- JAVA!ClassDeclaration.allInstances()->select(cd | cd.bodyDeclarations->includes(field))
39   )
40 }

```

Listing 1. Excerpt of the *Java2UML* transformation.

shows a sample input model. In the *first phase* each matched rule accesses the input model to get all elements whose type is compatible with its input element (specified in the *from* part of the rules). The filter is used to rule elements out. In the example, elements *jp1* and *jp3* are retrieved by the rule *Package2Package* but only *jp1* satisfies the filter and is matched. When an element is matched, the target elements specified in the *to* part of the rules are created and a traceability link is established (depicted by a dashed arrow in the image) which includes a reference to the rule producing the link. The *second phase* of the algorithm consists of traversing all traceability links and resolving each rule binding in order. To resolve a binding, its OCL expression in the RHS is evaluated. If the result is a primitive type, the value is directly assigned to the feature in the left hand side. If it is an object (or a collection of objects), the internal trace is looked up to retrieve the corresponding target element and it is assigned to the left hand side (if it is a collection, the value is added). In the example, to resolve the binding in line 16, the engine retrieves and assigns target objects *c1* and *c2* from source objects *jc1* and *jc2* respectively.

ATL also supports rules which must be explicitly invoked. This is the case of *lazy rules*. A lazy rule can be seen as a global function that takes model elements as parameters and returns a target model element, which is created and initialized by the rule. In the example, the lazy rule *createUsageDependency* (line 38) defines a dependency between the class that defines a field and the field type. In line 21, the rule is invoked. Since the lazy rule generates the target element, it can be assigned directly in the corresponding binding (i.e., no binding resolution is needed).

The ATL code is compiled into bytecode for its execution using two main runtime engines: the default ATL virtual machine [5], which was released along with the ATL language; and EMFTVM [18], which provides performance improvements as well as other advanced language features such as the possibility to execute in-place model transformations.

2.2 Static Analysis of Model Transformations: AnATLyzer

A model transformation is typed against its input and output meta-models. This means that the types and features used in the transformation program must exist in the

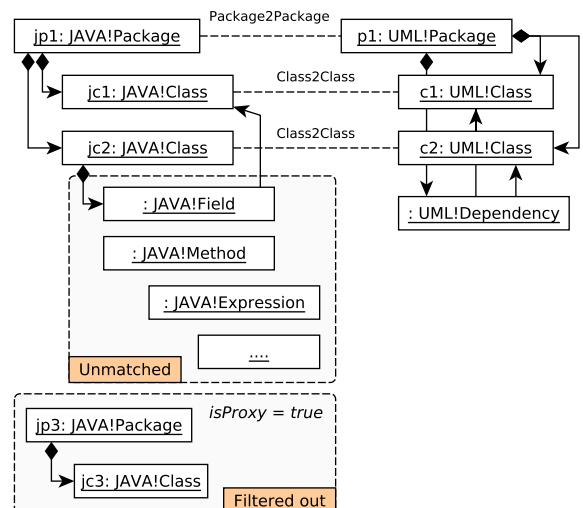


Fig. 3. Representation of a sample transformation execution.

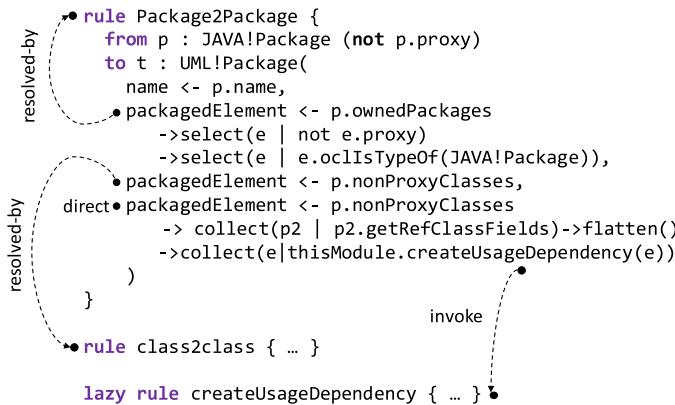


Fig. 4. Static analysis of the example transformation.

corresponding meta-model. This can be enforced dynamically (at runtime) or statically (at compilation time). Some languages, like QVT-Operational, enforce this statically, while others, like ATL and Epsilon ETL [19], do it dynamically. In addition, data dependencies between the input model and the rules which match the elements, as well as among the transformation rules (i.e., established by means of bindings in the case of ATL), may exist. In this work, we have used AnATLyzer [8], [12] to statically analyse ATL transformations and benefit from this information to implement our compiler.

Fig. 4 illustrates some of the static analysis information made available by AnATLyzer. First, every node of the abstract syntax tree of the transformation is annotated with its type (i.e., a reference to the corresponding meta-model element). There are three bindings to initialize packageElement, whose semantics is to add elements (i.e., the second binding does not override the previous setting because it is a collection). For example, the type of the first binding is Sequence (JAVA! Package) because AnATLyzer recognizes the use of *oclsTypeOf* and performs an implicit casting. From the inferred types, AnATLyzer builds a graph to make dependencies among rules explicit. For instance, the first binding packagedElement can only be resolved by rule Package2Package because of the implicit casting that determines that the RHS of the binding will only have JAVA! Package elements. The second binding is resolved by the rule Class2Class because the return type of nonProxyClasses is Sequence (JAVA! ClassDeclaration). Finally, the third binding does not need to be resolved because it directly assigns target elements generated by the createUsageDependency rule.

2.3 Limitations of Current Approaches

The original ATL transformation algorithm, based on the two phases described above, and its implementations (both in the standard ATL Virtual Machine (VM) [5] and EMFTVM [18]), can cope with scenarios involving small or medium-size models. However, their performance and scalability rapidly degrade as the size of the input models grows. Among other reasons, they fail to exploit a variety of interesting performance and optimization opportunities, which are described next.

Limited Parallelism. The algorithm and its current implementations are sequential. A relatively simple approach to make the algorithm parallel is to use task parallelism [10],

in which the parallelisation unit is the transformation rule. However, this approach is sub-optimal since it suffers from lock contention and unbalanced loads (i.e., some threads will be idle if they finish their tasks earlier than others). Using a data parallelism approach, all processes perform the same task, but on different chunks of data—it is the data that is split. Contrarily, in task parallelism, it is the model transformation that is split into separate smaller processes (e.g., a rule) and all of them work on the same data. As demonstrated in [20], using data parallelism to implement concurrent model transformations can produce significantly better results.

Inefficient Model Access. A transformation engine which does not exploit type information (such as the existing ATL virtual machines) does a “blind access” to the input model. This means that after loading the input, it cannot discard unused parts of the model. For instance, in the example (see Fig. 3), objects of types JAVA! Field and JAVA! Method will never be matched by a rule, and thus, they could be ruled out in the loading phase. We will improve rule matching by considering, in a pre-processing step, only those elements that are relevant for the transformation.

Expensive Runtime Checks. The ATL compiler does not perform any type checking, which means that it needs to insert code to perform dynamic checks, including, e.g., the cardinality of the LHS of the binding, or calls to helper methods. Moreover, it can only use the reflective EMF API, which also imposes an additional overhead. We will use the information made available by the type checker to avoid these kinds of overheads.

Lack of OCL Optimizations. Complex transformations typically contain many OCL expressions and operation helpers. These expressions are often devoted to navigating collections. A good implementation of OCL is critical to achieve a satisfactory performance on large models—especially when collection operations are involved. It has already been reported that the standard ATL VM does not handle large collections efficiently [21] and it is the EMFTVM engine which does provide a better implementation. However, both engines have not addressed optimizations yet. For instance, the expression

```

s1.nonProxyClasses->collect(p2| p2.getRefClassFields)
    ->flatten()->collect(e|thisModule.
        createUsageDependency(e))

```

requires two intermediate collections to be created (for the first collect and the flatten). An optimizer could identify this pattern and evaluate the expression without creating unnecessary intermediate collections.

To the best of our knowledge, there is no transformation engine that makes use of static analysis information to improve its performance, and combines this with parallelism to take advantage of all the computing power of current CPUs.

3 A2L: A COMPILER FOR ATL

Our technical approach to address the limitations presented above is based on a compiler from ATL to Java. Fig. 5 shows its architecture.

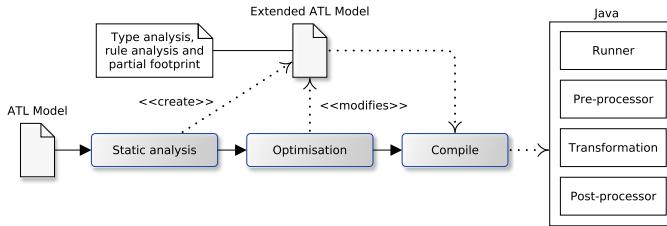


Fig. 5. Compiler architecture.

First, it performs *static analyses* using AnATLyzer. This produces an extended ATL abstract syntax model (AST), which includes type information, control flow and data flow information. This will be used throughout the compilation process.

The *optimization* phase (described in detail later in Section 5) is in charge of detecting common transformation patterns that can be particularly managed to generate efficient code. To implement the optimizations, special nodes are added to the AST. Each of these nodes represents an optimization pattern. The optimizer is in charge of detecting the patterns and replacing the AST nodes. Then, the compiler has specific extensions to produce specific code for these nodes. This optimization phase is optional and can be disabled.

The *compilation* phase generates all the code and related artefacts needed to execute the transformation in the JVM. An essential feature of A2L is that it targets a transformation algorithm specifically designed to execute the model transformations concurrently, using data parallelism to achieve adequate performance results (the algorithm is described in detail later in Section 4, and the gains in performance are presented in Section 6).

As a result of these three steps, the compiler generates four main artefacts (Fig. 5). The Runner allows the user to configure the transformation execution programatically (e.g., to set the input models, to configure the number of threads, etc.). The Pre-processor is in charge of filtering the input models to optimize the rule matching by considering only those elements required by the transformation rules. The Transformation contains the actual transformation behavior which will be executed by the parallel processes. Finally, the Post-processor is in charge of combining the results of all the processes that have been working in parallel to realize the transformation, and to generate the output models.

The following sections describe in more detail these features of the A2L compiler. We begin explaining our parallel transformation algorithm. Then, we explain the optimization strategies and mechanisms to implement them.

4 PARALLEL EXECUTION OF ATL TRANSFORMATIONS

To address the lack of parallelism of ATL we have designed a new transformation algorithm. The algorithm is intended to respect the semantics of the original one but, in addition, it enables data-based parallelism and focuses on minimizing the amount of lock contention among the worker threads.

In data-based parallelism, all threads execute the same code but on different chunks of data. The advantage over task-based parallelism, as proposed in [10] for ATL, is that processors are less prone to be idle. We have redesigned the

ATL transformation algorithm to make it amenable to data-based parallelism, generalizing the approach proposed in [20]. Our algorithm is presented in Algorithm 1. It works in three phases, pre-processing (line 4), execution (line 4) and post-processing (line 10). The architecture to execute these phases in parallel is illustrated in Fig. 6 and it is described next.

Algorithm 1. Data-Oriented ATL Algorithm

```

1 def transform(model, transformation)
2   // Step 1: Pre-processing
3   types ← footprint(transformation)
4   buffer ← preprocess(model, types)
5   // Step 2: Parallel execution
6   // This loop does sequential execution,
7   // parallel execution requires assigning jobs to threads
8   foreach e in buffer do
9     execute(element)
10  end
11  // Step 3: Post-processing
12  foreach b in pendingBinding do
13    resolve(b)
14  end
15 end
16 def execute(element)
17   foreach rule in transformation.rules do
18     if rule.filter(element) then
19       executeRule(rule, element)
20       break
21     end
22   end
23 def executeRule(rule, element):
24   foreach type in rule.outputElements do
25     target = createObject(type)
26     create trace link (element, target)
27   end
28   foreach binding in rule.bindings do
29     right = evaluate(binding) if binding is primitive then
30       target."binding.feature" ← right
31     else
32       // Resolve the binding
33       foreach resolving in binding.resolvingRules do
34         if resolving.filter(element) then
35           add to pendingRules (target, binding, right)
36         break
37       end
38     end
39   end
40 end

```

Pre-Processing. The input model is read from some source (label 1). Its elements are placed in a buffer which will be used by worker threads in the next phase when fetching work. However, not all model elements are required by all worker threads, only those whose type is declared by the matched rules in their source patterns. Thus, this set of types is extracted from the static analysis of the transformation, and used to filter the source model (lines 3-4). In the transformation example, this set consists of Package and ClassDeclaration types. The intended effect is to reduce the size of the buffer and to speed up rule matching, since there are less elements to consider. Although this step is

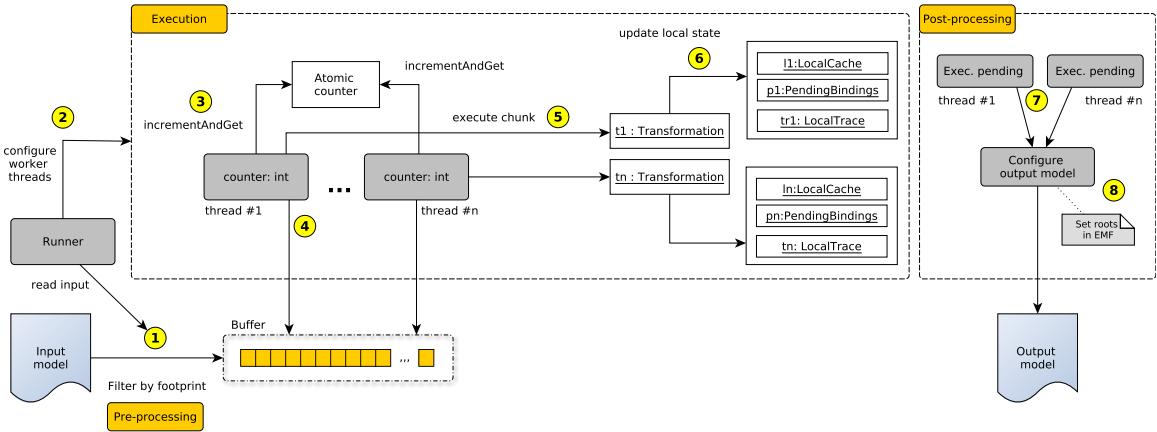


Fig. 6. Components of the technical realization of the parallel transformation algorithm.

done sequentially, the actual overhead due to the filtering is small since the engine needs to load and prepare the input model in any case.

Execution. This phase is in charge of executing the transformation logic. In the sequential version each element in the buffer is processed one after another. In the parallel version, we spawn worker threads (label 2). A worker obtains a chunk of data from the buffer (label 4), which is split into chunks of a given size (e.g., 512 model elements). Each worker has a counter to represent the chunk that is currently transforming. When it finishes, it asks for the next chunk to the scheduler, which uses an atomic integer variable to represent the last chunk given to a worker. The scheduler uses an atomic operation to increment the counter (label 3), which means that the increment operation for the chunk counter does not need to be guarded by a lock because it is done atomically. This way, there are no locks involved in the algorithm and we expect less contention. There are several possible strategies to split to decide the chunk size. The simplest one is to use a fixed chunk size. The larger the chunk size the less competition to get more work. However, it may happen that some threads are idle at the end of the transformation execution (i.e., load imbalance). On the contrary, setting a small chunk size would lead to more contention. The alternative is a dynamic scheduling policy in which chunks are larger at the beginning and smaller towards the end, but this implies some overhead. Therefore, we have implemented a mixed strategy in which we split the buffer in two parts. The first one is statically divided into chunks of size $0.75 \times \text{buffer_size}/\text{num_workers}$, which are large enough for each worker to start transforming elements. As soon as these chunks are finished, the algorithm continues with a dynamic scheduling strategy that uses smaller chunks (10 elements in the current configuration) to prevent workers from being idle.

Each worker uses a new instance of the transformation, given that we are using data-based parallelism, which has its own local state so that threads do not compete to access shared resources (label 6). For each element of a chunk, we try to find a matching rule. In practice, rule matching consists of checking the rules of the transformation in some order. Actually, in ATL the order is irrelevant because a given source element can only be matched by one rule, otherwise a runtime error is raised. Algorithm 2 illustrates the style of the code generated for matching the rules of the running example.

Algorithm 2. Example of Rule Matching

```

1 def transform(element)
2   if model2model_match(element) then
3     model2model_execute(element)
4   else if package2package_match(element) then
5     package2package_execute(element)
6   else if class2class_match(element) then
7     class2class_execute(element)
8   end
9 end

```

If a matching rule is found, the output elements are created and a trace record is generated to establish a mapping between the input and output elements. Thus, each transformation instance contains a partial trace (*LocalTrace* in the figure) to store such records locally to avoid any locking situation. In addition, the RHS of each binding is evaluated. Primitive bindings are assigned directly, but non-primitive bindings (i.e., those whose RHS evaluation returns a set of model elements) cannot be evaluated because the corresponding target models may not have been transformed yet. Thus, we delay this task by recording the fact that such dependency has yet to be resolved (*PendingBindings* in Fig. 6). This approach is a generalization of [20] in which special identifiers are used to construct model references which are yet to be resolved. The advantage in this case is that it is independent of the meta-modelling framework and the subsequent resolution does not require to traverse the full target model, but can be done by a constant-time look up.

Post-Processing. The main task of this phase is to resolve bindings, given that now all target model elements are available. For each unresolved binding its RHS is used to look up the partial traces in order to retrieve the corresponding target elements and assigning them to target features (label 7). In the sequential scenario, looking up a single trace has a constant cost $O(1)$ because the trace model can be indexed by source element using a hash map. However, an undesirable effect of the parallelization is that, now, each trace lookup has a cost proportional to the number of worker threads, because for p threads we may need p accesses to the partial traces (that is, the cost is $O(p)$). To mitigate this shortcoming, we also execute the post-processing in parallel. This requires classifying unresolved references

into overlapping and non-overlapping. Two references are overlapping if they may potentially cause a race condition when set in parallel.¹ This is the case of opposite references and references that are set in more than one location in the transformation. In the example, `client` and `supplier` are non-overlapping, but `packagedElement` is overlapping because there are several assignments in the same rule. At the end of this phase, the output model may be configured as required by the underlying meta-modeling framework (label 8). In particular, if the target model is in EMF format, we need to establish the root elements of the resulting EMF resource.

This algorithm exhibits almost the same functional behavior of the original ATL algorithm, but it enables data parallelism. The only observable difference is that the order in which root elements appear in the model is not deterministic (i.e., elements which are not assigned to any containment reference) because each run may allocate chunks differently. Section 6 evaluates the speed up that can be obtained by applying it to exploit multi-core CPUs. However, it is possible to achieve even greater performance by optimizing the compilation of the sequential part of the transformation.

5 OPTIMIZATIONS

The availability of typing information allows our compiler to target a typed runtime environment like the JVM. This already provides a performance improvement over dynamically typed languages like ATL or ETL, and over interpreter-based approaches like most QVT implementations. In addition, another significant increase in performance is possible by applying a number of optimizations to handle common transformation scenarios and idioms. The most relevant optimizations that A2L implements are described next.

5.1 Optimizations at the Transformation Level

These optimizations are intended to improve the performance of the execution of the transformation rules. They include the process of matching input elements and the binding execution.

Matched Rule Ordering. Our algorithm tries to match each input element against the input pattern of each matched rule. Algorithm 2 shows the generated code. The sooner the matching rule is identified, the more efficient the process is, because it avoids checking unnecessary conditions. Finding the matching rule as soon as possible depends on three main factors: the order in which the patterns are checked, the complexity of the rules' filters, and the structure and contents of the input model (some rules are matched more frequently than others depending on the particular model).

Our approach is to heuristically prioritize rules according to their filters. We count the number of OCL elements to be evaluated as part of the filter execution, and check the rules with fewer elements first. The rationale is that, in the worst case, all rules must be checked and thus it is preferable to

check the cheaper ones first. If a given element is eventually matched by a rule with a costly filter, the time spent on checking the wrong rule filters first is low. However, other heuristics are possible, such as estimating the chances of matching an element by considering the operations used in the filter (e.g., equality operation would match less elements than inequalities).

Transformation Footprinting. We use the transformation footprint to filter the input model in order to consider only those elements that may actually be matched by some matched rule. Notably, we only use the partial footprint, meaning that we are only interested on the types declared in the source pattern of the rules. In the running example, the partial footprint of that transformation consists of the set `footprint={Package, Class}`. This is specified in line 3 of Algorithm 1. We intend to significantly reduce the buffer size in scenarios in which the transformation only matches a small subset of the model. Although the default ATL algorithm behaves differently (it computes an associative table to gather objects per type) it would also benefit from this optimization since the table size could be smaller by only recording needed elements. Moreover, we use this step to pre-compute global data such as `ClassDeclaration.allInstances()`, thus avoiding another model traversal.

Binding Handling. ATL relies on binding resolution to assign target references implicitly. To resolve a binding, ATL checks if the value of the RHS is a primitive value, an object, or a collection of primitive values and/or objects. If it is a collection, it might be the case that it is a nested collection, in which case it needs to be flattened. A2L does not need to check these conditions at runtime since it knows at compile time whether the RHS contains primitive or object values, and whether collections are nested or not.

Trace Footprint Reduction. ATL relies on recording trace links between the input elements matched by a rule and the corresponding elements created upon execution. A transformation engine without access to information about rule relationships will generate trace links even in cases where they are not needed. In A2L, we apply an optimization to reduce the trace memory footprint, namely, we analyse which rules may need to resolve a given binding. There are two main scenarios: a) if a matched rule is never used to resolve a binding, the link between the input element and the primary output pattern element does not need to be recorded, and b) if a matched rule has more than one output pattern element, there is no need to record the trace link for the secondary elements (i.e., all elements except the first one) unless there is a `resolveTemp` operation which retrieves them (in ATL, the `resolveTemp` operation is used to explicitly retrieve a target element from a given source element).

This optimization is useful to reduce the memory footprint when there are rules that create, for a single input element, a large connected set of elements, but other rules only need to link a single element—typically the parent of these elements.

5.2 OCL-Related Optimizations

As mentioned in the introduction, A2L supports all ATL and OCL datatypes including collections, maps and

1. Some meta-modelling frameworks, including EMF, are not thread-safe.

tuples, and the standard OCL library. ATL makes heavy use of OCL expressions for navigating the models, selecting elements in the rule filters, and for calculating the values of the target elements' features. However, the evaluation of these OCL expressions is often inefficient, in particular regarding the use of collections. This is due to the fact that OCL datatypes are immutable, which means that each operation over a collection needs to return a new collection. The transformation engine should therefore use internally a library for immutable collections, but this is sometimes not enough when dealing with very large models. The problem is that these auxiliary collections can be huge and they can be unnecessarily created several times during an OCL expression evaluation. Moreover, the typical OCL access patterns for collections are not well suited for immutable collections (e.g., in sequences including appends an element, but in an immutable list prepend is typically more efficient).

To address this issue, we have a two-step approach. In the first step, we perform escape analysis to check whether a given collection may be modified in more than one location. In this case, it is not possible to apply any optimization and we resort to immutable collections. However, if a collection is not going to be shared, we mark each involved (sub-)expression as *mutable*, so that in the second step the compiler is free to generate code using mutable collections. For instance, in the following listing, the first example shows a piece of ATL for which it is possible to generate code using mutable collections, whereas in the second example we must use immutable collections

```

1 -- Intermediate collections do not escape the
2 -- expression
3 s1.ownedPackages->select(p | not p.proxy)
4           ->collect(p | p.ownedElements)
5 -- The pkgs collection is shared
6 let pkgs : s1.ownedPackages->select(p | not p.proxy)
7   in if pkgs->including(aPkg)->size() > 2
8     then pkgs else Set {} endif

```

In a second step, we use the results of the escape analysis. We try to optimize certain access patterns for which we can generate optimized code that avoids redundant creation of temporary collections by using mutable collections. For instance, the evaluation of the following OCL expression using immutable collections requires traversing two intermediate collections (one for select and another for collect) whose size is equal to the number of UML!Class instances.

```

UML!Class.allInstances()->
  select(c| not c.owningPackage.oclisUndefined())->
  collect(c| c.owningPackage())->asSet()

```

However, if the resulting collection is not shared we can generate more efficient code which avoids unnecessary processing and memory usage. Our compiler detects this particular pattern and generate specific code for it, using only one traversal of the source collection and without intermediate collections. The following listing illustrates the code that would be generated.

```

Set<Package> result = new HashSet<>();
for(Class c : model.allInstancesOf(Class.class)) {
  if (! (c.getOwningPackage() == null)) {
    result.add(c.getOwningPackage())
  }
}

```

Table 1 shows a summary of the most relevant optimizations, described by means of their context, a prototypical example of each one, and how they are implemented in A2L, i.e, the Java code generated for them.

5.3 Automatic Caching

This optimization deals with OCL code that computes the same value several times, and such computation can be potentially time consuming. It is possible to increase the execution performance if such computations are cached so that they are reused in subsequent accesses. ATL supports caching by factorising code in attribute helpers, but this requires the developer to identify which code locations should be cached. Our compiler detects some of these locations and generates code that caches repeated results. In particular, we consider a hot spot a sub-expression within a nested loop such that it starts with a variable that is independent of the outermost loop. For instance, in the following code the value `c2.allSuperClasses() -> reject(...)` is reused across iterations of the outer `forAll` because it is cached.

```

-- Classes in pkg must have a non-abstract subclass
pkg.ownedClasses->forAll(c1|
  UML!Class.allInstances() ->exists(c2|
    c2.allSuperclasses() ->reject(c | c.isAbstract) ->
    contains(c1))

```

The last row of Table 1 shows another form of automatic caching, in which certain access patterns are compiled as an indexing operation. The index is filled in the pre-processing phase in order to provide fast access during the transformation execution.

6 VALIDATION

To evaluate our approach, we have defined four research questions regarding the correctness of the obtained transformation output models, the completeness of the ATL language support, the speedup compared to existing ATL engines and, finally, the scalability of A2L, i.e., the speedup gained when raising the number of available cores. To answer these questions, we carried out an empirical case study [22] by following the guidelines for conducting empirical explanatory case studies by Roneson and Hörst [23]. Moreover, the implementation, case studies and scripts to reproduce our results are available at <http://github.com/analyzer/a2l>.

In the next subsections, we describe our research questions and the case studies and metrics we have used to answer these questions. Finally, we discuss the answer to each research question and the overall threats to validity of our proposal.

6.1 Research Questions

Our study addresses the following four research questions. With these questions, we aim to justify the use of our

TABLE 1
Summary of the Most Relevant OCL Optimisations

Name	Example	Generated code
Mutable addition	aPkg.ownedElements ->select(p p.isProxy) ->including(aClass)	<pre>res = new ArrayList<AbstractTypeDcl>() for(o : aPkg.getOwnedElements()) if (o.isProxy()) res.add(o) res.add(aClass)</pre>
Filter and check existence	aPkg.ownedElements ->select(p p.oclIsKindOf(JAVA!ClassDeclaration)) ->exists(c c.isAbstract)	<pre>boolean result = false for(o : aPkg.getOwnedElements()) if (o instanceof ClassDeclaration) ClassDeclaration cd = (ClassDeclaration)o if (cd.isAbstract()) result = true break</pre>
Filter and count	aClass.bodyDeclarations ->select(d not d.isProxy) ->select(d d.modifier.static) ->size()	<pre>size = 0 for(d : aClass.getBodyDeclarations()) if (!d.isProxy() && d.getModifier().isStatic()) size++</pre>
Filter and map	aPkg.ownedElements ->select(p not p.proxy) ->collect(p p.class) ->collect(c c.name)	<pre>res = new ArrayList<String>() for(p in aPkg.getOwnedElements()) if (!p.isProxy()) tmp1 = p.getClass() tmp2 = tmp1.getName() res.add(tmp2) // or res.addAll(tmp2) if flatten</pre>
Collection conversion	classes ->collect(c c.name) ->asSet()	<pre>// Target collection is created beforehand res = new HashSet<String>() for(c in classes) res.add(c)</pre>
Indexing	(1) JAVA!ClassDeclaration.allInstances() ->exists(p p.name = name) (2) JAVA!ClassDeclaration.allInstances() ->select(p p.name = name)	<pre>// A global object is initialized in // the pre-processing phase with // (1) Set<String> // (2) Map<String, List<ClassDeclaration>> globalContext.existsIndex.contains(name) globalContext.selectIndex.get(name)</pre>

The target code is written in a Java-like pseudocode for simplicity.

compilation strategy and our parallelization approach in order to significantly improve the performance of ATL transformations. At the same time, we argue about the correctness and completeness of our approach.

RQ1 Compiler Correctness: Does the code generated by the A2L compiler exhibit the same functional behavior as the standard ATL engine? To validate the correctness, we compared the results of running transformations compiled with A2L against the results of running the same transformations on the standard ATL VM. We resort to an available set of regression tests already used by ATL transformation engines [13] to test their correctness.

RQ2 Compiler Completeness: How much of the ATL language is the A2L compiler able to deal with? To validate its completeness, we evaluated the coverage of the ATL language, defined by the ATL metamodel, for which the A2L compiler provides support. Moreover, we have manually checked against the ATL documentation which features are actually supported by A2L.

RQ3 Performance: What is the gain in performance when compared with ATL VM and EMFTVM? To evaluate the performance of the implementation we have used seven case studies, which exercise several transformation styles and ATL constructs. We compared the execution times of different A2L versions (non-optimized, optimized, sequential and parallel) with

the standard ATL VM and EMFTVM. EMFTVM is a newer ATL engine that compiles to Java bytecode on the fly and it is reported to achieve gains of 80 percent in basic benchmarks. In the experiments, we used A2L in both sequential and parallel mode. The sequential execution allows us to show the gains obtained only by the use of static analysis and optimizations. The parallel execution aims to validate our parallelization strategy. Since the optimizations are optional, we also compared the executions with and without the optimizations, to assess their impact on performance.

RQ4 Parallelism: What are the effects of adding more cores? We analyse how the number of cores influences the execution times of ATL transformations parallelized by A2L. We have executed our case studies using an increasing number of threads and recorded the obtained speedups.

6.2 Experimental Setup

6.2.1 Case Studies

RQ1 and RQ2 are evaluated using the same set of 24 regression tests [13] that the ATL team used to validate the correctness of two consecutive versions of the ATL Virtual Machine. Each of these tests consists of one model transformation and all the necessary artifacts needed to execute the transformation, i.e., the input and output metamodels, and a sample input model. Since none of these tests provides large

TABLE 2
Main Features of the Case Studies

Name	M_R	L_R	H	B	I	I/O size	Match. cost	Exec. cost	FP size	OCL elements
airquality	1	0	0	0	0	$I_{size} > O_{size}$	high	low	equal	collections, allInstances
dblpv1	1	0	1	0	0	$I_{size} > O_{size}$	medium	medium	equal	collections
dblp2bibtex	7	2	1	1	0	$I_{size} = O_{size}$	medium	medium	equal	allInstances
findcouples	3	1	2	5	2	$I_{size} > O_{size}$	low	high	equal	set operations
identity	5	0	0	8	0	$I_{size} = O_{size}$	low	low	equal	property access
java2graph	2	0	3	2	0	$I_{size} > O_{size}$	low	medium	smaller	conditionals
java2uml	3	5	2	6	0	$I_{size} > O_{size}$	low	medium	smaller	collections

models, to answer RQ3 and RQ4, we considered seven additional case studies: *java2uml* (reverse engineering Java code into UML models), *java2graph* (creates a graph of dependencies between Java classes), *dblpv1* (query the DBLP database to obtain authors and associated information), *dblp2bibtex* (map DBLP entries to BibTeX records), *identity* (copy transformation of the IMDB database), *findcouples* (extracts actors from IMDB who played together) and *airquality* (queries weather data obtained from sensors). For these transformations, we have models with up to 5.6 millions elements, and 1.2 GB when serialized and stored in disk.

In order to characterize our benchmark, we have considered six dimensions, which are used in Table 2 to summarize the main characteristics of the performance case studies:

- 1) *I/O size* is the expected size of the output model with respect to the size of the input model.
- 2) *Matching cost* is the expected cost of rule matching, in particular the complexity of the rule filters.
- 3) *Rule cost* is the expected cost of rule execution, which is the complexity of the RHS of the bindings.
- 4) *FP size* is the footprint size with respect to the input model size, that is, if the transformation has rules that attempt to match all input elements.
- 5) *OCL* refers to the dominant OCL elements in the transformation: collection intensive, usage of `allInstances`, conditionals, etc.
- 6) Finally, we count the transformation elements in order to have an indication about its “size”: number of matched rules (M_R), number or called or lazy rules (L_R), number of helpers (H), dependencies between rules as the number of bindings for references (B), number of imperative blocks (I).

All transformations except *identity* and *dblp2bibtex* generate output models which are smaller, in terms of number of elements, than their corresponding input models (I/O size). This means that they either rule out many elements in the rule filters (they have a high matching cost) or its footprint with respect to the original meta-model is small (they purposely lack rules to match certain elements). Both *identity* and *dblp2bibtex* exercise the ability of the engine to handle many binding resolutions. Regarding the *matching cost*, we have that *airquality*, *dblpv1* and *dblp2bibtex* have at least one rule filter which traverses collections or accesses all instances of a given type, so it is expected to be costly, whereas the other four transformations have very simple or no rule filters. The execution of cost of *airquality* and *dblpv1* is low because they are “query transformation” whose

target elements have simple initialisations. In this respect *identity* also has simple initialisations, but it has more binding dependencies which makes the post-processing phase time consuming.

6.2.2 Evaluation Metrics

To answer our research questions, we use several metrics depending on the nature of the research question.

Model Comparison Metrics (RQ1): To evaluate research question RQ1, correctness, we need to compare the resulting models after running the code produced by the A2L compiler, with those obtained from the execution of the standard ATL VM. For this, we used EMF Compare, a model comparison framework that compares two models and reports differences between them, such as additions, deletions, and updated elements.

Language Coverage Metrics (RQ2): To evaluate RQ2, we computed the footprints of the ATL transformations with respect to the ATL metamodel. This gives an estimation of how many features are tested by the test cases.

Execution Performance Metrics (RQ3 and RQ4): To evaluate research questions RQ3 and RQ4, we calculated the execution time of the seven case studies listed in Table 2, using a large model as input. We run the experiments on a desktop machine with Ubuntu 18.04 and kernel 5.3.0, a i7-5820K CPU, with 6 cores at 3.30GHz and hyper-threading (12 threads) and 16 GB of RAM, which is expected to be representative of a typical setup of a professional developer. We have used Java 8 (OpenJDK 1.8.0_252) configured with the default options except for the heap size which was set to 8 GB (-Xms=8196m -Xmx=8196m), except for *dblp2bibtex* which was set to 12GB. Each case study is run 10 times with the different engines, discarding the first two runs (as warm up). We perform the 10 executions together in the same VM instance, but after each execution we wait until the garbage collector has released the used memory. We report the average results.

6.3 Result Analysis

6.3.1 Results for RQ1

We executed the 24 regression test cases and compared the output models produced by the standard ATL engine and by A2L. All test cases produced the same results, apart from five of them that could not be directly executed.

We could not compile the *ATL2Problem* transformation with A2L because its typing is too convoluted for AnATLyzer and it cannot properly infer the type of a couple of expressions. Another two unsupported transformations were *DSL2XML* and *KM32DSL*, because they set global variables

TABLE 3
ATL Coverage

Feature	Support	Observations
Matched rules	Yes	
Input elements = 1	Yes	
Input elements > 1	Yes	Via rewriting
Output elements = 1	Yes	
Output elements > 1	Yes	
Rule inheritance	No	
Binding resolution	Yes	
resolveTemp	Yes	Secondary elements resolution
Lazy rules	Yes	
Unique lazy rules	Yes	
Called rules	Yes	Also end/entry point rules
Helpers	Yes	Context and global helpers
OCL		
Collection types	Yes	
Tuple types	Yes	
Collection iterators	Yes	
Iterate operation	Yes	
Reflective operations	No	

using some reflective operations which are currently unsupported by A2L. Similarly, the *SpreadsheetMLSimplified2Trace* transformation uses *global attributes* in a way that adds serious performance penalties to the parallel algorithm. It is worth noting that all of these four transformations could be rewritten so that they could be compiled by A2L. For instance, when there are global variables involved, a strategy could be to use helpers to compute the global information from the source model each time (possibly with some penalty in the execution time). When the problem is related to typing, it might be possible to insert specific annotations (e.g., a dummy version of *oclAsType*) in dedicated places to guide the type inference performed by AnATLyzer [24]. Finally, the ATL and A2L output models of the *XML2DSLModel* transformation were slightly different because this transformation suffers from *child stealing*, i.e., an element is set to a containment reference more than once.

6.3.2 Results for RQ2

The test cases used in our experiments cover 88 percent of the ATL meta-model. The missing 12 percent belongs to metamodel elements used to represent declaration of libraries and query modules, unique lazy rules, entry point rules, rule inheritance and map types. We have separate tests for all of these elements, except query modules and rule inheritance

which are currently not supported. Although we do not foresee any difficulty in supporting rule inheritance in the future, we decided not to implement it at this stage because it is rarely used in practice [25].

We have used the ATL manual as reference for our implementation. Table 3 shows the language features covered by A2L. We support all forms of matched rules (i.e., 1:1, 1:N, N:1 and N:N) and all major ATL features.

6.3.3 Results for RQ3

Table 4 shows the execution times of the case studies in our desktop machine. It compares two versions of the ATL engine (the standard VM and EMFTVM) against four different configurations of A2L: sequential mode without optimizations (seq O-), sequential mode with optimizations (seq O+), parallel (using 12 cores) without optimizations enabled (par O-), and parallel (using 12 cores) with optimizations (par O+) — or simply A2L, since this is the default A2L mode. The figures shown in Table 4 correspond to the average execution time in seconds of eight runs of each transformation. Note that the execution time excludes model loading, but in the case of A2L, it includes the three phases of the algorithm: pre-processing (filling the buffer after model loading), transformation execution, and post-processing. We did not include the parallel ATL [9] implementation (pATL) because it is not supported anymore and we could not make it work reliably.

Table 5 shows the speedups obtained by the different transformation engines and A2L execution modes. For each one, we calculated the individual speedups achieved in all the case studies (excluding the *dblp2bibtex* model transformation in the comparisons between ATL/EMFTVM and A2L because it could not be executed in ATL and EMFTVM). Every cell shows a tuple with the minimum (left) and maximum (right) values, as well as its geometric mean (center), which is the most informative way to represent the average speedup as expected by users [26]. The standard ATL engine is consistently slower than the other options. This is due to its sub-optimal implementation of immutable collections which hinders its ability to handle scenarios with extensive processing of large collections [21]. In particular, the *airquality* test case heavily exercises this feature and shows that EMFTVM is far more efficient (50 versus 1,280 secs). This is because it uses a custom implementation of immutable (and lazy) collections. Moreover, EMFTVM compiles to JVM bytecode on the fly, which provides additional gains. These two

TABLE 4
Performance Comparison

	#elements (millions)	ATL	EMFTVM	A2L: seq O- (1 thread)	A2L: seq O+ (1 thread)	A2L: par O- (12 threads)	A2L: par O+ (12 threads)
airquality	0.1M	1280.40	50.70	6.65	1.32	3.39	0.21
findcouples	3.5M	1765.09	1908.89	1395.66	363.28	299.54	61.60
identity	3.5M	269.00	71.73	11.63	13.87	10.08	10.48
java2graph	4.4M	146.33	2.32	1.65	1.00	1.30	0.97
java2uml	4.4M	49.18	42.05	27.28	4.32	14.59	1.92
dblpv1	5.6M	75.66	48.68	4.68	2.84	2.27	1.05
dblp2bibtex	5.6M	-	-	783.07	11.73	455.15	8.51

Execution on an i7-5820K CPU@3.30GHz - 6 cores (12 threads). Time in seconds.

TABLE 5
Speedups Achieved Between the Transformation Engines: ATL, EMFTVM and the Different A2L Options

Speedups	EMFTVM	A2L seq O-	A2L seq O+	A2L par O-	A2L par O+
ATL	[0.92; 4.64; 63.06]	[1.26; 15.63; 192.6]	[4.86; 39.90; 966.63]	[3.37; 30.15; 377.25]	[25.58; 104.14; 6207.98]
EMFTVM	—	[1.37; 3.37 ; 10.4]	[2.32; 8.59 ; 38.28]	[1.79; 6.49 ; 21.43]	[2.40; 22.42 ; 245.83]
A2L: seq, O-	—	—	[0.84; 4.07 ; 66.76]	[1.15; 1.90 ; 4.66]	[1.11; 9.69 ; 92.02]
A2L: seq, O+	—	—	—	[0.03; 0.47 ; 1.38]	[1.04; 2.38 ; 6.42]
A2L: par, O-	—	—	—	—	[0.96; 5.11 ; 53.48]

features combined makes it normally much faster than the standard ATL engine (except for the *findcouples* example).

A2L obtains significant gains over both engines in all execution modes. Next, we only discuss improvements w.r.t. EMFTVM since they also imply gains over standard ATL.

In the slowest A2L execution mode (A2L seq O-), the speedup w.r.t. EMFTVM is between 1.37x and 10.4x depending on the application. Given that both A2L and EMFTVM target JVM bytecode, this improvement can be explained (in addition to differences in the engine internals) by the fact that A2L does not need to generate code for dynamic checks and model accesses. Table 4 shows that all case studies benefit from this improvement. The *findcouples* case study is the slowest, with a relatively modest performance gain of 1.37x. The main bottleneck is a nested loop which computes the same value in different rule applications. This shortcoming is addressed by our optimizer through automatic caching. Given these results, it can be stated that it is possible to have a significant performance improvement by changing the design of the transformation language from dynamically typed to statically typed. All other A2L execution modes (with optimizations and in parallel) outperform EMFTVM even more, with speedups that range between 2.32x and 38.28x (A2L seq O+), 1.79x and 21.43x (A2L par O-), and even between 2.4x and 245.83x in the case of the A2L default behavior (A2L par O+), depending on the test case. The geometric means indicate expected average speedups of A2L against EMFTVM of 3.37, 8.59, 6.49 and 22.42, respectively.

We also wanted to investigate the individual effects of optimization and parallelization in A2L. This is why we compared the performance of the four possible execution modes of A2L: A2L seq O-, A2L seq O+, A2L par O-, and A2L par O+.

Using as baseline the sequential mode without optimizations (seq O-), the speedup obtained by enabling optimizations (seq O+) ranges between 0.84x and 66.76x depending on the case study. The worse case is the *identity* application, in which the execution time even increases. This is because the partial footprint of the transformation is equal to the source meta-model, and therefore our footprint filtering optimisation only adds overhead to the optimised version but does not reduce the buffer size (there are no other optimisations in that transformation). Therefore, when the partial footprint is equal to the meta-model, it is better not to activate this optimisation. The best case occurs in the *dblp2bibtex* transformation, where optimizations achieve a speedup of 66.76x due to the automatic indexing optimisation. The speedup obtained by the use of optimizations in the *airquality* case study is 5.2x, because this transformation is particularly well-suited for collection optimisations given that it applies several OCL iterators to the set of objects

returned by *allInstances* (i.e., it needs to traverse the complete model). The *findcouples* case study gets a speedup of 3.84x thanks to the automatic caching optimisation. The rest of the transformations obtain speedups of 1.64x, 6.31x and 1.65x, respectively, when optimizations are enabled, with a geometric average of 4.07x.

To analyse the effects of parallelization on the performance of A2L (using the 12 threads in our experimental desktop machine), we compared our baseline A2L seq O- with A2L par O-. The resulting speedup ranges between 1.15x and 4.66x, which represents a significant improvement, although not as noticeable as the one obtained with the optimizations. This is clear in the comparison between optimizations (seq O+) and parallelization (par O-), where the former outperforms the latter 2.14x ($\approx 1.0 / 0.47$) on average.

The speedup obtained by combining optimizations and parallelization ranges between 1.11x and 92.02x depending on the case study, compared to the sequential baseline (seq O-). Again, the *identity* transformation gets the smallest improvement (1.11x) while *dblp2bibtex* obtains the largest gain (92.02x). The rest of the test cases achieve speedups of 32.23x, 22.66x, 1.70x, 14.19, and 4.47x, respectively.

Finally, if we compare the execution times of the two parallel modes (with and without optimizations), the one with optimizations obtain speedups that range between 0.96x (*identity*) and 53.48x (*dblp2bibtex*). This means that optimizations also have a positive effect in the parallel results, mainly because they reduce memory usage, thus reducing the pressure over the garbage collector (i.e., the less garbage collection pauses the better, because a pause stops all threads and causes an important degradation in the speedup).

In summary, A2L achieves significant speedups compared to ATL and EMFTVM. By combining both optimizations and parallel execution, A2L is able to outperform EMFTVM between 2.4x and 245.83x, with an expected average of 22.42x. In this way, A2L execution times for large models become acceptable in all cases, which is an indication that ATL can become a competitive model transformation language when compiled with A2L. In addition, A2L is capable of running transformations such as *dblp2bibtex* that may not be executed with the other engines because of the excessive usage of collection operations over very large collections. In practice, these results also mean a much better developer experience because the transformations can be used as part of the development process without incurring long waiting times.

6.3.4 Results for RQ4

To answer this question, we have executed the case studies using an increasing number of cores, from 1 to 12. Our algorithm has three phases: pre-processing, execution and post-processing, but only the execution phase is expected to have

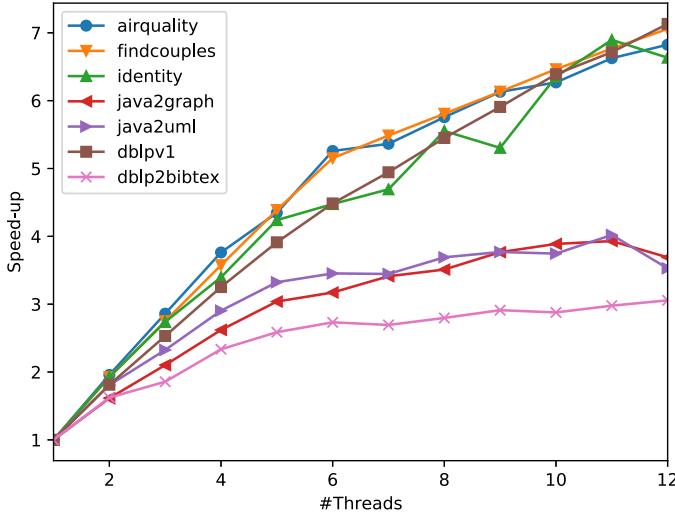


Fig. 7. Speedup of the execution phase of the transformations.

a speedup due to parallelism, because the parallel execution of the post-processing phase only amortizes the cost of accessing the partial traces created by each transformation instance. The speedups obtained in the execution phase for each case study are shown in Fig. 7. We can see how two of the test cases (*airquality* and *findcouples*) scale up very well, close to the theoretical limit, which is 6x with 6 threads. Then, *dblpv1* and *identity* also scale well, with speedups around 4.5x with 6 threads. However, the other three stop scaling soon. This can be caused by a variety of reasons. On the one hand, the rules of these transformations have a modest computation cost which may increase the competition for obtaining the next chunk from the scheduler. On the other hand, parallel computations are highly sensitive to external factors. For instance, a stop-the-world execution of the Java GC would provoke an important decrease in the speedup. Also, in Java and EMF in particular there is little control over the memory layout which may cause an increase in cache misses.

A related question is what is the scalability when taking into account all the phases of the algorithm, or in other words, how much the pre-processing and post-processing phases limit the parallel speedup. Table 6 shows the breakdown of the execution times of the case studies in sequential and in parallel mode with 6 threads. In some transformations, like *airquality*, *java2graph*, *java2uml* and *dblpv1*, the post-processing time is negligible because they do not resolve many bindings. However, transformations with more bindings or whose bindings have many objects in the RHS, incur in large postprocessing times. Therefore, the post-processing phase represents the main bottleneck for scalability in case there are many dependencies among the elements in the rules, because the traces are distributed in the threads and there is a $O(\text{numThreads})$ cost to access them.

Overall, it can be claimed that our algorithm exhibits good scalability, particularly for transformations whose rules have a high computational cost, but relatively few rule dependencies.

6.3.5 Further Findings

Compatibility With ATL. The fact that ATL is a dynamic language poses the challenge of correctly inferring type

TABLE 6
Execution Times for A2L O+

	Sequential			Par. 6 threads		
	Pre.	Exec.	Post.	Pre	Exec.	Post.
airquality	0.01	1.31	0.00	0.01	0.25	0.00
findcouples	0.60	353.65	9.03	0.61	68.64	8.48
identity	0.66	3.88	9.33	0.65	0.87	9.14
java2graph	0.94	0.05	0.01	0.93	0.01	0.01
java2uml	1.10	3.13	0.08	1.06	0.91	0.07
dblpv1	0.73	2.11	0.00	0.74	0.47	0.00
dblp2bibtex	1.88	4.70	5.15	1.84	1.72	5.20

information for its compilation to a typed target language such as Java. In practice this means that we require the transformation to be considered well-typed by AnATLyzer. However, when evaluating RQ1, we already found that *ATL2Prolemand* those transformations using reflective operations could not be compiled. The fact that we have been able to compile the rest of the test cases make us believe that this is a minor issue. Besides, it is always possible to write an equivalent program compatible with A2L.

At the execution level, there might be differences in the order in which the root elements appear in the serialized model, notably when executed in parallel mode. Nevertheless, this is not a real incompatibility since this order is not prescribed by the ATL manual. There are also some differences in the way errors are handled. In particular, ATL signals rule conflicts with a runtime exception, whereas A2L ignores them and relies on AnATLyzer for signalling them at compile time.

Additional Optimizations. In the experiments, we have observed that the post-processing phase is sometimes a bottleneck when a transformation needs to initialize many target references. It would be interesting to find out ways to reduce its impact. At the pre-processing level, it would be possible to filter the input model at loading time using approaches like partial model loading [27]. This would enable the engine to start the transformation without waiting for the model to be fully loaded. Regarding the evaluation of OCL expressions, more specific optimizations for common access patterns can be developed. A2L is prepared for this with a dedicated extension mechanism.

Using GPLs versus ATL. There is a recent trend in the modeling community to use general purpose programming languages (GPL), such as Java or Scala, to develop and execute model transformations. There are several reasons that justify such a decision. For example, (a) IDE features available for GPLs such as live error reporting, quick fixes and debugging have been traditionally missing for transformation languages like ATL; (b) the performance and scalability of ATL, when compared with those of GPL, are much worse; (c) ATL can only deal with EMF models, which is not the format in which many models are stored (for instance, in biology or automotive applications), being rather inefficient, too. The declarative nature of ATL for specifying the transformations and the facility to navigate models using OCL was at the beginning a strong point against Java, but this is not the case any more after Java 8 supported lambda expressions. Scala was also strong in this respect, and its efficient performance has made it a good recent replacement for ATL, too.

However, our goal with A2L is to improve the situation for model transformation languages in general, and ATL in particular, showing that our design can make MT languages competitive again against these GPLs. First, the AnATLyzer tool provides very helpful and useful analyses of the ATL code which are not possible if the transformation is written in a GPL. This, together with the quick-fixing and visualization capabilities that AnATLyzer provides [11], [12] help solving the first shortcoming. Second, we have seen how the performance and scalability of the A2L generated code can be quite acceptable for transforming large models, with competitive execution times. Furthermore, the specific optimizations of A2L for the ATL code and the OCL expressions can provide interesting advantages over GPLs since navigation code could be written in the most readable manner without impact in its performance. A transformation engine also hides the technological complexity related to the modelling platforms. For instance, we internally optimize certain type of accesses to multi-valued features for which we discovered poor performance of the default EMF's internal iterators.² Moreover, the declarative nature of ATL enables the automatic parallelization of the transformation code. If a transformation is written in a GPL, this improvement has to be done manually, which is typically difficult, cumbersome, and error prone.

6.4 Threats to Validity

In this section, we cover the four basic types of validity threats that can affect the validity of our study [28].

6.4.1 Conclusion Validity

Conclusion validity affects the ability to draw correct conclusion about the relations between the treatment and the outcome, i.e., how reasonable the conclusion is. Examples that influence this threat to validity include the choice of sample size, and the measurement of the experiment. In this respect, the correctness and coverage of the compiler have been assessed using the standard regression test suite for ATL [13]. The size of this suite is relatively low, but it covers a large part of the ATL language. Regarding the performance evaluation, there might be specific transformations for which A2L performs worse than ATL and EMFTVM. To mitigate this threat, we have used transformations which exercise different types of scenarios and its analysis shows that they cover different transformation styles. Moreover, the large improvements obtained with A2L makes us confident that improvements will be achieved even in unforeseen scenarios.

6.4.2 Construct Validity

Construct validity refers to the extent to which the experiment setting reflects the theory, i.e., whether the research/tests are well-constructed using established standards and methods. For example, whether the type of samples are representative of the population or not; or whether the number of classes taken reflects common experience. Again, by

² This happens because `EListIterator.hasNext()` invokes `List.size()` instead of having its own `size` property, which prevents JVM optimizations

using a standard test suite for ATL, we aimed at minimizing this threat, since the transformations that comprise the suite provide a representative subset of all kinds of transformations that can be written with ATL. Likewise, the seven additional transformations were carefully selected so that they contain the main features that can have impact on the performance and scalability of the results, and therefore can constitute a representative sample of the kinds of ATL transformations in which we are interested. Moreover, the sizes of the input models were also selected according to the model sizes used in similar tests [9], [20] in order to extract comparable conclusions.

6.4.3 External Validity

This kind of threat limits the ability to generalize the results beyond the experiment context. In this respect, the fact that we have used representative model transformations gives us some confidence that the performance results can be generalized to the rest of the ATL model transformations. Regarding how our results could be applicable to other languages, we believe that for model-to-model rule-based languages like ETL and RubyTL, the applicability would be straightforward, provided that an appropriate type checker is available. QVT-Operational is also rule-based but the rule structure is more explicit, which implies that there might be less opportunities for data-based parallelism at the matched rule level as we do. Nevertheless, the OCL compiler and optimizer could be adapted. In particular, applying these ideas to QVT-Operational is part of our future work.

Moreover, we have used differently sized input models for our study to cover diverse scenarios. However, there may be scenarios where even larger input models are required to be processed by model transformations. We cannot generalize our results for very large models (going beyond 10 millions of model elements such as present in the Train Benchmark [29] for continuous model queries) based on our performed study and leave this as subject to future work.

6.4.4 Internal Validity

Internal validity checks whether the test or instrument measures what it is supposed to. This threat can affect the independent variable with respect to causality. That is, the results may indicate a causal relationship, although there is none. In this respect, the optimizations at the transformation and OCL expression level, as well as the parallelization algorithm used to execute the transformation have proved to be effective to significantly improve its performance. The speedup results obtained independently for each separated feature seem to corroborate our hypotheses. The compiler was designed so that these features could be independently enabled or disabled, precisely to facilitate these kinds of analyses.

7 RELATED WORK

With respect to the contribution of this paper, namely the efficient execution of model transformations by data parallelism and optimizations based on static analysis information, we identify three lines of related work. First, we discuss general approaches for speeding up model transformations

which apply smart execution strategies for different contexts. Second, we discuss research dedicated to the evaluation of model transformation engines. Third, we discuss approaches focusing on the parallel execution of model transformations.

7.1 Model Transformation Execution Strategies

In this paper, we have investigated the creation of output models from input models by following the classical batch transformation strategy, i.e., the transformation run is considered as a fresh one which is creating a new output model from scratch for a given input model [30]. In addition to this strategy, there are several other strategies for executing model transformations in particular contexts. First, incremental transformations [31], [32], [33] have been proposed for cases where output models are already available from previous transformation runs. In such cases, the transformation engine may only propagate the changes of the input models to the existing output models. Second, lazy transformations [34] have been proposed for cases where only subsets of output models are needed in a first step. In such cases, the transformation engine only creates these subsets in a first phase, while other elements are created just-in-time when they are requested. Third, streaming transformations [35] have been proposed for transforming so-called streaming models, i.e., models are considered as an open and continuous stream of model elements opposed to a fixed set of elements given at once in a closed input model. Fourth, patch transformations [36] are used in cases where transformations are evolving and the existing output models have to be migrated to newer versions of the transformations.

All of the mentioned execution strategies mainly focus on not having to re-execute the whole transformation by providing some kind of reactivity, e.g., to changes in the input models and model transformations or read access to the output models. Needless to say, identifying the transformation statements that are unnecessary to be re-executed is probably the best optimization for running transformations outside the classical batch transformation area.

While we have focused on batch transformations in this work, we do not see any major obstacle to adopt the presented optimizations also for other kind of transformation execution strategies used for the discussed contexts. On the contrary, we see the introduction and evaluation of the presented optimization techniques for these additional transformation strategies as an interesting line for future research.

In addition to the already mentioned model transformation execution strategies, in recent years there have been also major efforts in optimizing the execution of full batch transformations by following the distributed execution paradigm for transforming large models which are fragmented over different computing nodes. For instance, MapReduce has been exploited for running ATL transformations on distributed models [9]. Camargo *et al.* presented a data-centric framework for distributed model transformations reusing the Bloom platform [37]. In one of our previous work, we have employed Linda-based tuple spaces to run distributed model transformations [38]. Finally, graph queries (comparable to the matching part of model transformation rules such as the in-pattern of ATL rules) are executed for large models in a distributed manner by combining incremental graph search techniques and cloud computing technologies [39] as well as

by providing dedicated optimization concepts such as node sharing for efficiently executing partially redundant code fragments [40].

In this paper, we did not consider the distribution aspect of models but rather focused on supporting the scenario of running model transformations directly on developer-scale machines having the models stored in-memory. Thus, we consider investigations on the distribution aspect in combination with our presented optimization techniques as a subject for future work.

7.2 Performance Evaluations of Transformation Engines

There is work on evaluating the performance of state-of-the-art model transformation engines. For instance, Amstel *et al.* [41] compared the runtime performance of transformations written in ATL and in QVT, finding that the standard ATL engine outperforms the available QVT engines. From these works we can conclude that A2L may also outperform existing QVT engines, although this needs to be confirmed with the more recent versions of the QVT engines. In [42], several implementation variants of the ATL language, e.g., using either imperative constructs or declarative constructs, of the same transformation scenario have been considered and their different runtime performance was compared. However, both mentioned works [41], [42] only consider the traditional sequential execution engines.

A performance evaluation of the standard ATL engines with respect to running the transformations within database technologies is performed in [43]. In our work, we currently do not use a dedicated database technology for storing the models to be transformed. However, the presented optimization techniques for ATL rules and OCL expressions based on static analysis information may also help in cases where dedicated database technologies are used to produce even more efficient code that is subsequently executed inside the databases. An interesting future work line in this respect is to study the combination of the presented approach of this paper with current advances achieved for graph databases (cf. [44] for a survey) as well as investigations of data-parallel operations inside databases (cf. [45] for a comparison of different execution models) to allow even more efficient transformations of very large models.

Finally, we also like to mention the Transformation Tool Contest (TTC)³ that is now running for 13 years. Every year, there are dedicated transformation cases developed and submitted by the community for the community. In previous years, there have also been some transformation cases that focused on performance and scalability of model transformations. In particular, worth to mention are the Train Benchmark case, the Movie Database case and the Program Comprehension case. We have reused and partially adapted the Movie Database case and the Program Comprehension case for our evaluation, as they represent outplace batch transformations that we are considering in this paper.

7.3 Parallel Model Transformations

In recent years, there is an increased interest in parallelizing different types of model transformations which resulted in

3. <https://www.transformation-tool-contest.eu>

dedicated extensions for graph transformation languages, model management languages based on Epsilon, and also ATL which we discuss in the following.

First, there is work in the field of graph transformations where multi-core platforms are used for parallel execution of graph transformation rules [46], [47]. In these papers, specific focus is put on the parallel execution of the matching phase of the left-hand sides of graph transformation rules, which is considered to be the most expensive part. Another work exploits the Bulk Synchronous Parallel model for executing transformations based on the Henshin graph transformation framework [48]. To make use of the Bulk Synchronous Parallel model, the Henshin graph transformation rules are compiled to Apache Giraph.

Second, efforts have been made to speed up different types of model management programs (which can be considered as specific kinds of model transformations) for the Epsilon framework [19], [49] available in Eclipse. In particular, parallel execution support for the different model management languages provided by Epsilon has been presented in [50], [51]. For instance, the authors provide data and rule parallelization approaches for the Epsilon Validation Language (EVL). However, the use of static analysis information for automatic performance improvement is only mentioned as future work.

Third, there have been pioneering approaches for the parallel execution of ATL transformations. In previous work, we have presented LinTra [20], an approach for running ATL transformations on Linda-based platforms following the data-based parallelization approach. Tisi *et al.* [10] presented another approach for the parallel execution of ATL transformations, using a task-based approach for parallelization, as already mentioned in Section 2.3.

7.4 Synopsis

While there are several approaches available for speeding up model transformations, the scalability of model transformations is still considered as a major challenge in MDE [52]. We are not aware of any approach that uses the information from static analyses to find improved (parallel) execution strategies for model transformations. Thus, we are confident that in this paper we provide an important cornerstone for scalable and high-performance execution of model transformations, which may also help to improve other model transformation engines beyond ATL. This line of research is considered critical to the long-term success of model transformation tools, as a recent survey has revealed [30].

8 CONCLUSION

This paper has presented A2L, a compiler for the ATL model transformation language that aims at achieving efficient transformations of large models. Improved performance and scalability is accomplished by two of the main features of A2L: the use of static information to achieve effective optimizations on both rule execution and the evaluation of the OCL expressions; and the use of data parallelism in the algorithm that implements and executes the transformation. The results show that A2L produces large performance gains with respect to existing ATL engines in both sequential and parallel modes. The figures presented

in this paper should be a baseline for the expected performance of future transformation languages.

Our work can be extended along different lines of research. First, further optimizations can be pursued, mainly for domain-specific transformations where the semantics of the particular domains can be taken into account. Second, we plan to study how our work can be generalised to other model transformation languages, notably QVT-Operational. To this end we aim at designing an intermediate representation from which it is easy to reuse most of the infrastructure. Implementation-wise, it would be interesting to create an intermediate representation in which it is easier to perform rewritings and make our optimizations composable. Handling models from different modeling platforms, beyond EMF, or even stored as plain Java objects, is another research line we are also working on, with the goal of widening the usability of A2L. This line of research may also require the development of a dedicated debugger which allows the observation and control of the executing transformations directly on the ATL code level [53]. Finally, we would like to explore the possibilities of creating a streaming transformation engine on top of A2L as well as of employing emerging database technologies for executing our transformations for very large models.

ACKNOWLEDGMENTS

This work was partially funded by Spanish Research Projects PGC2018-094905-B-I00, TIN2015-73968-JIN (AEI/FEDER/UE), a Ramón y Cajal 2017 research grant, and TIN2016-75944-R. Furthermore, this work was partially supported and funded by the Austrian Federal Ministry for Digital and Economic Affairs, the National Foundation for Research, Technology and Development, and by the FWF under the Grant Numbers P28519-N31 and P30525-N31.

REFERENCES

- [1] M. Brambilla, J. Cabot, and M. Wimmer, *Model-Driven Software Engineering in Practice*, 2nd ed. San Rafael, CA, USA: Morgan & Claypool, 2017.
- [2] B. Selic, "What will it take? A view on adoption of model-based methods in practice," *Softw. Syst. Model.*, vol. 11, no. 4, pp. 513–526, 2012.
- [3] D. S. Kolovos *et al.*, "A research roadmap towards achieving scalability in model driven engineering," in *Proc. Workshop Scalability Model Driven Eng.*, 2013, pp. 2:1–2:10.
- [4] E. Guerra, J. de Lara, D. S. Kolovos, R. F. Paige, and O. M. dos Santos, "Engineering model transformations with transML," *Softw. Syst. Model.*, vol. 12, no. 3, pp. 555–577, 2013.
- [5] F. Jouault, F. Allilaire, J. Bézivin, and I. Kurtev, "ATL: A model transformation tool," *Sci. Comput. Program.*, vol. 72, no. 1–2, pp. 31–39, 2008.
- [6] OMG, MOF QVT Final Adopted Specification, Object Management Group, 2005, *OMG doc. ptc/05-11-01*.
- [7] L. Burgueño, J. Cabot, and S. Gérard, "The future of model transformation languages: An open community discussion," *J. Object Technol.*, vol. 18, no. 3, pp. 7:1–11, Jul. 2019.
- [8] J. S. Cuadrado, E. Guerra, and J. de Lara, "Static analysis of model transformations," *IEEE Trans. Softw. Eng.*, vol. 43, no. 9, pp. 868–897, Sep. 2017.
- [9] A. Benelallam, A. Gómez, M. Tisi, and J. Cabot, "Distributing relational model transformation on MapReduce," *J. Syst. Softw.*, vol. 142, pp. 1–20, 2018.
- [10] M. Tisi, S. M. Pérez, and H. Choura, "Parallel execution of ATL transformation rules," in *Proc. Int. Conf. Model Driven Eng. Languages Syst.*, 2013, pp. 656–672.

- [11] J. S. Cuadrado, E. Guerra, and J. de Lara, "Quick fixing ATL transformations with speculative analysis," *Softw. Syst. Model.*, vol. 17, no. 3, pp. 779–813, 2018.
- [12] J. S. Cuadrado, E. Guerra, and J. de Lara , "AnATLyzer: An advanced IDE for ATL model transformations," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.: Companion*, 2018, pp. 85–88.
- [13] F. Jouault, "Regression tests for the ATL virtual machine," 2013. [Online]. Available: https://wiki.eclipse.org/ATL_VM_Testing
- [14] C. Eastman, P. Tiecholz, R. Sacks, and K. Liston, *BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers and Contractors*, 2nd ed. Hoboken, NJ, USA: Wiley, 2011.
- [15] Y. Dajsuren and M. van den Brand, Eds., *Automotive Systems and Software Engineering – State of the Art and Future Trends*. Berlin, Germany: Springer, 2019.
- [16] J. Troya, H. Brunelière, M. Fleck, M. Wimmer, L. Orue-Echevarria , and J. Gorroño-goitia, "ARTIST: Model-Based Stairway to the Cloud," in *Proc. Projects Showcase @ STAF*, 2015, pp. 1–8.
- [17] Object Management Group, Object Constraint Language (OCL) Specification. Version 2.2, Feb. 2010, *OMG Document formal/2010-02-01*.
- [18] D. Wagelaar, M. Tisi, J. Cabot, and F. Jouault, "Towards a general composition semantics for rule-based model transformation," in *Proc. Int. Conf. Model Driven Eng. Languages Syst.*, 2011, pp. 623–637.
- [19] D. Kolovos, L. Rose, R. Paige, and A. García-Domínguez, *The Epsilon Book. Eclipse*, 2010. [Online]. Available: <https://www.eclipse.org/epsilon/doc/book/>
- [20] L. Burgueño, M. Wimmer, and A. Vallecillo, "A Linda-based platform for the parallel execution of out-place model transformations," *Inf. Softw. Technol.*, vol. 79, pp. 17–35, 2016.
- [21] J. S. Cuadrado, F. Jouault, J. G. Molina, and J. Bézivin, "Optimization patterns for OCL-based model transformations," in *Proc. of Workshops and Symposia at MODELS*, vol. 5421. Berlin, Germany: Springer, 2008, pp. 273–284.
- [22] A. S. Lee, "A scientific methodology for MIS case studies," *MIS Quarterly*, vol. 13, no. 1, pp. 33–50, 1989.
- [23] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empir. Softw. Eng.*, vol. 14, no. 2, pp. 131–164, 2009.
- [24] "AnATLyzer tutorial," 2017. [Online]. Available: <https://github.com/jesusc/anatlyzer-models17>
- [25] A. Kusel, J. Schönböck, M. Wimmer, G. Kappel, W. Retschitzegger, and W. Schwinger, "Reuse in model-to-model transformation languages: Are we there yet?" *Softw. Syst. Model.*, vol. 14, no. 2, pp. 537–572, 2015.
- [26] T. Hoefer and R. Belli, "Scientific benchmarking of parallel computing systems: Twelve ways to tell the masses when reporting performance results," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, pp. 73:1–73:12.
- [27] R. Wei, D. S. Kolovos, A. Garcia-Domínguez, K. Barmpis, and R. F. Paige, "Partial loading of XMI models," in *Proc. ACM/IEEE 19th Int. Conf. Model Driven Eng. Languages Syst.*, 2016, pp. 329–339.
- [28] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, and B. Regnell, *Experimentation Softw. Eng.*, 2012.
- [29] G. Szárnyas, B. Izsó, I. Rath, and D. Varró, "The train benchmark: Cross-technology performance evaluation of continuous model queries," *Softw. Syst. Model.*, vol. 17, no. 4, pp. 1365–1393, 2018.
- [30] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, and D. Varró, "Survey and classification of model transformation tools," *Softw. Syst. Model.*, vol. 18, no. 4, pp. 2361–2397, 2019.
- [31] T. L. Calvar, F. Jouault, F. Chhel, and M. Clavreul, "Efficient ATL incremental transformations," *J. Object Technol.*, vol. 18, no. 3, pp. 2:1–17, 2019.
- [32] A. Razavi and K. Kontogiannis, "Partial evaluation of model transformations," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 562–572.
- [33] D. Varró, G. Bergmann, Á. Hegedüs, Á. Horváth, I. Ráth, and Z. Ujhelyi, "Road to a reactive and incremental model transformation platform: three generations of the VIATRA framework," *Softw. Syst. Model.*, vol. 15, no. 3, pp. 609–629, 2016.
- [34] M. Tisi, S. M. Perez, F. Jouault, and J. Cabot, "Lazy execution of model-to-model transformations," in *Proc. Int. Conf. Model Driven Eng. Languages Syst.*, 2011, pp. 32–46.
- [35] J. S. Cuadrado and J. de Lara , "Streaming model transformations: Scenarios, challenges and initial solutions," in *Proc. Int. Conf. Theory Practice Model Transformations*, 2013, pp. 1–16.
- [36] A. Bergmayr, J. Troya, and M. Wimmer, "From out-place transformation evolution to in-place model patching," in *Proc. 29th ACM/IEEE Int. Conf. Softw. Eng.*, 2014, pp. 647–652.
- [37] L. C. Camargo and M. D. D. Fabro, "Applying a data-centric framework for developing model transformations," in *Proc. 34th ACM/SIGAPP Symp. Appl. Comput.*, 2019, pp. 1570–1573.
- [38] L. Burgueño, "Concurrent and distributed model transformations based on linda," in *Proc. Doctoral Symp.*, 2013, pp. 9–16.
- [39] G. Szárnyas, B. Izsó, I. Ráth, D. Harmath, G. Bergmann, and D. Varró, "IncQuery-D: A distributed incremental model query framework in the cloud," in *Proc. Int. Conf. Model Driven Eng. Languages Syst.*, 2014, pp. 653–669.
- [40] G. Bergmann, "Incremental model queries in model-driven design," Ph.D. dissertation, Budapest Univ. Technol. Economics, Hungary, 2013.
- [41] M. van Amstel, S. Bosems, I. Kurtev, and L. F. Pires, "Performance in model transformations: Experiments with ATL and QVT," in *Proc. Int. Conf. Theory Practice Model Transformations*, 2011, pp. 198–212.
- [42] M. Wimmer, S. M. Perez, F. Jouault, and J. Cabot, "A catalogue of refactorings for model-to-model transformations," *J. Object Technol.*, vol. 11, no. 2, pp. 2:1–40, 2012.
- [43] G. Daniel, F. Jouault, G. Sunyé, and J. Cabot, "Gremlin-ATL: A scalable model transformation framework," in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 462–472.
- [44] M. Besta et al., "Demystifying graph databases: Analysis and taxonomy of data organization, system designs, and graph queries," *CoRR*, vol. abs/1910.09017, 2019. [Online]. Available: <http://arxiv.org/abs/1910.09017>
- [45] T. Kersten, V. Leis, A. Kemper, T. Neumann, A. Pavlo, and P. A. Boncz, "Everything you always wanted to know about compiled and vectorized queries but were afraid to ask," *Proc. VLDB Endowment*, vol. 11, no. 13, pp. 2209–2222, 2018.
- [46] G. Bergmann, I. Ráth, and D. Varró, "Parallelization of graph transformation based on incremental pattern matching," *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, vol. 18, pp. 1–15, 2009.
- [47] G. Imre and G. Mezei, "Parallel graph transformations on multicore systems," in *Proc. Int. Conf. Multicore Softw. Eng. Perform. Tools*, 2012, pp. 86–89.
- [48] C. Krause, M. Tichy, and H. Giese, "Implementing graph transformations in the bulk synchronous parallel model," in *Proc. Int. Conf. Fundam. Approaches Softw. Eng.*, 2014, pp. 325–339.
- [49] D. S. Kolovos, R. F. Paige, and F. A. C. Polack, "The epsilon transformation language," in *Proc. Int. Conf. Theory Practice Model Transformations*, 2008, pp. 46–60.
- [50] S. Madani, D. S. Kolovos, and R. F. Paige, "Towards optimisation of model queries: A parallel execution approach," *J. Object Technol.*, vol. 18, no. 2, pp. 3:1–21, 2019.
- [51] S. Madani, D. S. Kolovos, and R. F. Paige, "Parallel model validation with epsilon," in *Proc. Eur. Conf. Model. Foundations Appl.*, 2018, pp. 115–131.
- [52] A. Bucciarone, J. Cabot, R. F. Paige, and A. Pierantonio, "Grand challenges in model-driven engineering: An analysis of the state of the research," *Softw. Syst. Model.*, vol. 19, no. 1, pp. 5–13, 2020.
- [53] E. Bousse and M. Wimmer, "Domain-level observation and control for compiled executable DSLs," in *Proc. ACM/IEEE 22nd Int. Conf. Model Driven Eng. Languages Syst.*, 2019, pp. 150–160.



Jesús Sánchez Cuadrado is a Ramón y Cajal researcher at Universidad de Murcia. Previously he was an associate professor with Universidad Autónoma de Madrid. His research is focused on Model Driven Engineering (MDE) topics, notably model transformation languages, meta-modelling and domain specific languages. On these topics, he has published several articles in journals and peer-reviewed conferences, and developed several tools. For more information, please visit <http://sanchezcuadrado.es>.



Loli Burgueño is a postdoctoral researcher and lecturer with the Open University of Catalonia (Spain) and CEA LIST (France). Her research interests focus on model-driven engineering, in particular the performance, scalability and testing of model transformations, the modeling of uncertainty in software models for its use in the Industry 4.0 and the integration of artificial intelligence techniques into modeling tools and processes. For more information, please visit <https://som-research.uoc.edu/loli-burgueno/>.



Antonio Vallecillo is full professor of software engineering with the University of Málaga, Spain, where he leads the Atenea research group dedicated to systems modeling and analysis. His research interests include open distributed processing, model-based software engineering, and software quality. For more information, please visit <http://www.lcc.uma.es/~av>.



Manuel Wimmer is full professor leading the Institute of Business Informatics—Software Engineering with the Johannes Kepler University Linz, and he is head of the Christian Doppler Laboratory CDL-MINT. His research interests comprise foundations of model engineering techniques as well as their application in domains such as tool interoperability, legacy tool modernization, model versioning and evolution, and industrial engineering. For more information, please visit <https://www.se.jku.at/manuel-wimmer>.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

Why My App Crashes? Understanding and Benchmarking Framework-Specific Exceptions of Android Apps

Ting Su^{ID}, Lingling Fan^{ID}, Sen Chen^{ID}, Yang Liu^{ID}, Lihua Xu, Geguang Pu^{ID}, and Zhendong Su

Abstract—Mobile apps have become ubiquitous. Ensuring their correctness and reliability is important. However, many apps still suffer from occasional to frequent crashes, weakening their competitive edge. Large-scale, deep analyses of the characteristics of real-world app crashes can provide useful insights to both developers and researchers. However, such studies are difficult and yet to be carried out — this work fills this gap. We collected 16,245 and 8,760 unique exceptions from 2,486 open-source and 3,230 commercial Android apps, respectively, and observed that the exceptions thrown from Android framework (termed “framework-specific exceptions”) account for the majority. With one-year effort, we (1) extensively investigated these framework-specific exceptions, and (2) further conducted an online survey of 135 professional app developers about how they analyze, test, reproduce and fix these exceptions. Specifically, we aim to understand the framework-specific exceptions from several perspectives: (i) their characteristics (e.g., manifestation locations, fault taxonomy), (ii) the developers’ testing practices, (iii) existing bug detection techniques’ effectiveness, (iv) their reproducibility and (v) bug fixes. To enable follow-up research (e.g., bug understanding, detection, localization and repairing), we further systematically constructed, *DroidDefects*, the first comprehensive and largest benchmark of Android app exception bugs. This benchmark contains 33 *reproducible* exceptions (with test cases, stack traces, faulty and fixed app versions, bug types, etc.), and 3,696 *ground-truth* exceptions (real faults manifested by automated testing tools), which cover the apps with different complexities and diverse exception types. Based on our findings, we also built two prototype tools: Stoat+, an optimized dynamic testing tool, which quickly uncovered three previously-unknown, fixed crashes in Gmail and Google+; ExLocator, an exception localization tool, which can locate the root causes of specific exception types. Our dataset, benchmark and tools are publicly available on <https://github.com/tingsu/droiddefects>.

Index Terms—Mobile applications, android applications, empirical study, exception analysis, software testing, bug reproducibility

1 INTRODUCTION

MOBILE apps have become ubiquitous recently. For example, Google Play, Google’s official Android app market, contains over three million apps; over 50,000 apps are continuously published on it [1] each month. To ensure the competitive edge, app developers strive to deliver high-

quality apps [2]. One of their primary concerns is to prevent fail-stop errors (i.e., app crashes) from releases [3], [4].

1.1 Motivations

In industry, many testing frameworks (e.g., Robotium [5], Appium [6]) and static checking tools (e.g., Lint [7], FindBugs [8]) are available [9], [10] to improve app quality. However, many released apps still suffer from crashes. Two recent studies [11], [12] discovered hundreds of previously unknown crashes in popular and well-tested commercial apps. This may make developers wondering “why my app crashes?”. Researchers have proposed a number of testing techniques and tools [11], [12], [13], [14], [15], [16], [17], [18], [19], [20], [21], [22], [23], [24] to reveal app crashes. However, none of them investigated the root causes of these crashes. Without the answer to this question, developers may not know how to effectively avoid and fix these bugs. By analyzing the 272,629 issues mined from 2,174 Android apps hosted on GitHub and Google Code, we find nearly 40 percent of the reported crash issues remain open/unfixed (filtered by the keywords “crash” or “exception” in their issue descriptions). This situation could compromise the app quality, considering these issues may probably lead to fail-stop errors after releasing. Even worse, due to the lack of understanding of root causes, the follow-up research, e.g., bug detection, localization and repairing, might be constrained. For example, existing fault localization [25] and repairing [26],

- Ting Su is with the School of Software Engineering, East China Normal University, Shanghai, China, and also with the Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland. E-mail: tsuletgo@gmail.com.
- Lingling Fan is with the College of Cyber Science, Nankai University, Tianjin, China, and also with the Nanyang Technological University, Singapore 639798. E-mail: llfan@ntu.edu.sg.
- Sen Chen is with the College of Intelligence and Computing, Tianjin University, Tianjin, China, and also with the Nanyang Technological University, Singapore 639798. E-mail: chensen@ntu.edu.sg.
- Yang Liu is with the School of Computer Science and Engineering, Nanyang Technological University, Singapore 639798. E-mail: yangliu@ntu.edu.sg.
- Lihua Xu is with the Department of Computer Science and Engineering, New York University Shanghai, Shanghai 200122, China. E-mail: lihua.xu@nyu.edu.
- Geguang Pu is with the School of Software Engineering, East China Normal University, Shanghai 200062, China. E-mail: ggpu@sei.ecnu.edu.sg.
- Zhendong Su is with the Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland. E-mail: zhendong.su@inf.ethz.ch.

Manuscript received 4 Feb. 2020; revised 29 May 2020; accepted 13 July 2020. Date of publication 31 July 2020; date of current version 18 Apr. 2022.

(Corresponding authors: Lingling Fan and Geguang Pu.)

Recommended for acceptance by K. Sen.

Digital Object Identifier no. 10.1109/TSE.2020.3013438

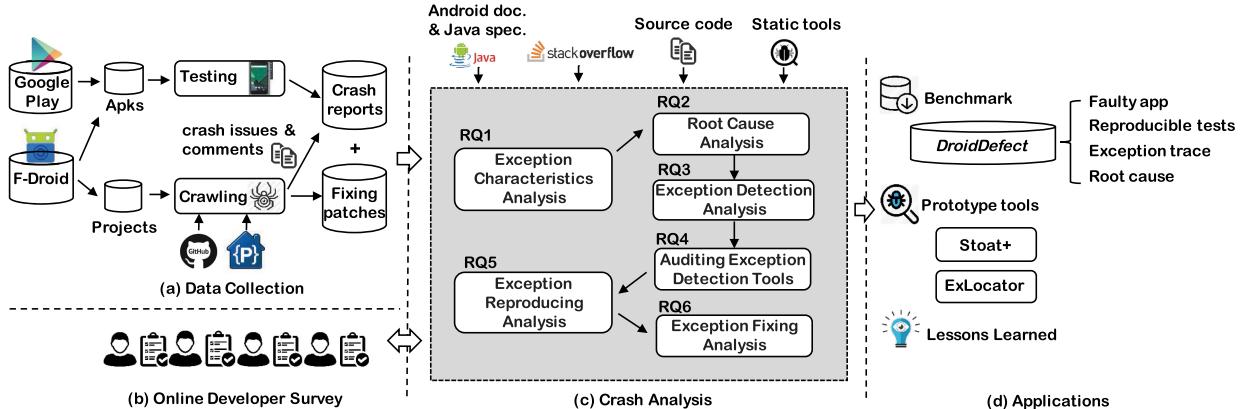


Fig. 1. Overview of our study.

[27] tools for Android apps are limited to a small set of trivial crash bugs. Thus, it is important to conduct such a study — characterizing the root causes from a large-scale, diverse set of real-world app crashes, and investigating how to effectively detect, reproduce, and fix them. However, such a study is difficult and yet to be carried out, which has motivated this work.

Routinely, when an app crashes, the Android runtime system will dump an exception trace that provides certain clues of the issue (e.g., the exception type, message, and the stack of invoked methods). Based on the architecture layer throwing the exception, each exception can be classified into one of three categories — *application exception*, *framework exception*,¹ and *library exception* (cf. Section 2.1). Specifically, we find framework exceptions account for the majority of app crashes, affecting over 75 percent of the projects (cf. Section 3). Thus, we focus on analyzing framework exceptions, and also brief the other two exception types (cf. Section 3.1).

1.2 Challenges

We face three key challenges in this study. (1) The first is the *lack of comprehensive dataset*. To enable crash analysis, we need a comprehensive set of crashes from many apps. Ideally, each crash is associated with the exception trace, the buggy code version, the bug-triggering test, and the patch (if exists). However, to our knowledge, no such dataset exists. Despite open-source project hosting platforms (e.g., GitHub and Google Code) maintain issue repositories, we find only a small set of crash issues (~ 16%) are accompanied with exception traces. Among them, only a small fraction has clear reproduction steps (with target app versions and environment); even if the issue is closed, the faulty code version may not be linked with the fixed one. (2) The second concerns *difficulties in crash analysis*. Analyzing crashes requires deep knowledge of app logic, Android framework, and even third-party libraries. However, no reliable tool exists that can help our analysis. As a result, the crash analysis requires considerable human expertise and efforts. (3) The third is the *validation of analysis results and findings*. To reduce the threats to validity, we need to consider diverse categories/types of

apps, and cross-check our findings by referring to the developers' expertise and experience.

To achieve this study, we made substantial efforts in several aspects. Fig. 1 shows the overview of our study.

1.3 Data Collection and Online Survey

We collected 16,245 unique exception traces from 2,486 open-source (F-Droid) apps as our analysis data (see Fig. 1a) by (1) mining the issue repositories; and (2) applying the three state-of-the-art app testing tools (Monkey [28], Sapienz [11], and Stoat [12]). We also run the three testing tools on 3,230 Google Play apps, and collected 8,760 unique exception traces, to complement our analysis data. Moreover, we conducted an online survey, and received 135 app developers' responses about how they analyze, test, reproduce and fix exception bugs to cross-validate our analysis results and gain more insights (see Fig. 1b).

1.4 Crash Analysis

We aim to answer the following research questions.

- RQ1 (Exception Characteristics): *What are the characteristics of these exceptions, e.g., exception categories, distributions, and locations of manifestation?*
- RQ2 (Root Causes): *What are the root causes of framework exceptions? What are the difficulties app developers face when analyzing them?*
- RQ3 (Exception Detection): *What tools are commonly used by developers to detect exception bugs? Are they satisfactory?*
- RQ4 (Auditing Tools): *How effective is the state-of-the-art bug detection techniques in manifesting framework exceptions?*
- RQ5 (Exception Reproduction): *How is the reproducibility of app exception bugs? Are there any difficulties of reproducing?*
- RQ6 (Exception Fixing): *How do developers fix framework exceptions? Are there any difficulties app developers face?*

Through these questions, we find framework exceptions account for the majority in both open-source and commercial apps. They have lower issue closing rate² (only 53 percent),

1. For brevity, we use *framework exception* to indicate *framework-specific exception*, which can be any exception thrown from Android framework.

2. The percentage of how many issues has been closed by developers.

compared with application exceptions (67 percent). Through careful analysis, we distilled 11 common fault categories, which have not been well-investigated before (cf. Section 3.2).

Informed by the developer survey, we further audited existing automated bug detection tools on framework exceptions (cf. Section 3.4). We find dynamic testing tools can reveal framework exceptions, but are still less effective on certain fault categories. Their testing strategies have a big impact on the detection ability. In addition, these testing tools have low reproducing rates (cf. Section 3.5). We also find most exceptions can be fixed by five common practices with small patches (fewer than 20 code lines), but developers face several difficulties in fixing (cf. Section 3.6).

1.5 Applications

Based on our study, we made several applications: (1) We constructed *DroidDefects*, the first comprehensive and largest benchmark of Android app exception bugs. It contains 33 reproducible and 3,696 ground-truth exception bugs, and covers diverse exception types, root causes, app complexities and categories, and relevant bug information. It can help follow-up research, e.g., bug understanding, detection, localization, prediction, and patch generation for Android apps. (2) We optimized *Stoat*,³ a GUI testing tool, by integrating a number of testing strategies, which quickly revealed three previously unknown bugs in Gmail and Google+. (3) We built *ExLocator*,⁴ an exception localization tool, which can help localize the root causes of specific exception types. (4) We also demonstrated the possibility of enhancing static checking and mutation testing for Android apps.

1.6 Contributions

To summarize, we made the following contributions:

- We conducted the first large-scale study to investigate exception bugs (framework exceptions in particular) of Android apps, and identified 11 common fault categories. The results provide useful insights for both researchers and developers.
- Our study evaluated the state-of-the-art bug detection techniques, reviewed the reproducibility of these exceptions, and investigated common fixing practices. The findings motivate more effective bug detection, reproduction, and fixing techniques.
- We conducted an online survey to understand how developers analyze, test, reproduce and fix crashes. This survey gains more insights from the developers' experiences, and also validates our analysis results.
- We constructed *DroidDefects*, the first comprehensive and largest benchmark of Android app exceptions, to enable follow-up research. We built two prototype tools *Stoat+* and *Exlocator* to improve bug detection and debugging, and summarized several lessons learned.

In our prior work [29], we investigated framework-specific exceptions in Android apps. In this journal version, we have made substantial extensions: (1) We additionally analyzed

8,760 exception bugs from 3,230 commercial apps from Google Play. It provided more observations on the characteristics of exception bugs, and validated the generability of our conclusion (Sections 2.2 and 3.1). (2) We conducted an online survey among 135 Android app developers. It provided more insights from the developers' experiences and complemented our analysis results (Sections 2.3, Sections 3.1, 3.2, 3.3, 3.4, 3.5, and 3.6 (RQ1~RQ6)). (3) We revisited our research questions (i.e., RQ1, RQ2, RQ4 and RQ6) in depth and analyzed together with the results from the online survey. For example, we additionally investigated the difficulties the developers face when analyzing root causes, the common fix practices, and the reasons of library exceptions, etc. (4) We additionally studied two new research questions, i.e., the testing practices of exception bugs by developers (RQ3 in Section 3.3), and the reproducibility of exception bugs from the perspectives of both developers and testing tools (RQ5 in Section 3.5). It reveals the unsatisfactory points of existing testing tools, and the challenges that app developers and state-of-the-art tools face in reproducing exceptions, which have not yet been explored before. (5) We constructed the benchmark repository *DroidDefects*. It now contains 33 reproducible and 3,696 ground-truth exception bugs and the utility program for facilitating other research work. For each bug, we provided the faulty code version, the reproducible test, the exception trace and the explanation of root cause. *DroidDefects* can serve follow-up research work (Section 4). (6) We further illustrated more application domains of our study. We also extended our analysis on the empirical study and analysis results, and concluded with several lessons learned that were not identified before (Section 5 and 6). Importantly, our dataset, benchmark and tools were made publicly available at <https://github.com/tingsu/droiddefects>.

2 PRELIMINARY AND STUDY PREPARATION

2.1 Android Exception Model

The architecture of Android platform is composed of four layers, i.e., application, framework, library and Linux kernel. Android apps run at the application layer. The Android framework APIs form the building blocks of apps. To provide different functionalities and services, Android reuses a number of libraries (e.g., Apache, SSL, OpenGL). When an app crashes, a (Java) exception will be thrown from one of these three layers, which corresponds to application, framework or library exception.

Android apps (implemented in Java) inherit the exception model of Java, which has three kinds of exceptions. (1) *RuntimeException*, the exceptions that are thrown during the normal operation of the Java Virtual Machine when the program violates the semantic constraints (e.g., null-pointer dereferences, divided-by-zero errors). (2) *Error*, which represents serious problems that a reasonable application should not try to catch (e.g., *OutOfMemoryError*). (3) *CheckedException* (all exceptions except (1) and (2)), these exceptions are required to be declared in a method or constructor's throws clause (statically checked by compilers), and indicate the conditions that a reasonable client program might want to catch. The programmers are responsible to handle *RuntimeException* and *Error* by themselves at runtime.

Fig. 2 shows an example of *RuntimeException*. The bottom part represents the *root exception*, i.e., *NumberFormatException*,

³ Stoat is available at <https://github.com/tingsu/stoat>.

⁴ Exlocator is available at <https://github.com/crashanalysis/ExLocator>.

```

java.lang.RuntimeException: Unable to resume activity {*}: 
java.lang.NumberFormatException: Invalid double: "" 
    at android.app.ActivityThread.performResumeActivity(...) 
    ... 
Caused by: java.lang.NumberFormatException: Invalid double: "" 
    at java.lang.StringToReal.invalidReal(StringToReal.java:63) 
    at java.lang.StringToReal.parseDouble(StringToReal.java:248) 
    ...

```

Fig. 2. An example of RuntimeException trace.

which indicates the root cause. Java uses *exception wrapping*, i.e., one exception is caught and wrapped in another to propagate exceptions. In this case, `RuntimeException` in the top part wraps `NumberFormatException`. Note that the root exception can be wrapped by multiple exceptions, and the flow from the bottom to the top denotes the order of exception wrappings. An *exception signaler*, the first called method under the root exception declaration (e.g., `invalidReal` in this case), is the method that throws the exception. To classify each exception, we referred to Android documentation [30] (API level 18) and the heuristic rules defined by prior work (Table II in [31]) according to the signaler's origin: (1) *Application Exception*: the signaler is defined in the application code. We can recognize it by the application's package name. (2) *Framework Exception*: the signaler is defined in the Android framework, i.e., from these packages: “`android.*`”, “`com.android.*`”, “`java.*`”, and “`javax.*`”. (3) *Library Exception*: the signaler is defined in the lib-core in Android framework (e.g., “`org.apache.*`”, “`org.json.*`”, “`org.w3c.*`”) or third-party libraries used by the app. Note that, in this study, we do not consider native crashes caused by C++ exceptions, and do not consider Java exceptions caused by the bugs of Android framework itself.

2.2 Data Collection

2.2.1 App Subjects

We collected our app subjects from F-Droid [32] and Google Play Store [33]. We chose F-Droid due to three reasons. First, it is the largest repository of open-source Android apps. At the time of our study, it contains over 2,104 unique apps and 4,560 different releases (each app has 1~3 recent releases), and maintains their metadata (e.g., project addresses, history versions). Second, the apps have diverse categories (e.g., Internet, Personal, Tools), covering different maturity levels of developers, which are the representatives of real-world apps. Third, all apps are hosted on GitHub, Google Code, SourceForge, etc, which makes it possible to access their source code and issue repositories. Additionally, we randomly selected 3,230 closed-source apps from Google Play, Google's Android app market, which has millions of commercial apps with diverse categories. We uniformly selected these apps from the top ten categories (e.g., Education, Lifestyle, Business, Tools) [1], and each app has at least *one million installations*. These apps could be regarded as the representatives of commercial apps.

2.2.2 Exception Trace Collection

Table 1 summarizes the statistics of collected exception traces from hosting platforms (GitHub and Google Code) and testing tools. We applied testing tools on both F-Droid apps and Google Play apps to collect exceptions.

TABLE 1
Statistics of Collected Crashes

Sources	#Projects	#Crashes	#Unique Crashes
Platforms (GitHub/Google Code)	2,174 (2,035/137)	7,764 (7,660/104)	6,588 (6,494/94)
F-Droid (M./Sa./St.)	2,104 (4,560 versions)	13,271 (3,758/4,691/4,822)	9,722 (3,086/4,009/3,535)
Google Play (M./Sa./St.)	3,230	293,266 (169,869/58,551/64,846)	13,764 (5,634/3,839/4,291)
Total	5,716 (1,792 overlap)	314,301	30,009

(“M.”: Monkey; “Sa.”: Sapienz; “St.”: Stoat)

Issue Repositories. We collected exception traces from GitHub and Google Code since they host over 85 percent (2,174/2,549) F-Droid apps. To automate data collection, we implemented a web crawler to automatically crawl the issue repositories of these apps, and collected the issues that contain exception traces. In detail, the crawler visits each issue and its comments to extract valid exception traces. Additionally, it utilizes GitHub and Google Code APIs to collect project information such as package name, issue id, number of comments, open/closed time. We took about two weeks and successfully scanned 272,629 issues from 2,174 apps, and finally mined 7,764 valid exception traces (6,588 unique) from 583 apps.

Automated GUI Testing Tools. To test F-Droid apps (4,560 recent release versions of 2,104 apps) and Google Play apps (3,230 apps), we chose three state-of-the-art Android app testing tools with different strategies: Monkey [28] (random testing), Sapienz [11] (search-based testing), and Stoat [12] (model-based testing). Each tool is configured with default settings and each app is given 3 hours to thoroughly test on a single Android emulator. Each emulator is configured with Jelly Bean Android OS (SDK 4.3.1, API level 18). The evaluation is deployed on three physical machines (64-bit Ubuntu/Linux 14.04). Each machine runs 10 emulators in parallel. Since Sapienz and Stoat leverage code coverage to optimize test generation, we instrumented apps by Emma [34] or Jacoco [35] to collect coverage data.

This data collection phase took 6 months in total, and we finally detected 13,271 crashes (9,722 unique) for open-source apps, and 293,266 crashes (13,764 unique) for commercial apps. During testing, when an app crashes, the exception trace with bug-triggering inputs, screenshots, detection time, etc, are recorded to help our analysis.

Notably, for F-Droid apps, we find that the issue repositories of GitHub and Google Code only contain 545 unique crashes that were reported with stack traces, for the 4,560 recent release versions. These crashes only accounts for 5.6 percent of those detected by testing tools. This indicates these exception traces collected by testing tools can indeed effectively complement the mined exceptions.

2.2.3 Other Resource Collection

To help analysis, we also collected the most relevant posts with the most votes on Stack Overflow by searching key words with “Android”, exception types and exception messages. We recorded the creation time, number of votes, number of answers, summary, etc. Finally, we mined 15,678 posts of various exceptions.

TABLE 2
Survey Questionnaire of Our Study

ID	Question Options/Types
Part I: Background Information	
Q1	Experience in years as an Android developer/tester? (<1 / 1~3 / 3~6 / >6 years)
Q2	Working place? (country, company/institution)
Q3	App category developed? (e.g., Education, Lifestyle, Business, Entertainment)
Q4	Awareness of Android app exceptions? (Yes/No)
Part II: App Exception Experiences and Practices	
Q5	Ever encountered all the three exception categories? (Yes/No)
Q6	Pervasiveness of framework-specific exceptions? (< 10%, 10%~30%, 30%~50%, 50%~70%, > 70%)
Q7	Ever encountered root causes of app exceptions? (the 11 fault taxonomies, and "Others")
Q8	How difficult of understanding each root cause? (Difficult / Medium / Easy)
Q9	Main difficulties of diagnosing root causes? (e.g., reproduction steps, bug environment, and "Others")
Q10	Importance of resolving exceptions before release? (Very important/Important/Normal/Not important/Ignored)
Q11	Tools/Platforms to reveal app exception bugs? (e.g., Monkey, UIAutomator, Lint, R&R tools)
Q12	Unsatisfactory points of existing testing tools? (e.g., manual efforts, false alarms, inefficiency, and "Others")
Q13	Failure rate of reproducing exception bugs given reproducing steps? (< 10% / 10%~30% / 30%~50% / >50%)
Q14	Reasons affecting the reproducibility? (e.g., concurrency, device models, system settings, and "Others")
Q15	Practices/Tools for improving reproducibility? (open question)
Q16	Popularity of common fix practices? (the 5 common fix practices found by our study)
Q17	Fix rate with only an exception trace? (< 10%, 10%~30% / 30%~50% / 50%~70% / >70%)

2.3 Online App Developer Survey

2.3.1 Questionnaire Design

To gain more understanding and validate our own analysis results on exception bugs, we conducted an online app developer survey. This survey aims to solicit Android app developers to share their experience of analyzing, testing, reproducing and fixing exception bugs. Table 2 presents the questionnaire of our study, which includes Q1~Q17. Specifically, the survey is designed as two parts.

Part I: Background Information. We collected the background information of developers via Q1~Q4. By these questions, we can filter invalid developers (e.g., the survey only proceeds if the developer is aware of app exceptions), and get the survey results of different developer groups (e.g., groups of developers with different experience levels, different app categories and countries).

Part II: App Exception Experiences and Practices. We collected developers' experiences and practices information via Q5~Q17. We initially designed a number of questions according to our research questions RQ1~RQ6, and sent them to three experienced Android app developers (with 5-year+ development experience) from Google, Tencent and Alibaba, respectively, for early feedback. We later refined these questions several rounds, and come up with Q5~Q17. This design process aims to make the questions intuitive to developers and concentrate on those questions that both developers and researchers really concern.

For the developers who are aware of app exceptions, we provided three examples for each exception category to make sure the developers can fully understand the survey's purpose and related terminologies. Then, we presented Q5~Q17 to systematically understand the developers' practices from

different perspectives. Specifically, we collected information about (1) whether developers have encountered the three exception categories via Q5 and Q6 (cf. Section 3.1.1), (2) how developers understand framework exceptions via Q7~Q9 (cf. Section 3.2.3), (3) how developers detect these exceptions in practice via Q10~Q12 (cf. Section 3.3), (4) how developers reproduce these exceptions via Q13~Q15 (cf. Section 3.5), and (5) how developers fix these exceptions via Q16~Q17 (cf. Section 3.6). In particular, some questions (e.g., Q5, Q6, Q7, Q16) aim to validate our analysis results; some questions (e.g., Q9, Q12, Q13, Q14, Q15, Q17) aim to understand developers' experiences and practices; some questions (e.g., Q7, Q9, Q12, Q14) are given with some options (summarized and refined according to our research experience and discussions with three senior developers), and an "Others" option to allow any additional comments.

2.3.2 Participants

To get sufficient number of responses from developers, we solicited the participants from three channels. First, we contacted 4,428 open-source app developers from GitHub and 1,226 commercial app developers from Google Play by scrawling their emails. Second, we invited the app developers in industry to distribute the survey within their companies and networks. These contacts are from Google, Tencent, Huawei, Alibaba and other IT companies. Third, we recruited app developers from Amazon Mechanical Turk [36] to participate in our survey. We paid 1.5 USD payment for each approved submission. Finally, we received valid responses from 135 professional app developers. Specifically, These developers come from 32 different countries across four different continents (Asia, Europe, North America, Oceania), and develop a diverse categories of apps (22 different categories). Among them, *Business*, *Tools*, *Education*, *Lifestyle*, *Entertainment* are the most popular categories. 10 developers are also involved in banking, insurance, financial apps, which emphasize more on robustness and safety. Among these 135 participants, 25 developers (18.5 percent) have less than 1-year experience, 67 developers (49.6 percent) have 1~3 years' experience, 35 developers (25.9 percent) have 3~6 years' experience, and 8 developers (6.0 percent) have more than 6 years' experience. Most of the developers, i.e., 100 participants (81.5 percent) have more than 1-year development experience.

3 EMPIRICAL STUDY

3.1 RQ1: Characteristics of Exceptions

3.1.1 Exception Category and Distribution

Based on the data collected in Section 2.2, Table 3 lists the exception categories of open-source and closed-source apps, and shows the number of the affected projects, occurrences, number of exception types and issue closing rate. Since Google Play apps do not have publicly available issue repositories, we only collected the closing rate for F-Droid apps. We can see two facts: (1) Framework exceptions are more pervasive and affect most of the apps. For example, 75.3 percent of open-source apps (revealed by the data of GitHub & Google Code) and 84.5 percent of closed-source apps (revealed by the data of testing tools) suffer from framework exceptions. In terms of exception occurrences,

TABLE 3

Statistics of the Exceptions Crawled From GitHub & Google Code and Collected by Testing Tools on F-Droid and Google Play Apps

Source	Exception Category	#Projects	Occurrences	#Types	Closing Rate
F-Droid (GitHub & Google Code)	Application	268 (45.8%)	1552 (23.6%)	88 (34%)	67%
	Framework	441 (75.3%)	3,350 (50.8%)	127 (50%)	53%
	Library	253 (43.2%)	1,686 (25.6%)	132 (52%)	57%
F-Droid (Testing tools)	Application	1,869 (50.9%)	4,017 (41.3%)	35 (35.0%)	-
	Framework	2,400 (65.3%)	5,072 (52.2%)	62 (62.0%)	-
	Library	366 (10.0%)	633 (6.5%)	44 (44.0%)	-
Google Play (Testing tools)	Application	389 (23.4%)	1,199 (14.4%)	20 (27.8%)	-
	Framework	1,405 (84.5%)	6,205 (74.1%)	44 (61.1%)	-
	Library	402 (24.2%)	965 (11.5%)	40 (55.6%)	-

(classified into Application Exception, Framework Exception, and Library Exception, respectively)

framework exceptions occupy more than half of all exceptions (50.8 percent for open-source apps revealed by GitHub/Google Code data, 74.1 percent for closed-source apps revealed by testing tools). This observation also conforms to the results of our survey question Q5 and Q6: 108 developers (80 percent), report they have encountered framework exceptions, and 88 developers (57.8 percent) report, in their experience, framework exception occupies around 30% ~ 50% (reported by 35 developers) and 50% ~ 70% (reported by 43 developers) among the three exception categories. (2) The closing rate of framework exceptions is 53 percent, which is relatively lower than those of the others (67 percent for application and 57 percent for library exception).

3.1.2 Locations of Framework Exception Manifestation

To understand framework exceptions, we grouped them by the class names of their signalers. In this way, we got more than 110 groups. To distill our findings, we further grouped these classes into 17 modules by following the insights of popular Android development tutorials [37], [38]. In our context, the classes in one *module* achieve either one general purpose or stand-alone functionality from developers' perspective. For example, we grouped the classes that manage the Android application model (e.g., *Activities*, *Services*) into *App Management* (corresponding to `android.app.*`); the classes that manage app data from *content provider* and *SQLite* into *Database* (`android.database.*`); the classes that provide basic OS services, message passing and inter-process communication into *OS* (`android.os.*`). Other modules include *Widget* (UI widgets), *Graphics* (graphics tools that

handle UI drawing), *Fragment* (one special visual element), *WindowManager* (manage window display), etc.

Fig. 3 shows the exception-proneness⁵ of Android framework modules in terms of unique exception instances. We find *App Management*, *Database* and *Widget* are the top 3 exception-prone modules. In *App Management*, the most common exceptions are *ActivityNotFound* (due to no activity is found to handle a given intent) and *IllegalArgumentException* exceptions (due to improper registering/unregistering *BroadcastReceiver* in the activity's callbacks). Surprisingly, although *Activity*, *BroadcastReceiver* and *Service* are the basic building blocks of apps, developers make the most number of mistakes on them.

As for *Database*, *CursorIndexOutOfBoundsException*, *SQLException*, *SQLiteDatabaseLocked* account for the majority, which reflect the various mistakes of using *SQLite*, the default database of Android. As for the other modules, we find: (1) improper use of *ListView* with *Adapter* throws a large number of *IllegalStateException* exception (account for 47 percent) in *Widget*; (2) In *OS*, *SecurityException*, *IllegalArgumentException*, *NullPointerException* are the most common ones. (3) improper use of *Bitmap* causes *OutOfMemoryError* (48 percent) in *Graphics*; (4) improper handling callbacks of *Fragment* brings *IllegalStateException* (85 percent) in *Fragment*; improper showing or dismissing dialogs triggers *BadTokenException* (25 percent) in *WindowManager*.

3.1.3 Locations of Library Exception Manifestation

To investigate the library exception, we used the exception data collected in Table 3. We grouped these exceptions by the class names of their signalers, and integrated the exceptions that are thrown from the same library. We finally got 100+ exception-prone libraries. Fig. 4 shows the top 15 libraries in terms of number of unique exception occurrences. We find *libcore*, *org.apache*, and *org.json* are the three most exception-prone libraries, which are in fact the most basic ones and more frequently used than the others.

We further randomly selected 10 library exceptions from each of these top 15 libraries, and analyzed the root causes. We find that although these libraries provide different functionalities, their exceptions still have some common root causes. For example, most of exceptions are due to the misuse of APIs, e.g., giving incorrect parameter values/formats, failing to validate specific resources (e.g., network)

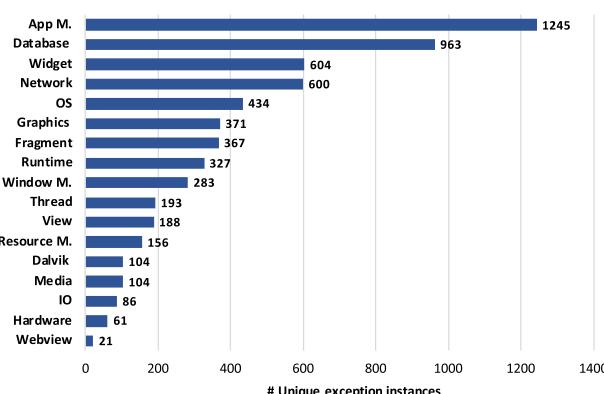


Fig. 3. Exception-proneness of Android framework modules in terms of unique instances (M. = Management)

5. In our context, exception-proneness indicates how often developers may misuse specific framework or library functionalities, and does not indicate the correctness of Android framework or libraries themselves. Specifically, these misuses manifest themselves as exceptions.

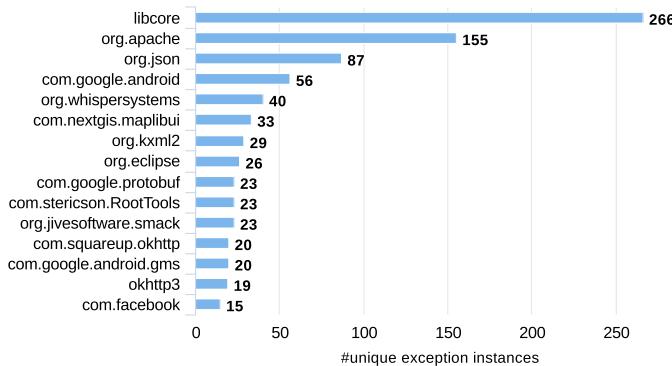


Fig. 4. Top 15 exception-prone libraries in terms of unique instances based on the data in Table 3.

before use. Some exceptions are caused by the API incompatibility issues [39] between the Android SDK/app and the library version, lack of specific hardware support or permissions [40]. Only a small portion of exceptions are due to the bugs of libraries themselves. These observations reveal that library exceptions do share similarity with framework exceptions (detailed in Section 3.2) in terms of common root causes. The Android framework can be actually viewed as a basic “library” that forms the building blocks of Android apps. In this paper, we focus on investigating framework exceptions. Different apps may use different libraries. Thus, giving a thorough analysis of library exceptions is not possible in this work alone. Thus, we leave it as future work. We have not given the manifestation locations of application exceptions, since these exceptions can be thrown from arbitrary locations at the app code level. We inspected a number of application exceptions, but most of them were generic programming errors. Thus, we do not give further exploration on application exception in this study.

Answer to RQ1: Framework exceptions are more pervasive than the other two exception categories, among which App Management, Database and Widget are the three most exception-prone modules for developers. Library exceptions are similar with framework exceptions in the terms of root causes.

3.2 RQ2: Taxonomy of Framework Exceptions

This section characterizes the framework exceptions and classifies them into different categories based on their root causes. According to ISTQB [41], “Root cause is a source of a defect such that if it is removed, the occurrence of the defect type is decreased or removed.” Specifically, in our context, we define *root cause*, from the view of developers, is the *initiating cause* [42] of either a condition or a causal chain that leads to a visible exception bug. Section 3.2.1 explains how we analyze and abstract these framework exceptions into different categories. Section 3.2.2 illustrates these categories with concrete examples.

3.2.1 Exception Analysis Method

First, we collected 8,243 framework exceptions and partitioned them into different *exception buckets*. Each bucket contains the exceptions that share the similar root cause. Specifically, we used the exception type, message and signaler to approximate the root cause. We also removed app specific

information in the exception message to scale the partition. For example, the exception in Fig. 2 is labeled as (*NumberFormatException*, “*invalid double*”, *invalidReal*). Here, we removed the empty string from the original exception message. We finally got 2,016 buckets, and the top 200 buckets contain over 80 percent of all exceptions. The remaining buckets have only 5 exceptions or fewer in each of them. Therefore, we focus on the top 200 buckets.

Second, we randomly selected a number of exceptions from each bucket, and used three complementary resources to facilitate root cause analysis: (1) *Exception-Fix Repository*. We set up a repository that contains pairs of exceptions and their fixes. In particular, (i) from 2,035 Android apps hosted on GitHub, we mined 284 framework exception issues that are closed with corresponding patches. To set up this mapping, we checked each commit message by identifying the keywords “fix”/“resolve”/“close” and the issue id. (ii) We manually checked the remaining issues to include valid ones that are missed by the keyword rules. We finally got 194 valid issues. We investigated each exception trace and its patch to understand the root causes. (2) *Exception Instances Repository*. From the 9,722 exceptions detected by testing tools (see Table 3), we filtered out framework exceptions, and linked each of them with its exception trace, source code, bug-triggering inputs and screenshots. When an exception type under analysis is not included or has very few instances in the exception-fix repository, we referred to this repository to facilitate analysis by using available reproducing information. (3) *Technical Posts*. For each exception type, we referred to the posts from Stack Overflow collected in Section 2.2.3 when needing more information from developers and validating our understanding.

Finally, we analyzed 86 distinct exception types, which covers 84.6 percent of all framework exceptions,⁶ and distilled 11 common fault categories. Specially, we abstracted the common faults by the three steps. First, we read the official Android documentation and popular developer tutorials to identify and understand Android’s important mechanisms (e.g., activity lifecycle, single-GUI-thread model), components (e.g., activity, service, thread, database), and features (e.g., XML-based UI design, API compatibility). Second, we inspected each exception bug to understand its own root cause by using the resources stated above. Third, we abstracted the root cause into which mechanism it violates, or which component or feature it fails in. By these information, we classified an exception into one specific fault category, which is named after specific mechanism errors (i.e., Component Lifecycle Error, UI Update Error, Framework Constraint Error), component usage errors (i.e., Concurrency Error, Database Management Error), feature errors (i.e., API Updates and Compatibility, Memory/Hardware Error, XML Design Error) or generic errors (Resource Not Found Error, API Parameter Error, Indexing Error).

3.2.2 Taxonomy

- *Component Lifecycle Error*. Each Android component has its own lifecycle and is required to follow the prescribed lifecycle paradigm, which defines how the component is

⁶ We found 13.2 percent of all exceptions are *NullPointerException*, which are caused by null pointer dereferences and highly related to the specific logic of each app. Thus, we did not inspect this generic exception type in our analysis.

```

class DataRetrieverTask extends AsyncTask<String, ..., ...> {
    private BankEditActivity context;
    protected Void doInBackground(final String... args) {
        ... //update bank info via the remote server
    }
    protected void onPostExecute(final Void unused) {
        ... //show the update progress
        AlertDialog.Builder builder = new AlertDialog.Builder(context);
        ... //set dialog message
        AlertDialog alert = builder.create();
        + if(!context.isFinishing()) {
            alert.show();
        }
    }
}

```

Fig. 5. Bankdroid Issue #471 (Simplified).

created, used and destroyed [43]. For example, `Activity` provides six core callbacks to allow developers to be aware of its current state. If developers improperly handle the callbacks or miss state-checking before some tasks, the app can be fragile considering the complex environment interplay (e.g., device rotation, network interruption). *Bankdroid* [44] (Fig. 5) is a Swedish banking app. It utilizes a background thread `DataRetrieverTask` to perform data retrieval, and pops up a dialog to inform that the task is finished. However, if the user presses the back button on `BankEditActivity` (which starts `DataRetrieverTask`), the app will crash when it tries to pop up a dialog. The reason is that the developers fail to check `BankEditActivity`'s state (in this case, *destroyed*) after the background task is finished. The bug triggers a `BadTokenException` and was fixed in revision 8b31cd3 [45]. Besides, `Fragment` [46], a reusable class implementing a portion of `Activity`, has much more complex lifecycle. It provides 12 core callbacks to manage its state transition, which makes lifecycle management more challenging, e.g., state loss of `Fragments`, attachment loss from its activity.

- *UI Update Error*. Android enforces the single GUI thread model. A UI thread is in charge of dispatching events and rendering user interface. Each app owns one UI thread and should offload intensive tasks to background threads to ensure responsiveness. *cgeo* [47] (Fig. 6) is a popular full-featured client for geocaching. When refreshing `cacheList` (`cacheList` is associated with a `ListView` via an `ArrayAdapter`), the developers query the database and substitute this list with new results (via `clear()` and `addAll()`) in `doInBackground`. However, the app crashes when it tries to refresh the list. Because `cacheList` is maintained by the UI thread, which internally checks the equality of item counts between `ListView` and `cacheList`. But when a background thread modifies `cacheList`, the checking will fail and an exception will be thrown. The developer fixed it by moving the refreshing operations into `onPostExecute`, which instead runs in the UI thread (in revision d6b4e4d [48]).

```

private List<Geocache> cacheList = new ArrayList<>();
private CacheListAdapter adapter =
    ... // adapter binds cacheList and ListView
new AsyncTask<Void, Void, Void>() {
    protected Void doInBackground(final Void... params) {
        ... //run in the background thread
        final Set<Geocache> cacheListTmp = ... //query database
        - if (CollectionUtils.isNotEmpty(cacheListTmp)){
            - cacheList.clear();
            - cacheList.addAll(cacheListTmp);
        }
    }
}

```

Fig. 6. cgeo Issue #4569 (Simplified).

```

public class GSMService extends LocationBackendService{
    protected Thread worker = null;
    ... //start the service
    worker = new Thread() {
        public void run() {
            + Looper.prepare();
            final PhoneStateListener listener =
                new PhoneStateListener() {
                    ... //callbacks to monitor phone state change
                };
            worker.start();
        }
    }
}

```

Fig. 7. Local-GSM-Backend Issue #2 (Simplified).

- *Framework Constraint Error*. Android defines a number of constraints when using its framework to build an app. For example, *Each Handler* [49] instance must be associated with a single thread and the message queue of this thread [50]. Otherwise, a runtime exception will be thrown. *Local-GSM-Backend* [51] (Fig. 7), a popular cell-tower based location lookup app, uses a thread worker to monitor the changes of telephony states via `PhoneStateListener`. However, the developers are unaware that `PhoneStateListener` internally maintains a `Handler` instance to deliver messages [52], which requires setting up a message loop in `worker`. They later fixed it by calling `Looper#prepare()` (in revision 07e4a759 [53]). Other constraints include performance consideration (avoid performing network operations in the main UI thread [54], permission consideration (require runtime permission grant for dangerous permissions [55] since Android 6.0, otherwise `SecurityException`) and etc.

- *Concurrency Error*. Android provides a number of asynchronous programming constructs, e.g., `AsyncTask`, `Thread`, to concurrently execute intensive tasks. However, improper handling them may cause data race [56] or resource leak [57], and even app crashes. *Nextcloud Notes* [58], a cloud-based notes-taking app, automatically synchronizes local and remote notes. It attempts to re-open an already-closed database, causing app crash [59]. The exception can be reproduced by executing two steps repeatedly: (1) open any note from the list; (2) close the note as quickly as possible by pressing back-button. The app creates a new `NoteSyncTask` every time when a note sync is requested, which connects with the remote sever and updates the local database by calling `updateNote()`. However, when there are multiple update threads, such interleaving may happen and crash the app: *Thread A* is executing the update, and *Thread B* gets the reference of the database; *Thread A* closes the database after the task is finished, and *Thread B* tries to update the closed database. The developers fixed this exception in revision aa1a972 [60] by leaving the database unclosed (since `SQLiteDatabase` already implemented thread-safe database access mechanism).

- *Database Management Error*. Android uses `SQLite` as its default database. Many errors are caused by improper manipulating database columns/tables. Besides, improper data migration for version updates is another major reason. *Atarashii* [61] (Fig. 8) is a popular app for managing the reading and watching of anime. When the user upgrades from v1.2 to v1.3, the app crashes once started. The reason is that the callback `onCreate()` is only called if no old version database file exists, so the new database table `friends` is not successfully created

```

public void onCreate(SQLiteDatabase db) {
    ... //create database tables
    db.execSQL(CREATE_FRIENDS_TABLE);
}
public void onUpgrade(SQLiteDatabase db, int oldVersion,
    int newVersion) {
    // upgrade database
    if (oldVersion < 5) { ... }
    if (oldVersion < 6) {
        - db.execSQL("create table temp_table as
                    select * from " + TABLE_FRIENDS);
        - db.execSQL("drop table " + TABLE_FRIENDS);
        + db.execSQL(CREATE_FRIENDS_TABLE);
        ...
    }
}

```

Fig. 8. Atarashii Issue #82 (Simplified).

when upgrading. Instead, `onUpgrade()` is called, it crashes the app because the table `friends` does not exist (fixed in revision b311ec3 [62]).

- *API Updates and Compatibility.* Android features fast API updates. For example, `Service` should be started explicitly since Android 5.0; the change of the comparison contract of `Collections#sort()` [63] since JDK 7 crashes many apps due to the developers are unaware of this. It also has device fragmentation issues, which were already investigated by prior work [64], [65]. For example, problematic deriver implementation, non-compliant OS customization, and peculiar hardware configuration may cause compatibility issues.

- *Memory/Hardware Error.* Android devices have limited resources (e.g., memory). Improper using of resources may cause app crashes. For example, `OutOfMemoryError` occurs if loading too large Bitmaps; `RuntimeException` appears when `MediaRecorder#stop()` is called without valid audio/video data received.

- *XML Design Error.* Android supports UI design and resource configuration in the form of XML files. Although IDE tools have provided much convenience, mistakes still exist, e.g., misspelling custom UI control names, forgetting to escape special characters (e.g., "\$", "%") in string texts, failing to specify correct resources in `colors.xml` and `strings.xml`.

- *Resource Not Found Error.* Android apps heavily use external resources (e.g., databases, files, sockets, third-party apps and libraries) to accomplish tasks. Developers make this mistake when they fail to check their availability.

- *API Parameter Error.* Developers make such mistakes when they fail to consider all possible input contents or formats, and feed malformed inputs as the parameters of APIs. For example, they directly use the results from `SharedPreference` or database queries without any checking.

- *Indexing Error.* Indexing error happens when developers access data, e.g., `database`, `string`, and `array`, with a wrong index value. One typical example is the `CursorIndexOutOfBoundsException` exception caused by accessing database with incorrect cursor index.

3.2.3 Understanding Root Causes From Developers

To further validate the results of root cause analysis, we surveyed the developers with three questions. In the first question (Q7), we aimed to check the correctness and completeness of root causes. We listed the 11 root causes

(accompanied with 2~3 issue examples) that can cause framework exceptions, and asked developers to choose anyone that he or she has ever encountered. We also provided an additional option “*Others*” for developers to fill in any root causes we may have missed in our study. In the second question (Q8), we aimed to understand how difficult the developers may feel when resolving the exceptions with these root causes (including the effort to inspect the exception message, understand the root case, and locate the faulty code). We gave them the three options, i.e., *Difficult*, *Medium*, and *Easy*, to rate each root cause. In the third question (Q9), we aimed to understand the difficulties of diagnosing root causes. We gave the four options, i.e., *understand exception type and message*, *get the reproduction steps* (the user actions to trigger the exception), *get the bug environment* (e.g., app version, device info), *understand the principles or usages of specific Android APIs*, and an additional option “*Others*”.

The responses of the first question support our root cause analysis. All of the 11 root causes were encountered by the developers. Specifically, Framework Constraint Error (encountered by 62 developers (45.9 percent of all developers)), API Updates and Compatibility Error (60 developers (44.4 percent)), Lifecycle Error (53 developers (39.3 percent)), UI Update Error (53 developers (39.3 percent)) are the four most commonly encountered errors reported by developers. This finding conforms to our analysis results. In Table 4, “#Occ.” denotes the exception occurrences of each root cause among the 8,243 framework exceptions. We can see, besides those “trivial” errors such as Resource-Not-Found Error, Index Error and API Parameter Error, app developers are indeed more likely to make Android specific errors, e.g., Lifecycle Error, Memory/Hardware Error, Framework Constraint Error. Some developers also mentioned some exception instances in the “*Others*” option. For example, one developer mentioned *improperly using of Android APIs*, which was categorized into the API Parameter Error category; another developer mentioned *not properly handling the state of the listeners for sensors*, which was categorized into the Framework Constraint Error. Additionally, 42 developers (31.1 percent of all developers) mentioned Android system errors (i.e., the bugs of Android framework itself) can also lead to framework exceptions, which is indeed true but out of our scope.

In the second question, we find developers have different assessments on the difficulties of these root causes according to their experience. Resource-Not-Found Error, API Parameter Error, Index Error, and XML Error were the top four most *Easy* errors rated by 50.4, 48.1, 44.4, 43 percent of all developers, respectively, since these errors are usually induced by trivial human mistakes and easy to fix. On the other hand, Memory/Hardware Error, Concurrency Error, and API Updates and Compatibility Error were the top three most *Difficult* errors rated by 46.7, 34.8, 29.6 percent developers, respectively, because these errors are notoriously difficult to debug [56], [66]. As for Database Management Error, UI Update Error, Framework Constraint Error, Lifecycle Error, almost half of participants, i.e., 51.8, 48.1, 46.7, and 46.7 percent of all developers, respectively, rated them as *Medium*. This finding also conforms to our observation on Stack Overflow. In Table 4, “#S.O. posts” counts the number of Stack Overflow posts on discussing these faults.

TABLE 4
Statistics of 11 Common Fault Categories, Sorted by *Closing Rate* (Collected From GitHub) in Descending Order
 (“Occ.”: Occurrences, “S.O.”: Stack Overflow)

Category (Name for short)	#Occ.	#S.O. posts	Closing Rate
API Updates and Compatibility (API)	68	60	93.3%
XML Layout Error (XML)	122	246	93.2%
API Parameter Error (Parameter)	820	819	88.5%
Framework Constraint Error (Constraint)	383	1726	87.7%
Others (Java-specific errors)	249	4826	86.1%
Index Error (Index)	950	218	84.1%
Database Management Error (Database)	128	61	76.8%
Resource-Not-Found Error (Resource)	1303	7178	75.3%
UI Update Error (UI)	327	666	75.0%
Concurrency Error (Concurrency)	372	263	73.5%
Component Lifecycle Error (Lifecycle)	608	1065	58.8%
Memory/Hardware Error (Memory)	414	792	51.6%

We can see developers indeed discuss more on Android Framework Constraint Error and Lifecycle Error.

Fig. 9 shows the responses for Q9. We can see 96 developers (71.1 percent of all developers) reached the consensus that the most difficult point is to get the reproduction steps, which is quite crucial for diagnosing the root cause. The second difficult point, mentioned by 72 developers (53.3 percent), is to get the bug environment. 57 developers (42.2 percent) confirmed the exception type and message sometimes also bring confusions, while 45 developers (33.3 percent) reported some specific Android APIs usages or features also affect the understanding of root causes. We received 5 answers from the “Others” option, but all of them can be grouped into the previous four difficulties due to similarity. Thus, we believe these four difficulties are the most typical ones.

Answer to RQ2: We distilled 11 fault categories of framework exceptions. Developers make more mistakes on Lifecycle Error, Memory/Hardware Error and Framework Constraint Error. Developers feel it difficult to resolve Concurrency Error, Memory/Hardware Error, and API Updates and Compatibility Error. Getting reproduction steps and bug environment are the two most difficult problems when diagnosing root causes.

3.3 RQ3: Detecting Exception Bugs

This section investigates the testing practices against exception bugs from developers’ perspective. Different from prior surveys [9], [10], [67] on how developers test Android apps, our investigation focuses on how developers detect these exception bugs that can lead to crashes. Specifically, we aim to understand (1) the importance of detecting exception bugs, (2) the commonly-used tools to detect exception bugs, and (3) the unsatisfactory points of these tools. This section

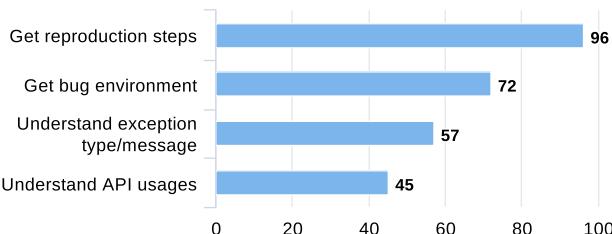


Fig. 9. Difficulties of root cause analysis.

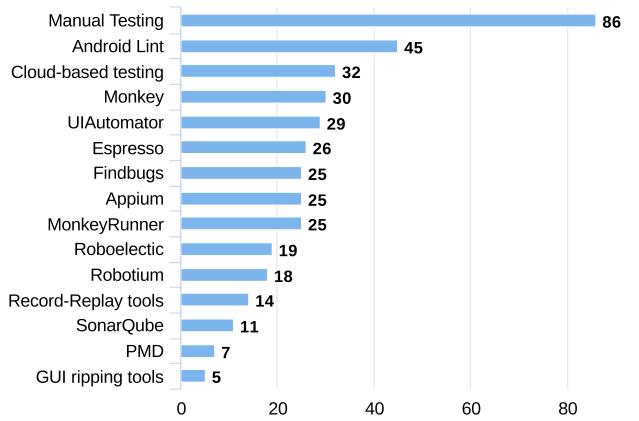


Fig. 10. Tools/Frameworks used by developers.

motivates our deep investigation on these bug detection tools in Section 3.4 (RQ4) and Section 3.5 (RQ5).

3.3.1 Tools for Detecting Exception Bugs

For the question Q10 “Do you think it is important to detect (and resolve) exception bugs before releasing your apps?”, the responses were very consistent: 56.3 percent developers chose *Very Important*, 34.8 percent developers chose *Important*, and 8.9 percent developers chose *Normal*. This result indicates that detecting exception bugs is indeed one of top priorities for developers.

In practice, many bug detection tools or frameworks are available to help detect potential app exceptions. Fig. 10 shows the tools that are used by app developers to test or check exception bugs (the responses of Q11). These tools can be categorized into different groups by their principles. For example, Monkey [28] is a random fuzzing tool that tests apps by emitting a stream of random input events; MonkeyRunner [68] is an API-based testing tool that tests apps/devices from functional or framework level. Other tools include unit/integration testing frameworks (e.g., Robolectric, Espresso, UIAutomator), script-based testing frameworks (e.g., Robotium, Appium), R&R (record & replay)-based tools, cloud-based testing service (e.g., Google Firebase, Microsoft Xamarin) and static checking tools (e.g., Findbugs, Android Lint, PMD, SonarQube).

We can see manual testing is still the most preferable way of 86 developers (63.7 percent) to find exception bugs. Android Lint is the most commonly-used tool by 45 developers (33.3 percent) to automatically scan app bugs, which is more popular than other static checking tools (i.e., FindBugs, PMD, SonarQube). 74 developers (54.8 percent) preferred using AndroidJUnitRunner-based unit and integration testing frameworks (e.g., Espresso and UIAutomator), and 32 developers (23.7 percent) resorted to cloud-based testing services (e.g., Google Firebase). We also notice only a few (5 developers) use automated GUI ripping tools. Sapienz [11] and Stoat [12], the two state-of-the-art tools, were used. Different from all the other tools, these GUI ripping tools are developed and maintained by researchers to achieve automated app testing.

3.3.2 Unsatisfactory Points of Existing Tools

In the survey, we further asked the developers Q12 “which points do you think the tools you used are still not satisfactory for

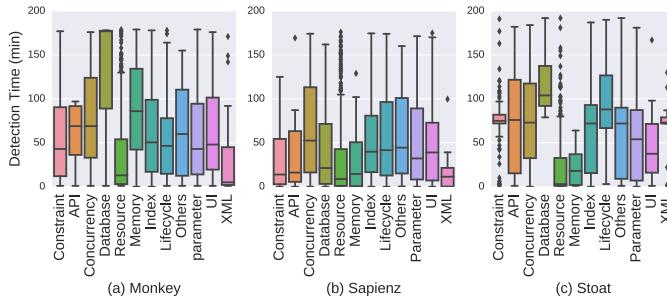


Fig. 11. Detection time of exceptions by each tool.

detecting exception bugs?”. From the responses, we have several findings. (1) 67 developers (49.6 percent) complained about the demanding human efforts required by *manually writing tests and setting up the testing environment*. Manual testing and those non-fully automatic testing methods (e.g., MonkeyRunner, AndroidJUnitRunner-based and script-based testing frameworks and R&R tools) all need manual efforts. (2) The inefficiency of uncovering exception bugs is another major concern of 64 developers (43.7 percent). They reported some tools either *cost too much testing time* (e.g., R&R tools) or *miss bugs* (e.g., Android Lint and other static checking tools). (3) 56 developers (41.5 percent) complained that *even if the tool finds an exception, the generated test cannot guarantee to reproduce the bug*. This indicates the bug reproducibility problem of mobile apps. Monkey and cloud-based testing service are the two typical methods that have this issue. For example, a Monkey test is a stream of low-level events (based on the device screen coordinates), which may probably fail to reproduce the bug if the screen size changes. Section 3.5 gives a deep investigation of this problem. (4) 47 developers (34.8 percent) mentioned that the static checking tools (e.g., Lint) and R&R tools can bring false alarms, i.e., *the reported issues are not real bugs*. This issue usually wastes developers’ time for inspecting them. (5) 43 developers (31.9 percent) reported that *some tools fail to consider various environment* (e.g., screen rotation, network stability, different geographic locations, heavy memory/CPU usage), which are quite crucial for testing the usability and robustness of mobile apps. (6) 42 developers (31.1 percent) hoped the testing or checking tools could generate tests for verification or generate more readable tests for debugging. For example, some developers desired to get more readable tests from Monkey. Developers have not provided other comments in the “Others” option.

Answer to RQ3: Most developers agree detecting exception bugs is crucial, however, manual testing is still the most preferable testing method. Although different bug detection tools are used, developers still have unsatisfactory points, e.g., high manual efforts, insufficient bug detection, low reproducibility rate, many false positives, lack of considering environment etc.

3.4 RQ4: Auditing Automatic Bug Detection Tools

Informed by the study of RQ3, this section aims to investigate the effectiveness of bug detection tools. As revealed by RQ3, most of the bug detection tools require human assistance (e.g., writing tests). We note two groups of tools, i.e., dynamic testing and static analysis tools, can fully automate app exception checking. However, our previous investigation on the four

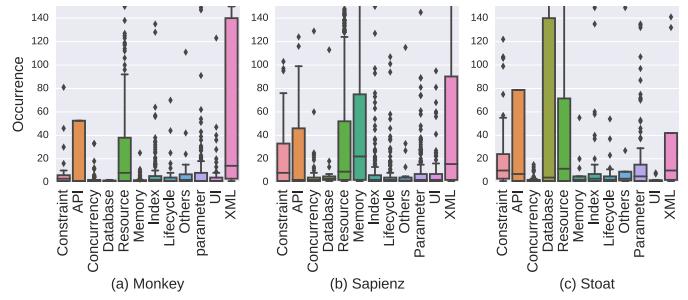


Fig. 12. Occurrences of exceptions by each tool.

static analysis tools, i.e., Lint, FindBugs, PMD, SonarQube, shows these tools are almost ineffective in detecting framework exceptions due to the lack of specific checking rules [29]. Unfortunately, these tools have not provided handy APIs or command line options to accept customized checking rules, and require considerable code-level extensions. Thus, we decided not to include them in this evaluation, otherwise the results could be unfair. Section 5 discusses plausible ways of improving static analysis tools. We do not consider the cloud-based testing services as well, which are pay-by-use and not convenient to conduct large-scale evaluation on thousands of apps. Therefore, we only focus on dynamic testing tools, and evaluate them on the framework exceptions categorized in Section 3.2. We selected 3 state-of-the-art dynamic testing tools, i.e., Monkey [28], Sapienz [11], and Stoat [12]. The survey in Section 3.3 shows these tools are used by a number of real app developers (35 developers ever used). More importantly, recent studies [69], [70] show, these tools are proved to be the most effective on both open-source and commercial apps, and have found hundreds of previously-unknown crash bugs in well-tested apps.

We applied dynamic testing tools on each of 2,104 apps with the same configurations in Section 2.2.2. We observed that they could detect many framework exceptions. To understand their abilities, we used two metrics.⁷ (1) *detection time* (the time to detect an exception). Since one exception may be found multiple times, we used the time of its first occurrence. (2) *Occurrences* (how many times an exception is detected during a specified duration). Figs. 11 and 12, respectively, show the detection time and occurrences of exceptions by each tool grouped by the fault categories.

From Fig. 11, we can see the abilities of these tools vary across different fault categories. But we also note some obvious differences. For example, following the guidelines of statistical tests [71], we used Mann-Whitney U test [72], a non-parametric statistical hypothesis test for independent samples, to compare the detection time of some specific fault categories across three tools. We find Sapienz is better at database errors (i.e., use significantly less testing time) than Monkey ($p\text{-value}=0.02$, and standardized effect size is *medium* (0.41)) and Stoat ($p\text{-value}=0.05*10^{-4}$, and standardized effect size is *large* (0.65)). One important reason is that Sapienz implements a strategy, i.e., fill strings in EditTexts, and then click “OK” instead of “Cancel” to maximize code

7. We do not present the results of trace length, since we find the three tools cannot dump the exact trace that causes a crash. Instead, they output the whole trace, which cannot reflect their detection abilities.

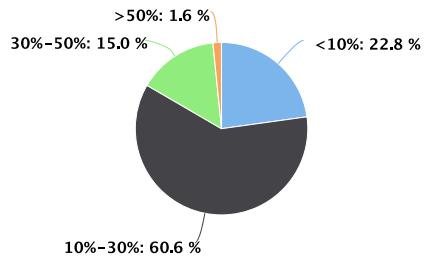


Fig. 13. Percentage of cases in failing to reproduce exceptions even if the reproduction steps are given.

coverage, which is more likely to trigger database operations. Monkey and Sapienz, respectively, are better at lifecycle errors than Stoat (*p*-values are, respectively, 0.002 and 0.001, and standardized effect sizes are, respectively, *medium* (0.35) and *small* (0.25)). Because both Monkey and Sapienz emit events very quickly without waiting for the previous ones to take effect, e.g., open and quickly close an activity without waiting for the activity finishes its task.

In addition, we note concurrency errors are non-trivial for all three tools, i.e., Monkey, Sapienz and Stoat. But their detection times are not significantly different according to our statistical test. The medians of their detection times are, respectively, 52, 69 and 58 minutes. In Fig. 12, the occurrences of API compatibility, Resource-Not-Found and XML errors are much more than those of many other fault categories across three tools. It indicates these errors are easier to be repeatedly detected. But, on the other hand, Concurrency, Lifecycle, UI update errors are more difficult to be repeatedly detected, regardless of the testing strategies of these tools. The main reason is that these errors contain more non-determinism (interacting with threads).

After an in-depth inspection, we find that some Database errors are hard to trigger because the app has to construct an appropriate database state (e.g., create a table or insert a row, and fill in specific data) as the precondition of the bug, which may take a long time. As for Framework Constraint Error, some exceptions require special environment interplay. For example, `InstantiationException` of Fragment can only be triggered when a Fragment is destroyed and recreated. To achieve this, a testing tool needs to change device rotation at an appropriate timing (when the target Fragment is on the screen), or pause and stop the app by switching to another one, and stay there for a long time (let Android OS kill the app), and then return back to the app. Concurrency bugs (e.g., data race) are hard to trigger since they usually need right timing of events.

Answer to RQ4: Dynamic testing tools are less effective in detecting concurrency, database and lifecycle errors. Different testing strategies have a big impact on the bug detection ability against different types of framework exceptions. More effective dynamic testing strategies are demanded to help detect framework exceptions.

3.5 RQ5: Reproducibility of Exception Bugs

This section investigates the reproducibility of exception bugs, which is crucial for bug diagnosing and fixing. Android apps are event-centric programs and run in complex environment. Typically, the bug-triggering inputs are described as a

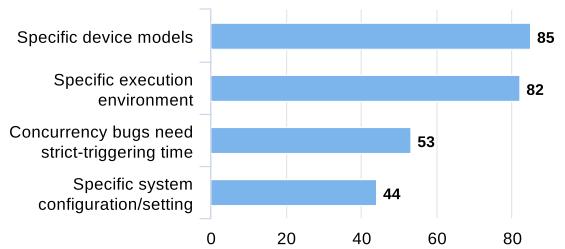


Fig. 14. Difficulties of reproducing exceptions.

few reproducing steps (in the form of natural language by humans or event sequences generated by testing tools) and contextual conditions (e.g., device models, network status, and other device settings [73]). Prior work improves the reproducibility of crash bugs by augmenting bug reports [74], [75], [76], [77], translating a bug report (written in natural language) into an executable UI test [78], [79], and leveraging crowd-sourced monitoring [80]. However, to our knowledge, no previous efforts has investigated the reproducibility of exception bugs, from these two perspectives: (1) how do app developers, and (2) how do automatic testing tools, perform in reproducing bugs, which this section will explore.

3.5.1 Perspective of App Developers

In our survey, we asked developers Q13 “*In your experience, given the reported reproducing steps, how much percentage of cases in which you still cannot reproduce the crash exception?*”. We gave them the four options, i.e., < 10%, 10% ~ 30%, 30% ~ 50%, and > 50%. Fig. 13 shows the responses. We find 82 developers (60.6%) reported they fail to reproduce 10% ~ 30% exception bugs, which were not ignorable. Further, 20 developers (15 percent) could not successfully reproduce 30% ~ 50% exception bugs, and 2 developers even could not reproduce over 50 percent exceptions. 31 developers (22.8 percent) chose the option < 10%. Further, from the responses of 43 senior developers (with over 3 years working experience), we find only 11 of them (25.6 percent) choose the option < 10%, which indicates experienced developers also face difficulties in reproducing bugs. Based on the above observations, although developers in fact are quite familiar with their own apps and implementations, we can see reproducing exception bugs is still difficult for human developers, even if the reproduction steps are given.

We further asked developers Q14 “*If you cannot reproduce the crash exception, in your experience, which reasons may affect the reproducibility?*”. Five options are provided: (A) concurrency or asynchronous bugs (e.g., data race), (B) specific running environment (e.g., low memory, external file access, usage of specific third-party library), (C) specific device models (e.g., framework API version, OS customization), (D) specific system configurations or settings (e.g., WiFi/4G, GPS on/off, enable/disable specific developer options), and (E) Others (for any developers’ comments). The first four options were distilled from three sources: (1) the app developers’ comments and discussions from GitHub issue repositories when they resolve bug reports, (2) our own experience of reproducing bugs during our own research [29], [79], [81], and (3) the previous work on bug reproduction [76], [78], [80].

Fig. 14 shows the results. 85 developers (63 percent) selected (C). They indicated different API versions and

TABLE 5
Statistics of Reproducible Exceptions Across the
Three Exception Categories

Tool	#Total	#Application	#Framework	#Library
Sapienz	279	82 (6.5%)	169 (7.2%)	28 (6.9%)
Stoat	269	189 (9.6%)	76 (5.3%)	4 (3.2%)

vendor models could affect the reproducibility because the platform where the apps are developed is usually different from the one where the apps are used. 82 developers (60.7 percent) chose (B). They indicated some specific execution environment (e.g., heavy system load, external file access, etc) may affect the reproducibility. The developers felt difficult to record and restore the exact environment when the app crashes. 53 developers (39.3 percent) selected (A), since some concurrency bugs require specific thread scheduling and strict timing [81]. 44 developers (32.6 percent) reported missing “specific system configurations or settings” in the reproduction steps could also affect the reproducibility. For example, some bugs can only be manifested with mobile data instead of WiFi.

We further asked app developers an open question Q15 “How do you improve the reproducibility of exception bugs during your development?”. 12 app developers answered this question. They added customized logging interfaces to gain important running information, or used some off-the-shelf crash reporting systems, e.g., ACRA [82], Google Firebase Crashlytics [83], Splunk MINT [84], to collect raw analytics. Specifically, these crash reporting systems (integrated as app plugins) collect the contextual environment (e.g., SDK, OS, app version, hardware model, memory usage), the exception traces, the steps leading to crash (usually in the form of screenshots) to facilitate crash analysis. However, these developers still felt quite challenging to faithfully reproducing exception bugs the users experience in vivo.

3.5.2 Perspective of Testing Tools

To investigate how testing tools perform in reproducing bugs, we chose two Android GUI testing tools, i.e., Sapienz and Stoat. As stated in Section 3.4, these two tools are now the state-of-the-art in finding crash bugs. Specifically, to record & replay the tests, we used Android Monkey script [85] for Sapienz, and UIAutomator script [86] for Stoat. When an app crashes during the testing, we will record the exception trace, and the corresponding crash-triggering test (i.e., the event sequences that led to the crash).

To mitigate test flakiness [87], [88], [89], we deployed the reproducing process on two physical machines, each of which ran 6 emulators with the exact same environment and configurations as the previous testing process in Section 3.4. In addition, we ran each test for five times, and recorded how many times the exception bugs could be triggered. The machine state was cleared between each test run. If the exactly same exception (with the same exception type and stack trace) was triggered among the 5 runs, we regarded the test as a valid one that can *faithfully* reproduce the crash. In total, we replayed the tests of 4,009 and 3,535 exception bugs (including all the three exception categories) found by Sapienz and Stoat, respectively. The whole

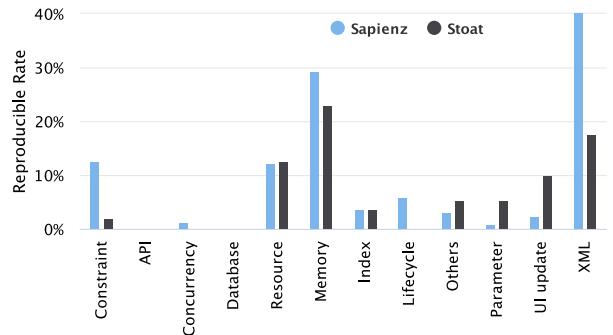


Fig. 15. Reproducibility rate of the 11 root causes of framework exceptions w.r.t. Sapienz and Stoat.

reproducing process took two months. Note that we have not included Monkey in this investigation, since we find the tests of Monkey are very flaky.⁸

Finally, Sapienz and Stoat triggered 15.7 percent (629/4,009) and 28.2 percent (996/3,535) of all exception bugs, respectively, by replaying the recorded tests. However, among these triggered exceptions, only 279 exceptions of Sapienz (6.9 percent, including 82 application exceptions, 169 framework exceptions, and 28 library exceptions) and 269 exceptions of Stoat (7.6 percent, including 189 application exceptions, 76 framework exceptions, and 4 library exceptions), respectively, were faithfully reproduced. Obviously, the reproducibility rate of exception bugs were quite low. In the remaining cases that triggered exceptions, we find the tests either triggered (1) the exceptions with different types, or (2) the exceptions with the same types but different stack traces. We further inspected a number of those “unfaithfully” reproduced exceptions (i.e., the cases in (2)), and found some of them actually triggered the same bugs but the stack traces were slightly different from the expected ones.

Table 5 shows the numbers of reproducible exceptions across the three exception categories, respectively. In the parentheses, the percentage numbers indicate the ratios of reproducible exceptions among all exceptions of that category. We can see the reproducibility rates of these three exception categories do not have much differences, although Stoat has lower rates on framework and library exceptions, compared to Sapienz. Fig. 15 shows the reproducibility rates of the 11 root causes of framework exceptions. We can see that both Stoat and Sapienz can reproduce more exceptions of Resource-Not-Found, Memory/Hardware, and XML Layout errors (over 10 percent), while neither of them has good performance at Concurrency, API Update and Compatibility, and Database Management errors.

Overall, the reproducibility of exception bugs is low for both Sapienz and Stoat. We further investigated the reasons behind, and observed the three main difficulties.

- *Test dependency.* Both Sapienz and Stoat only record the current test that triggers the exception. However, many exceptions can only be manifested under specific preconditions, which need to be created by some previous tests. As a result, only replaying the current test may fail the

⁸. Our preliminary investigation reveals Monkey’s tests are much more flaky than those of Stoat and Sapienz. We find most of Monkey’s tests have thousands of events, while those of Sapienz and Stoat have merely hundreds or tens of events, respectively.

```
// MozStumbler revision 6adbfe5
public class ServiceBroadcastReceiver extends BroadcastReceiver{
    public void onReceive(Context context, Intent intent) {
        String action = intent.getAction();
        ... // handle the intent
        if (mMainActivity != null) {
            - mMainActivity.updateUI();
            + mMainActivity.runOnUiThread(new Runnable() {
            +     public void run() {
            +         mMainActivity.updateUI();
            +     }
            + });
        }
    }
}
```

Fig. 16. Example fixes by moving code into correct thread.

reproduction. Simply recording all the previous tests is ineffective, while selectively recording the necessary tests *w.r.t.* the exception is nontrivial.

- *Timing of Events.* The execution timing of events are crucial for manifesting some types of exceptions. For example, concurrency bugs require critical timing of events, so as to create specific thread scheduling [81]. In other scenarios, due to the latency of network or computation, some UI widgets may not be quickly ready for executing the next event — causing the ignorance of the next event. Such ignorance may have negative effect on the execution of the whole event sequence, leading to totally different execution paths and results. Thus, to improve reproducibility, the tests should contain timing control operations.

- *Specific Running Environment or Configurations.* Triggering some exceptions require specific running environment, e.g., the existence of specific files on the SD card. For example, one of the reasons for *OutOfMemoryError* is that the app tries to load a large-size file from the SD card. Without this file, such exceptions could not be reproduced. Some exceptions can only be triggered under specific system configurations, e.g., disabling network access or granting the permission of using camera.

Answer to RQ5: Reproducing exceptions is difficult for developers, and also challenging for automated testing tools. Specific device models, specific execution environment, concurrency issues, specific system configurations are the four main difficulties rated by developers. The reproducibility rates of Sapienz and Stoat are quite low (only 6.9 and 7.6 percent, respectively). Test dependency, timing of events, and specific running environment are the three main observed challenges for testing tools to faithfully reproduce exceptions.

3.6 RQ6: Fixing Patterns and Characteristics

This section uses the exception-fix repository constructed in RQ2 (194 instances) to investigate the common practices of developers to fix framework exceptions. We categorized their fixing strategies by (1) the types of code modifications (e.g., modify conditions, reorganize/move code, tweak implementations); (2) the issue comments and patch descriptions. We finally summarized five common fix patterns, which can resolve over 90 percent of the issues in the repository. We further presented Q16 to the developers, and asked them to choose which fix practice they have ever used to fix framework exceptions. Fig. 18 shows the responses. We detail these fix practices as follows, which are ordered by the popularity from the most to the least.

```
(a) qBittorrent-Controller revision 8de20af
    Cursor cursor = contentResolver.query(...);
    - cursor.moveToFirst();
    + if( cursor != null && cursor.moveToFirst()) {
        int columnIndex = cursor.getColumnIndex(filePath);
        ... // get the result from the cursor
    }

(b) WordPress revision df3392f
    public class AbstractFragment extends Fragment{
        protected void showError(int messageId) {
        + if(!isAdded()) { return; }
            FragmentTransaction ft = getFragmentManager()...
            ... //commit a transaction to show a dialog
        }
    }
```

Fig. 17. Example fixes by adding conditional checks.

- *Work in Right Callbacks.* Inappropriate handling lifecycle callbacks of app components (e.g., Activity, Fragment, Service) can severely affect the robustness of apps. The common practice to fix such problems is to work in the right callback. For example, in Activity, putting BroadcastReceiver’s register and unregister into *onStart()* and *OnStop()* or *onResume()* and *OnPause()* can avoid *IllegalArgumentException*; and committing a *FragmentTransaction* before the activity’s state has been saved (i.e., before the callback *onSaveInstanceState()*) can avoid state loss exception [90], [91].

- *Refine Conditional Checks.* Missing checks on API parameters, activity states, index values, database versions, external resources can introduce unexpected exceptions. Developers usually fix them via adding appropriate conditional checks. For example, Fig. 17a checks cursor index to fix *CursorIndexOutOfBoundsException*, Fig. 17b checks the state of the activity attached by a Fragment to fix *IllegalStateException*. Most exceptions from *Parameter Error*, *Indexing Error*, *Resource Error*, *Lifecycle Error*, and *API Error* were fixed by this strategy.

- *Move Code into Correct Thread.* Messing up UI and background threads may incur severe exceptions. The common practice to fix such problems is to move related code into correct threads. Fig. 16 fixes *CalledFromWrongThreadException* by moving the code of modifying UI widgets back to the UI thread (via *Activity#runOnUiThread()*) that creates them. Similar fixes include moving the showings of *Toast* or *AlertDialog* into the UI thread instead of the background thread since they can only be processed in the Looper of the UI thread [92], [93]. Additionally, moving extensive tasks (e.g., network access, database query) into background threads can resolve the exceptions *NetworkOnMainThreadException* and “Application Not Responding” (ANR) [94].

- *Change APIs or Design Patterns.* Developers may fix an exception by using other APIs to achieve similar functional-

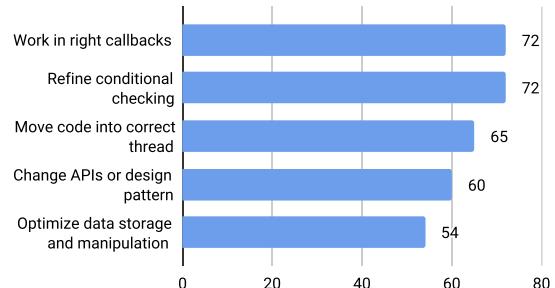


Fig. 18. Popularity of common fix practices by developers.

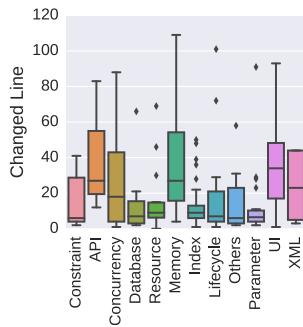


Fig. 19. Fixing in terms of number of changed lines.

ties. For example, they will replace deprecated APIs with newly imported ones. Sometimes, they directly change the design pattern to avoid exceptions, which cannot be easily fixed in the original design.

- *Optimize data storage and manipulations.* To resolve other exceptions, developers have to carefully adjust implementation algorithms, e.g., optimize data storage and manipulations. For example, to fix *OutOfMemory* caused by loading Bitmap, the common practice is to optimize memory usage by resizing the original bitmap [95]; to fix data race exceptions, the common practice is to adopt mutex locks (e.g., add *synchronized* to allow the execution of only one active thread) or back up the shared data [96].

To further understand the characteristics of developer fixes, we grouped these issues by their root causes, and computed (1) the number of code lines⁹ the developers changed to fix this issue (Fig. 19), and (2) the issue closing rate (column “Closing Rate” in Table 4). We can see that the fixes for *Parameter Error*, *Indexing Error*, *Resource Error*, and *Database Error* require fewer code changes (most patches are fewer than 20 lines). Because most of them can be fixed by refining conditional checks. We can also note *UI Update Error*, *API Updates and Compatibility Error*, *Concurrency Error*, *Memory/Hardware Error* and *XML Design Error* require larger code patches. Because fixing these issues usually require more manipulations on UI components, API compatibility, threads, memory and GUI design resources, respectively.

Further, by investigating the discussions and comments of developers when fixing, we find three important reasons that reveal the difficulties they face.

- *Difficulty of Reproducing and Validation.* One prominent difficulty is how to reproduce exceptions and validate the correctness of fixes [76]. Most users, testing tools or platforms do not report complete reproducing steps/inputs and other necessary information (e.g., exception trace, device model, code version) to developers. In most bug reports, we find only an exception trace is provided.

We surveyed the developers with Q17 “*In your experience, how much percentage of exceptions you are able to fix if you are only provided with an exception trace?*”. We find only 12 developers (8.9 percent) reported they could fix over 70 percent exception bugs (only three developers say they could fix more than 90 percent exceptions). 27 developers (20.0 percent) selected 10 ~ 30% exceptions, 54 developers (40.0 percent) selected

⁹ To reduce “noises”, we excluded comment lines (e.g., “//...”), annotation lines (e.g., “@Override”), unrelated code changes (e.g., “import *;”, the code for new features).

30 ~ 50% exceptions, and 39 developers (28.9 percent) selected 50 ~ 70% exceptions, respectively. We can see fixing exceptions could be rather difficult if only exception traces are available. In other cases, reproducing and validating non-deterministic exceptions (e.g., concurrency errors) could be harder. After fixing these issue, developers choose to leave the app users to validate before closing the issue. As shown in Table 4, concurrency errors have low fixing rate.

- *Inexperience with Android System.* A good understanding of Android system is essential to correctly fix exceptions. As the closing rates in Table 4 indicate, developers are more confused by *Memory/Hardware Error*, *Lifecycle Error*, *Concurrency Error*, and *UI Update Error*. We find some developers use simple *try-catch* or compromising ways (e.g., use *commitAllowingStateLoss* to allow activity state loss) as workarounds. However, such fixes are often fragile.

- *Fast Evolving APIs and Features.* Android is evolving fast. As reported, on average, 115 API updates occur each month [97]. Moreover, feature changes are continuously introduced. However, these updates or changes may make apps fragile when the platform they are deployed is different from the one they were built; and the developers are confused when such issues appear. For example, Android 6.0 introduces runtime permission grant — If an app uses dangerous permissions, developers have to get permissions from users at runtime. However, we find several developers choose to delay the fixing since they have not fully understand this new feature.

Answer to RQ6: Working in the right callbacks, using correct thread types, refining conditional checks, changing APIs or design patterns, optimizing data storage or manipulation are the five common fix practices. When developers fix framework exceptions, UI Update Error, API Updates and Compatibility Error, Concurrency Error, Memory/Hardware Error and XML Design Error require larger code patches. Meanwhile, reproducing exceptions and validating the fixes, understanding different mechanisms in Android system and adapting to fast-evolving Android APIs and features are the three main difficulties that developers face during fixing.

4 THE BENCHMARK DROIDDEFECTS

Based on the data and analysis results in Section 3, this section aims to construct a benchmark of exception bugs for Android apps. This benchmark can facilitate follow-up research (e.g., static fault analysis [98], fault localization [25], program repair [26]), and help measure effectiveness of proposed techniques in a controlled and systematic way.

However, constructing such a benchmark is non-trivial. First, for Android apps, most bug-triggering tests are reported in natural language, which describe the specific user actions to manifest the defects. These tests cannot be directly executed against the app to validate bugs [74]. Automatically translating these tests to executable ones are extremely difficult [78], [79]. Second, Android system and its apps are evolving fast, and use a diverse set of third-party libraries and different build systems (e.g., *Gradle*, *Ant*). These dependencies make it rather difficult to fully automate the build process, and usually involve considerable human efforts to resolve issues. Third, GUI tests can be notoriously flaky [87], [88], [89], which may not be able

to deterministically manifest the defects. Due to these challenges, we cannot follow prior benchmarking methods [99], [100] to automate the construction. To bridge the gap, we made tremendous efforts to construct this benchmark.

4.1 Android App Defect Scenario

Our bug repository, *DroidDefects*, now only considers *reproducible, crash* defects that are the bugs of apps themselves. Other defects like Android system bugs [101], third-party library bugs [102], device fragmentation bugs [64], and non-crashing bugs (e.g., performance and energy bugs [103], resource and memory leaks [57], [104], GUI failures [105], [106], security bugs [107], [108]) are not considered. To characterize an Android app bug in our context, we define the defect scenario as follows, which includes

- A *complete app project with one specific defect*, which incorporates the source code, the dependency libraries and the build scripts (e.g., *Gradle* or *Ant*). The project can be successfully compiled into an *apk* file for running on an emulator or a real device; and the defect can be deterministically reproduced by one or more tests.
- An *exception stack trace*, which is induced by the defect. It provides certain clues of the defect (e.g., the exception type, message, and the invoked methods). In particular, it tracks the sequence of called methods up to the point where the exception is thrown.
- A *bug-triggering test and its environment*, which can deterministically manifest the defect of the app, given the specific environment (e.g., API version, system configuration). The test usually is composed of a sequence of user actions and/or system events. The test can be written in the form of natural language, JUnit-based test scripts (e.g., Espresso [109], UIAutomator [86]) or low-level events (e.g., Monkey scripts [85]).
- Optionally, a *developer-written repair or patch*, which fixes the faulty behavior *w.r.t.* the defect. It can be used for understanding the defect.

4.2 Artifacts of *DroidDefects*

DroidDefects contains three main artifacts: (1) dataset of reproducible defects, (2) dataset of ground-truth defects, and (3) utility scripts.

Dataset of Reproducible Defects. This dataset now contains 33 reproducible defects from 29 Android apps, and covers 26 distinct exception types. This dataset helps researchers to understand the characteristics of different app exceptions, and enables detailed analysis. Although this dataset is relatively small, but it covers different types of exception bugs from the 11 common root causes, and provides with detailed reproducibility and root cause information. All these information has never been considered in those previously constructed dataset [26], [27], [69], [78]. We will continuously evolve and enhance this dataset to include more exception instances, although our experience indicate this process requires tremendous manual efforts [79]. Section 4.3.1 gives the setup details. For each defect, it provides:

- *Source code of faulty app version*, the complete source code of the faulty app version with the build scripts and the compiled *apk* file.

- *Reproducible tests*, the test cases that can deterministically manifest the defect (written in natural language).
- *Exception trace*, the exception trace *w.r.t.* the defect.
- *Root cause analysis*, the explanation of the defect.

In the current version of dataset, we have not yet included non-deterministic defects (e.g., data race bugs [56], [66]), since they require specific timing controls.

Dataset of Ground-Truth Defects. This dataset provides 3,696 distinct real faults from 821 apps, which cover all the 11 root causes summarized in Section 3.2. For each fault, we provide the app project source code, the executable *apk* file and the exception trace. This dataset can be used to evaluate the effectiveness of the fault detection, localization or repairing techniques at the large-scale. Section 4.3.2 details the setup.

Utility Scripts. The utility scripts contain the APIs to run existing tools, including dynamic testing tools (Monkey, Sapienz, and Stoat) and static analysis tools (Lint and FindBugs). This can ease the setup of evaluation. For example, researchers can evaluate his/her newly-proposed testing tool with the three state-of-the-art ones on our dataset via calling dedicated APIs.

4.3 Benchmark Setup Details

Apps. To construct *DroidDefects*, we chose to use open-source apps since the availability of source code enables detailed analysis. We chose app subjects from F-Droid. As discussed in Section 2.2.1, F-Droid apps are the representatives of real-world apps and most of them are maintained on GitHub.

4.3.1 Setup of Reproducible Defects

Selection Criteria. To construct a comprehensive dataset, we used *exception types* as the main guidance. Specifically, we purposely selected a number of typical Android app defects to cover each exception type from each root cause group (summarized in Section 3.2), respectively.

Source of Defects. We mainly collected Android app defects from GitHub issue repositories, since these defects may be reported with the reproduction steps and other information. We also referred to the defects used by recent literature [26], [78]. We have not considered the defects from testing tools in Section 3.4, since as revealed in Section 3.5, the reproducibility rates of generated tests are very low.

Manual Validation. To collect valid defects from GitHub issue repositories, we reused the dataset of app exceptions collected in Section 2.2.2. We used the keywords “crash/stop”, “reproduce”, “replicate”, “version” to further filter the exceptions, and only considered the issues submitted in recent years. We constrained our focus on these keywords since we hoped to focus on those fail-stop defects¹⁰ with clear reproduction steps on the specific app versions, which are quite important for manual validation and reproduction. We selected recent issues by considering Android apps could have outdated dependencies. Finally, we got 448 issues. However, by randomly inspecting some filtered issues, we note there were still many invalid ones (e.g., the

10. Note that not all exceptions can trigger app crashes, e.g., *caught exceptions* or system warning exceptions (e.g., the *WindowLeaked* exception only gives resource-leak warning without failing apps).

TABLE 6
Statistics of Ground-Truth Defects w.r.t. 11 Common Root Causes of Framework Exceptions

Category (Name for short)	#Defects	#Apps
API Updates and Compatibility (API)	33	16
XML Layout Error (XML)	66	30
API Parameter Error (Parameter)	675	181
Framework Constraint Error (Constraint)	168	95
Index Error (Index)	551	183
Database Management Error (Database)	51	15
Resource-Not-Found Error (Resource)	1,238	286
UI Update Error (UI)	170	53
Concurrency Error (Concurrency)	241	71
Component Lifecycle Error (Lifecycle)	301	160
Memory/Hardware Error (Memory)	123	63
Others (Java-specific errors)	79	40
Total	3,696	821

reproduction steps are incomplete, the keyword “version” did not match with the “app version”, etc).

Next, three authors of this paper spent one month to manually validate and reproduce these issues. Specifically, we worked in the following steps. First, we randomly sampled some candidate issues for each exception type. Second, to get the faulty code version V_{bug} , we either (1) checked out the code commit right preceding the fixed version V_{fix} if the bug fix is explicitly mentioned, or (2) checked out V_{bug} according to the specified app version or the issue submission time. Third, we built the app into an executable *apk* via build scripts or Android Studio. Last, we installed the app on an Android device to replay the described reproduction steps and observe whether the exact exception will be thrown. In our experience, various reasons may fail the above reproduction process. For example, the compilation may fail due to outdated dependencies; the exception cannot be manifested due to incomplete reproduction steps or environment issues. Therefore, if we could not successfully reproduce an exception within one hour, we resorted to the other candidates. Finally, we got 33 reproducible defects.

4.3.2 Setup of Ground-Truth Defects

To construct a large dataset of ground-truth defects, we leveraged the exceptions revealed by dynamic testing tools in Section 3.4. Table 6 shows the statistics of this dataset *w.r.t.* the root causes of framework exceptions. In total, we collected 3,696 framework exceptions across 11 common root causes, which were discovered in 821 unique Android apps. To facilitate the use, we characterized the complexity of each faulty app by number of lines, number of methods, number of activities, and number of classes, and the diversity by the app category. Finally, we got 3,696 ground-truth defects.

5 APPLICATIONS OF OUR STUDY

5.1 Improving Exception Detection

Dynamic Testing. Enhancing testing tools to detect specific errors is very important. For example, (1) *Generate meaningful as well as corner-case inputs to reveal parameter errors.* We find random strings with specific formats or characters are very likely to reveal unexpected crashes. For instance, Monkey

detects more *SQLiteDatabase*s than the other tools since it can generate strings with special characters like “”” and “%” by randomly hitting the keyboard. When these strings are used in SQL statements, they can fail SQL queries without escaping. (2) *Enforce environment interplay to reveal lifecycle, concurrency and UI update errors.* We find some special actions, e.g., change device orientations, start an activity and quickly return back without waiting it to finish, put the app at background for a long time (by calling another app) and return back to it again, can affect an app’s internal states and its component lifecycle. Therefore, these actions can be interleaved with normal UI actions to effectively check robustness. (3) *Consider different app and SDK versions to detect regression errors.* We find app updates may introduce unexpected errors. For example, as shown in Fig. 8, the changes of database scheme can crash the new version since the developers have not carefully managed database migration from the old version. (4) *More advanced testing criteria* [110], [111] are desired to derive effective tests.

Static Analysis. Incorporating new checking rules into static analysis tools to enhance their abilities is highly valuable. We find FindBugs and SonarQube have not included any Android-specific checking rules, while PMD defines three rules [112], although these tools all support checking Android projects. Lint defines 281 Android rules [113] but only covers a small portion of framework-specific bugs [29]. However, there are plausible ways to improving these tools. For example, to warn the potential crash in Fig. 7, static analysis can check whether the task running in the thread uses Handler to dispatch messages, if it uses, *Looper#prepare()* must be called at the beginning of *Thread#run()*; to warn the potential crash in Fig. 5, static analysis can check whether there is an appropriate checking on activity state before showing a dialog from a background thread. In fact, some work [98] already implements the lifecycle checking in Lint.

Demonstration of Usefulness. We implemented Stoa+, an enhancement version of Stoa [12] with two new strategies. These two strategies include eight enhancement cases: (1) *five* specific input formats (e.g., empty string, lengthy string, null) or characters (e.g., “”, “%”) to fill in *EditTexts* or *Intent*’s fields; (2) *three* specific types of *environment-interplay* actions mentioned in Section 5.1. These two strategies were implemented in the MCMC sampling phase of Stoa, and randomly inject these specific events into normal GUI tests to improve fault detection ability (see Section 4.4 in [12]). We applied Stoa+ on dozens of most popular apps (e.g., Facebook, Gmail, Google+, WeChat) from Google Play, and each app was tested for ten hours on a Google Pixel 3 device. At last, we successfully detected 3 previously unknown bugs in Gmail (one parameter error) and Google+ (one UI update error and one lifecycle error). All of these bugs were detected in the latest versions at the time of our study, and have been reported to Google and got confirmed. The detailed issue reports were available at the Stoa’s website [114]. However, these bugs were not found by Monkey and Sapienz, while other testing tools, e.g., CrashScope [115] and AppDoctor [116], only consider 2 and 3 of these 8 enhancement cases, respectively.

5.2 Enabling Exception Localization

We find developers usually take days to fix a framework exception. Thus, automatically locating faulty code and

proposing possible fixes are highly desirable. Our study can shed light on this goal.

Demonstration of Usefulness. We built a framework exception localization tool, ExLocator, based on Soot [117], which takes as input an APK file and an exception trace, and outputs a report that explains the root cause of this exception. It currently supports 5 exception types from UI Update, Lifecycle, Index, and Framework Constraint errors. In detail, it first extracts method call sequences and exception information from the exception trace, and classifies the exception into one of our summarized fault categories according to the root exception and signalers. As shown in Section 3.2, these specific exception types have obvious fault patterns (e.g., incorrect handling background threads). Exlocator utilizes these patterns and data-/control-flow analysis to locate the root cause. More technical details can be found in our descendant tool APEchecker [81], which automatically localizes UI update errors. The report gives the lines or methods that causes the exception, the description of the root cause and possible fixing solutions, and closely related Stack Overflow posts. From our benchmark *DroidDefects*, we randomly selected 6 exception cases for each of five supported exception types. At last, we got 30 exception cases in total. ExLocator was successfully able to locate 28 exceptions out of 30 (93.3 percent precision) by comparing with the patches by developers. By incorporating additional context information from Android framework (e.g., which framework classes use Handler), our tool successfully identified the root causes of the remaining two cases. However, all previous fault localization work [25], [118], [119], [120] can only handle generic exception types.

5.3 Enhancing Mutation Testing

Mutation testing is a widely-adopted technique to assess the fault-detection ability of a test suite, as well as to guide test case generation and prioritization [121]. One crucial step of applying mutation testing in a new application domain (e.g., Android apps) is to design specific mutation operators, which can represent typical programming faults, in addition to those generic mutation operators. For example, a number of mutation testing tools for Java programs (e.g., Pit [122] and Major [123]) are available, but they do not include any Android-specific mutation operators. As a result, they may generate trivial mutants that may directly crash themselves when startup or cannot be compiled into executables.

We identified 75 different exception instances (with unique exception types and messages) from the data in Table 4. But we find existing mutation operators [124], [125], [126], [127], [128] designed for Android apps only cover a few of these instances. Specifically, only 4 mutation operators (i.e., *Intent Payload Replacement*, *Activity/Service Lifecycle Method Deletion*, *Fail on Back*) of Deng *et al.*'s 17 operators [125], [126] may help reveal some specific framework exceptions (e.g., lifecycle-related issues). Their remaining operators focus on detecting UI, event handling and energy failures instead of fatal crashes. MDroid+ [127] proposes 38 operators, but can only cover 8 exception instances in our study. Based on the results of our study, researchers could add more mutation operators. For example, we can delete Activity state checking statements from those methods

running in background threads to inject Lifecycle errors (see Fig. 5); we can also remove specific statements (e.g., app state storage) from some Fragment's lifecycle callbacks (e.g., `onSaveInstanceState`) to inject state loss errors [90], [91]; we can also change some data access from UI threads to background threads to inject UI update errors (see Fig. 6). We can also inject many Framework constraint errors (e.g., see the example in Fig. 7). All these generated mutants can be successfully compiled and only detectable at runtime with specific GUI tests. Thus, more mutation operators can be introduced for framework exception types to improve mutation testing of Android apps.

6 DISCUSSION

6.1 Lessons Learned

We have learned several lessons from this study. We summarize them to inspire practitioners and researchers, and motivate future work.

Automatically Reproducing Exceptions Need More Research Efforts. Reproducing exceptions is very important for bug diagnosing and fixing. First, in practice, only (incomplete) reproduction steps (written in natural language) or exception traces are available to developers. Although some tools [75], [78], [79], [115] have been developed to improve or automate bug reproduction, their effectiveness and usability are still limited. CrashScope [115] improves the reproducibility by recording more contextual information of bug-triggering event sequences. However, it still cannot handle exception bugs caused by inter-app communications. Yukusu [78] translates a bug report written in natural language into executable test cases. However, according to our replication of their evaluation, we find Yukusu still focuses on creating test cases instead of reproducing the expected bugs. RecDroid [79] is a further step of Yukusu, which aims to automatically reproduce the expected crash bugs from a bug report. However, it cannot cover all types of exception bugs (e.g., concurrency bugs) and its ability is limited by its predefined grammar patterns. Thus, how to effectively and faithfully reproducing the intended bug described in a bug report still requires more research efforts. Second, how to reproduce an exception with a short event trace is also important [129], [130]. Existing testing tools (e.g., Monkey and Sapienz) usually generate quite long traces which are flaky and not suitable for reproducing. However, when applied for GUI programs, existing test reduction techniques, e.g., delta debugging [129], [131], still have high performance overhead. Thus, how to efficiently reduce bug-triggering tests is still an open problem. Third, if only an exception trace is available, effective techniques for locating faulty code and then generating the bug-triggering tests at the UI level are quite useful for bug reproduction. However, existing fault localization techniques for Android apps [25] are far from mature, and limited to trivial types of exceptions. Little research efforts has been done to link the app logic code with UI widgets for interactive debugging of Android apps. This deserves more research efforts.

Effective Bug Detection Tools are in Great Demand. Both dynamic and static bug detection techniques are needed to effectively reveal as many exception bugs as possible before app release. First, Android apps could be complicated and have different types of bugs, and different testing strategies

could have very different performance in detecting exceptions. Thus, one plausible idea is to combine the strengthens of these strategies together, e.g., combining random testing and systematic GUI exploration [70], or using static analysis to guide dynamic testing [81]. Second, static analysis tools could include more specific rules to check potential bugs and keep update with the evolution of Android system. The rules that are closely related to Android programming errors (e.g., Component Lifecycle Error, Framework Constraint Error, UI Update Error) could have higher fault detection abilities. Third, bug detection tools should improve their usability. For example, dynamic testing tools should provide mechanisms to automatically bypass user logins or accept user-provided account information, otherwise, they are likely trapped at the login pages. Other tool features are also very useful, e.g., leveraging user-provided oracles, generating more readable and less flaky tests, reducing number of false positives. These can improve bug detection and reproduction, and save debugging efforts.

Better Documentation and Technical Tutorials are Needed. A comprehensive and intuitive technical documentation is very important for developers to quickly understand Android system and avoid programming errors. However, during this study, we find this issue is still prominent. For example, we notice developers are more capable of fixing trivial errors (e.g., Parameter Error, Index Error) according to their Java programming knowledge, but takes more time and needs more discussions when fixing such Android-specific issues as Component Lifecycle, Memory/Hardware, Concurrency, and UI Update errors. However, some sophisticated mechanism are not well documented in the official Android documentation. One typical example is about the state loss issue when handling Activity and Fragment lifecycle [90]. Junior developers have to refer to those technical posts from experienced developers.

Second, we find some developers cannot quickly get familiar with the newly-introduced features. We observe some developers chose to delay the upgrading of their apps to new Android platforms. For example, Android introduces runtime permission granting since API 23; and supports Kotlin since API 27. Better documentation and training courses should be continuously updated to help developers gain more understanding of new mechanisms, and let them know the feature evolution of Android system.

6.2 Threats to Validity

External Validity. First, our selected apps may not be the representatives of all possible real-world apps. To counter this, we collected all 2,486 apps from F-Droid at the time of our study, which is the largest database of open-source apps, and covers diverse app categories. We also collected a diverse set of 3,230 closed-source Google Play apps as subjects. Second, our mined exceptions may not include all possible exceptions. To counter this, we mined the issue repositories of 2,174 apps on GitHub and Google Code; and applied testing tools on 5,334 unique apps, which leads to total 30,009 exceptions. To our knowledge, this is the largest study for analyzing Android app exceptions.

Internal Validity. First, our exception analysis may not be absolutely complete and correct. For completeness, (i) we

investigated 8,243 framework exceptions, and carefully inspected all common exception types. (ii) We surveyed previous work [11], [12], [15], [19], [21], [31], [56], [98], [116], [132], [133], [134], [135] that reported exceptions, and observed all their exception types and patterns were covered by our study. For correctness, we cross-validated our analysis on each exception type, and also referred to the patches from developers and Stack Overflow posts. More importantly, we surveyed 135 professional app developers to gain more understandings and insights to validate our analysis. Second, the classification of app exceptions (Section 3.1) and the taxonomy of root causes (Section 3.2) may be subjective. This may affect the validity of some analysis results. To counter this, we carefully analyzed these exceptions based on our understanding, and the knowledge from Android documentation and development tutorials.

Construct Validity. The online developer survey may have some limitations. The designed questions may not fully cover all aspects, and affect the validity of our conclusions drawn from this survey. But we tried our best to design appropriate questions, and refined these questions according to early feedback from three experienced Android developers and our own long-time research experience of inspecting developers activities on GitHub. In the survey, we also provided some questions with open options to receive any comments from developers, which complemented our provided options. Our constructed benchmark may also subject to construct validity. To counter this, we manually verified all reproducible cases. For ground-truth cases, we also automatically checked the validity of exception traces.

7 RELATED WORK

A number of fault studies exist in the literature for Android apps from different perspectives, e.g., performance [103], energy [136], compatibility issues [64], [137], permission issues [138], memory leak [139], [140], GUI failures [105], [106], [141], resource usage [57], [104], API stability [97], security [142], [143], [144] etc. However, none of these work particularly focuses on app crashes and exceptions, which is the main goal we target at in this work.

Hu *et al.* [132] make one of the first attempts to analyze functional bugs for Android apps. They manually classify 8 bug types (e.g., *Activity* errors, *Event* errors, *Type* errors) from 158 bug reports of 10 apps. Other efforts include [31], [134], which however have different goals compared to our study: Coelho *et al.* [31] analyze exception traces to investigate the bug hazards of exception-handling code (e.g., cross-type exception wrapping), Zaeem *et al.* [134] study 106 bugs of 13 apps to generate testing oracles for a specific set of bug types. However, none of them give a large-scale and comprehensive analysis in this direction, and the validity of their conclusions is also unclear.

Linares-Vásquez *et al.* [127] recently also analyze a large number of android app bugs. But our study is significantly different from theirs. First, we focus on analyzing crash bugs caused by framework exceptions, while they focus on designing mutation operators to evaluate the effectiveness of test suites. Second, we give a much more comprehensive, deep analysis on the root causes, exception detection, reproduction and fixing.

Based on our dataset and analysis results, we constructed the benchmark *DroidDefects*. Although prior work also construct some benchmarks of Android app faults, our benchmark is more systematic in the number of faults, exception types and root causes. For example, *AndroTest* [69], [145] is a dataset of 68 apps collected from early research work [13], [15], [18], [133], to evaluate the fault detection abilities of Android app testing tools. But these subjects are randomly selected from F-Droid without any specific selection criteria. Many of these apps are quite out-of-date and error-prone. *DroidBugs* [27], [146], the only available dataset for automated program repair of Android apps, merely contains 13 bugs from 5 apps. This dataset is introductory, and has not provided any information about bug types.

Researchers have also constructed benchmarks for other bug types. MUBench [147] is a benchmark of 89 API misuses mined from 33 real-world projects, including Android. AppLeak [148] is a benchmark of 40 resource leak bugs in Android apps, which contains the faulty apps, bug-fixed versions (when available), and reproducible test cases. Mostafa *et al.* [149] study behavioral backward incompatibilities of Java software libraries, including Android. They archived a number of backward incompatibility faults. In contrast, our work focus on exception bugs, and covers diverse categories and root causes.

8 CONCLUSION

In this paper, we conducted the first large-scale, comprehensive study to understand framework exceptions of Android apps, which account for the majority of app exception bugs. Specifically, we investigated framework exceptions from several perspectives, including exception characteristics, root causes, testing practice of developers, abilities of existing bug detection tools, exception reproducibility and common fix practices. To validate and generalize our analysis results, we considered both open-source and commercial apps, and further conducted an online developer survey to gain more insights from the developers' knowledge and experiences. Through this study, we constructed *DroidDefects*, the first comprehensive and largest benchmark of exception bugs, to enable follow-up research; and built two prototype tools, Stoat+ and ExLocator, to demonstrate the usefulness of our findings. We pointed a number of research directions that deserve more research efforts.

ACKNOWLEDGMENTS

We would like to thank the constructive and valuable comments from the TSE reviewers. We also appreciate the Android app developers who participate in our online survey, and share us many experience and feedback in this field. This work was partially supported by SNSF Spark Project CRSK-2_190302; partially supported by NSFC Project 61632005 and 61532019; partially supported by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2018NCR-NCR005-0001), the Singapore National Research Foundation under NCR Award Number NRF2018NCR-NSOE004-0001, and NRF Investigatorship NRFI06-2020-0022.

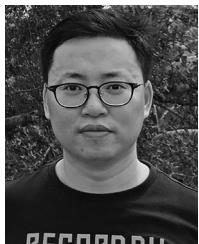
REFERENCES

- [1] "Number of Android applications," 2018. [Online]. Available: <http://www.appbrain.com/stats/number-of-android-apps>
- [2] S. Chen *et al.*, "Storydroid: Automated generation of storyboard for android apps," in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 596–607.
- [3] H. Khalid, E. Shihab, M. Nagappan, and A. E. Hassan, "What do mobile app users complain about?" *IEEE Softw.*, vol. 32, no. 3, pp. 70–77, May/Jun 2015.
- [4] X. Xia, E. Shihab, Y. Kamei, D. Lo, and X. Wang, "Predicting crashing releases of mobile applications," in *Proc. 10th ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2016, pp. 29:1–29:10.
- [5] "Robotium," 2020. [Online]. Available: <http://www.robotium.org>
- [6] "Appium," 2020. [Online]. Available: <http://appium.io/>
- [7] "Android Lint," 2020. [Online]. Available: <https://developer.android.com/studio/write/lint.html>
- [8] "FindBugs," 2020. [Online]. Available: <http://findbugs.sourceforge.net/>
- [9] P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, "Understanding the test automation culture of app developers," in *Proc. IEEE 8th Int. Conf. Softw. Testing Verification Validation*, 2015, pp. 1–10.
- [10] M. Linares-Vásquez, C. Vendome, Q. Luo, and D. Poshyvanyk, "How developers detect and fix performance bottlenecks in android apps," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2015, pp. 352–361.
- [11] K. Mao, M. Harman, and Y. Jia, "Sapienz: Multi-objective automated testing for android applications," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 94–105.
- [12] T. Su *et al.*, "Guided, stochastic model-based GUI testing of android apps," in *Proc. 11th Joint Meeting Foundations Softw. Eng.*, 2017, pp. 245–256.
- [13] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proc. ACM SIGSOFT 20th Int. Symp. Foundations Softw. Eng.*, 2012, Art. no. 59.
- [14] H. van der Merwe, B. van der Merwe, and W. Visser, "Verifying android applications using java pathfinder," *SIGSOFT Softw. Eng. Notes*, vol. 37, no. 6, pp. 1–5, 2012.
- [15] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for Android apps," in *Proc. 9th Joint Meeting Foundations Softw. Eng.*, 2013, pp. 224–234.
- [16] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of Android apps," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Languages Appl.*, 2013, pp. 641–660.
- [17] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated GUI-model generation of mobile applications," in *Proc. 16th Int. Conf. Fundam. Approaches Softw. Eng.*, 2013, pp. 250–265.
- [18] W. Choi, G. C. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Languages Appl.*, 2013, pp. 623–640.
- [19] R. Mahmood, N. Mirzaei, and S. Malek, "EvoDroid: Segmented evolutionary testing of Android apps," in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Languages Appl.*, 2014, pp. 599–609.
- [20] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "PUMA: Programmable UI-automation for large-scale dynamic analysis of mobile apps," in *Proc. 12th Annu. Int. Conf. Mobile Syst. Appl. Services*, 2014, pp. 204–217.
- [21] D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR: Automated model-based testing of mobile apps," *IEEE Softw.*, vol. 32, no. 5, pp. 53–59, Sept.-Oct. 2015.
- [22] T. Su, "FSMdroid: Guided GUI testing of android apps," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. Companion*, 2016, pp. 689–691.
- [23] T. Gu *et al.*, "Aimdroid: Activity-insulated multi-level automated testing for android applications," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 103–114.
- [24] W. Song, X. Qian, and J. Huang, "EHBDroid: Beyond GUI testing for android applications," in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 27–37.
- [25] H. Mirzaei and A. Heydarnoori, "Exception fault localization in android applications," in *Proc. 2nd ACM Int. Conf. Mobile Softw. Eng. Syst.*, 2015, pp. 156–157.
- [26] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 187–198.

- [27] L. Azevedo, A. Dantas, and C. G. Camilo-Junior, "Droidbugs: An android benchmark for automatic program repair," *CoRR*, vol. abs/1809.07353, 2018. [Online]. Available: <http://arxiv.org/abs/1809.07353>
- [28] "Monkey," 2020. [Online]. Available: <http://developer.android.com/tools/help/monkey.html>
- [29] L. Fan *et al.*, "Large-scale analysis of framework-specific exceptions in android apps," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 408–419.
- [30] "Android developers documentation," 2020. [Online]. Available: <https://developer.android.com/reference/packages.html>
- [31] R. Coelho, L. Almeida, G. Gousios, and A. van Deursen, "Unveiling exception handling bug hazards in android based on github and google code issues," in *Proc. 12th Working Conf. Mining Softw. Repositories*, 2015, pp. 134–145.
- [32] "F-Droid," 2020. [Online]. Available: <https://f-droid.org/>
- [33] "Google play store," 2020. [Online]. Available: <https://play.google.com/store/apps>
- [34] "EMMA," 2020. [Online]. Available: <http://emma.sourceforge.net/>
- [35] "JaCoCo," 2020. [Online]. Available: <http://www.eclemma.org/jacoco/>
- [36] "Amazon mechanical turk," 2020. [Online]. Available: <https://www.mturk.com>
- [37] "CodePath android cliffnotes," 2020. [Online]. Available: <http://guides.codepath.com/android>
- [38] "Advanced android development," 2020. [Online]. Available: <https://developer.android.com/courses/advanced-training/overview>
- [39] E. Derr, S. Bugiel, S. Fahl, Y. Acar, and M. Backes, "Keep me updated: An empirical study of third-party library updatability on android," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2187–2200.
- [40] G. Nunez, "Party pooper: Third-party libraries in android," Tech. Rep. School Elect. Eng. Comput. Sci. Univ. California, Berkeley, 2011.
- [41] I. G. W. Group, "Standard glossary of terms used in software testing," *Int. Softw. Testing Qualifications Board*, pp. 1–25, 2015.
- [42] "Root cause," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Root_cause
- [43] "Activity lifecycle," 2020. [Online]. Available: https://developer.android.com/guide/components/activities/activity-life_cycle.html
- [44] "Bankdroid," 2020. [Online]. Available: <https://github.com/liato/android-bankdroid>
- [45] "Bankdroid revision 8b31cd3," 2020. [Online]. Available: <https://github.com/liato/android-bankdroid/commit/8b31cd36fab5ff746ed5a2096369f9990de7b064>
- [46] "Fragments," 2020. [Online]. Available: <https://developer.android.com/guide/components/fragments.html>
- [47] "cgeo," 2020. [Online]. Available: <https://github.com/cgeo/cgeo>
- [48] "cgeo revision d6b4e4d," 2020. [Online]. Available: <https://github.com/cgeo/cgeo/commit/d6b4e4d958568ea04669f511a85f24ac08f524b6>
- [49] "Handler," 2020. [Online]. Available: <https://developer.android.com/reference/android/os/Handler.html>
- [50] "Looper," 2020. [Online]. Available: <https://developer.android.com/reference/android/os/Looper.html>
- [51] "Local-GSM-Backend," 2020. [Online]. Available: <https://github.com/n76/Local-GSM-Backend>
- [52] "PhoneStateListener," 2020. [Online]. Available: http://grepcode.com/file/repository.grepcode.com/java/ext/com.google.android/android/4.3.1_r1/android/telephony/PhoneStateListener.java#PhoneStateListener.0mHandler
- [53] "Local-GSM-Backend revision 07e4a759," 2020. [Online]. Available: <https://github.com/n76/Local-GSM-Backend/commit/07e4a759392c6f2c0b28890f96a177cb211ffc2d>
- [54] "NetworkOnMainThreadException," 2020. [Online]. Available: <https://developer.android.com/reference/android/os/NetworkOnMainThreadException.html>
- [55] "Requesting permissions at runtime," 2020. [Online]. Available: <https://developer.android.com/training/permissions/requesting.html>
- [56] P. Bielik, V. Raychev, and M. Vechev, "Scalable race detection for android applications," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program. Syst. Languages Appl.*, 2015, pp. 332–348.
- [57] Y. Liu *et al.*, "Droidleaks: A comprehensive database of resource leaks in android apps," *Empir. Softw. Eng.*, vol. 24, no. 6, pp. 3435–3483, 2019.
- [58] "Nextcloud notes," 2020. [Online]. Available: <https://github.com/stefan-niedermann/nextcloud-notes>
- [59] "Nextcloud notes issue," 2020. [Online]. Available: <https://github.com/stefan-niedermann/nextcloud-notes/issues/199>
- [60] "Nextcloud notes revision," 2020. [Online]. Available: <https://github.com/stefan-niedermann/nextcloud-notes/pull/212/commits/a1a97292b5f7511473282cc40f23e786f019d7f>
- [61] "Atarashii," 2020. [Online]. Available: <https://github.com/AnimeNeko/Atarashii>
- [62] "Atarashii revision b311ec3," 2020. [Online]. Available: <https://github.com/AnimeNeko/Atarashii/commit/b311ec327413aa4ef4aaabb8a8496c61d342cfe9>
- [63] "JDK 7 compatibility issues," 2020. [Online]. Available: <http://kb.yworks.com/article/550/>
- [64] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 226–237.
- [65] L. Wei, Y. Liu, S.-C. Cheung, H. Huang, X. Lu, and X. Liu, "Understanding and detecting fragmentation-induced compatibility issues for android apps," *IEEE Trans. Softw. Eng.*, to be published.
- [66] C. Hsiao *et al.*, "Race detection for event-driven mobile applications," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 326–336.
- [67] M. L. Vásquez, C. Bernal-Cárdenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 613–622.
- [68] "MonkeyRunner," [Online]. Available: <https://developer.android.com/studio/test/monkeyrunner/>
- [69] S. R. Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (E)," in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2015, pp. 429–440.
- [70] W. Wang *et al.*, "An empirical study of android test generation tools in industrial cases," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 738–748.
- [71] A. Arcuri and L. C. Briand, "A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering," *Softw. Test., Verif. Reliab.*, vol. 24, no. 3, pp. 219–250, 2014.
- [72] "Mann-Whitney U test," 2020. [Online]. Available: https://en.wikipedia.org/wiki/Mann-Whitney_U_test
- [73] E. Kowalczyk, M. B. Cohen, and A. M. Memon, "Configurations in android testing: They matter," in *Proc. 1st Int. Workshop Advances Mobile App Anal.*, 2019, pp. 1–6.
- [74] K. Moran, M. L. Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "Auto-completing bug reports for android applications," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 673–686.
- [75] K. Moran, M. L. Vásquez, C. Bernal-Cárdenas, and D. Poshyvanyk, "FUSION: A tool for facilitating and augmenting android bug reporting," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng. Companion*, 2016, pp. 609–612.
- [76] K. Moran, M. L. Vásquez, C. Bernal-Cárdenas, C. Vendome, and D. Poshyvanyk, "Automatically discovering, reporting and reproducing android application crashes," in *Proc. IEEE Int. Conf. Softw. Testing Verification Validation*, 2016, pp. 33–44.
- [77] M. White, M. L. Vásquez, P. Johnson, C. Bernal-Cárdenas, and D. Poshyvanyk, "Generating reproducible and replayable bug reports from android application crashes," in *Proc. IEEE 23rd Int. Conf. Program Comprehension*, 2015, pp. 48–59.
- [78] M. Fazzini, M. Prammer, M. d'Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2019, pp. 141–152.
- [79] Y. Zhao *et al.*, "Recdroid: Automatically reproducing android application crashes from bug reports," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 128–139.
- [80] M. Gómez, R. Rouvoy, B. Adams, and L. Seinturier, "Reproducing context-sensitive crashes of mobile apps using crowdsourced monitoring," in *Proc. IEEE/ACM Int. Conf. Mobile Softw. Eng. Syst.*, 2016, pp. 88–99.
- [81] L. Fan *et al.*, "Efficiently manifesting asynchronous programming errors in android apps," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 486–497.
- [82] "ACRA: Application crash reports for android," 2020. [Online]. Available: <https://github.com/ACRA/acra>
- [83] "Google analytics for firebase," 2020. [Online]. Available: <https://firebase.google.com/products/analytics/>

- [84] "Monitor the performance and usage of your Android, iOS apps with splunk enterprise," 2020. [Online]. Available: <https://mint.splunk.com/>
- [85] "MonkeyScript," 2020. [Online]. Available: https://android.googlesource.com/platform/development/+/android-4.2.2_r1/cmds/monkey/src/com/android/commands/monkey/MonkeySourceScript.java
- [86] "UIAutomator," 2020. [Online]. Available: <https://developer.android.com/training/testing/ui-automator>
- [87] A. M. Memon and M. B. Cohen, "Automated testing of GUI applications: Models, tools, and controlling flakiness," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 1479–1480.
- [88] Q. Luo, F. Hariri, L. Eloussi, and D. Marinov, "An empirical analysis of flaky tests," in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2014, pp. 643–653.
- [89] S. Thorve, C. Sreshtha, and N. Meng, "An empirical study of flaky tests in android apps," in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2018, pp. 534–538.
- [90] "Fragment transactions and activity state loss," 2020. [Online]. Available: <http://www.androiddesignpatterns.com/2013/08/fragment-transaction-commit-state-loss.html>
- [91] Z. Shan, T. Azim, and I. Neamtiu, "Finding resume and restart errors in android applications," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program. Syst. Languages Appl.*, 2016, pp. 864–880.
- [92] "NextGIS mobile revision 2ef12a7," 2020. [Online]. Available: https://github.com/nextgis/android_gisapp/commit/2ef12a75eda6ed1c39a51e2ba18039cc571e5b0e
- [93] "WordPress revision 663ce5c," 2020. [Online]. Available: <https://github.com/wordpress-mobile/WordPress-Android/commit/663ce5c1bbd739f29f6c23d9ecacbd666e4f806f>
- [94] "Keeping your app responsive," 2020. [Online]. Available: <https://developer.android.com/training/articles/perf-anr.html>
- [95] "Managing bitmap memory," 2020. [Online]. Available: <https://developer.android.com/topic/performance/graphics/manage-memory.html>
- [96] "MPDroid issue," 2020. [Online]. Available: <https://github.com/abarisain/dmix/issues/286>
- [97] T. McDonnell, B. Ray, and M. Kim, "An empirical study of API stability and adoption in the android ecosystem," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2013, pp. 70–79.
- [98] S. GRAZIUSI, "Lifecycle and event-based testing for android applications," Master's thesis, School Ind. Eng. Inf., Politecnico, 2016.
- [99] R. Just, D. Jalali, and M. D. Ernst, "Defects4j: A database of existing faults to enable controlled testing studies for java programs," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 437–440.
- [100] C. Le Goues *et al.*, "The manybugs and introclass benchmarks for automated repair of C programs," *IEEE Trans. Softw. Eng.*, vol. 41, no. 12, pp. 1236–1256, Dec. 2015.
- [101] A. K. Maji, K. Hao, S. Sultana, and S. Bagchi, "Characterizing failures in mobile oses: A case study with android and symbian," in *Proc. IEEE 21st Int. Symp. Softw. Rel. Eng.*, 2010, pp. 249–258.
- [102] J. Kochhar, J. Keng, and T. Biying, "An empirical study on bug reports of android 3rd party libraries," *Singapore Manage. Univ.*, 2013.
- [103] Y. Liu, C. Xu, and S. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1013–1024.
- [104] Y. Liu, C. Xu, S.-C. Cheung, and V. Terragni, "Understanding and detecting wake lock misuses for android applications," in *Proc. 24th ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2016, pp. 396–409.
- [105] C. Q. Adamsen, G. Mezzetti, and A. Møller, "Systematic execution of android test suites in adverse conditions," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 83–93.
- [106] D. Amalfitano, V. Riccio, A. C. R. Paiva, and A. R. Fasolino, "Why does the orientation change mess up my android application? from GUI failures to code faults," *Softw. Test., Verif. Rel.*, vol. 28, no. 1, 2018, Art. no. e1654.
- [107] S. Chen *et al.*, "Are mobile banking apps secure? what can be improved?" in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 797–802.
- [108] S. Chen *et al.*, "An empirical assessment of security risks of global android banking apps," in *Proc. Int. Conf. Softw. Eng.*, 2020, pp. 596–607.
- [109] "Espresso," 2020. [Online]. Available: <https://developer.android.com/training/testing/espresso/>
- [110] N. P. B. Jr., "Data flow oriented UI testing: Exploiting data flows and UI elements to test android applications," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2017, pp. 432–435.
- [111] T. Su *et al.*, "A survey on data-flow testing," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 5:1–5:35, Mar. 2017.
- [112] "PMD rules," 2020. [Online]. Available: <https://pmd.github.io/pmd-5.8.1/pmd-java/rules/java/android.html>
- [113] "Android lint checks," 2020. [Online]. Available: <http://tools.android.com/tips/lint-checks>
- [114] "Stoat," 2020. [Online]. Available: <https://github.com/tingsu/Stoat>
- [115] K. Moran, M. L. Vásquez, C. Bernal-Cádenas, C. Vendome, and D. Poshyvanyk, "Crashscope: A practical tool for automated testing of android applications," in *Proc. IEEE/ACM 39th Int. Conf. Softw. Eng. Companion*, 2017, pp. 15–18.
- [116] G. Hu, X. Yuan, Y. Tang, and J. Yang, "Efficiently, effectively detecting mobile app bugs with AppDoctor," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2014, pp. 18:1–18:15.
- [117] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot-a java bytecode optimization framework," in *Proc. Conf. Centre Advanced Studies Collaborative Res.*, 1999, Art. no. 13.
- [118] S. Sinha, H. Shah, C. Görg, S. Jiang, M. Kim, and M. J. Harrold, "Fault localization and repair for java runtime exceptions," in *Proc. 9th Eur. Conf. Comput. Syst.*, 2009, pp. 153–164.
- [119] S. Jiang, H. Zhang, Q. Wang, and Y. Zhang, "A debugging approach for java runtime exceptions based on program slicing and stack traces," in *Proc. 10th Int. Conf. Quality Softw.*, 2010, pp. 393–398.
- [120] R. Wu, H. Zhang, S.-C. Cheung, and S. Kim, "Crashlocator: Locating crashing faults based on crash stacks," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 204–214.
- [121] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep./Oct. 2011.
- [122] "PIT," 2020. [Online]. Available: <http://pitest.org/>
- [123] R. Just, "The major mutation framework: Efficient and scalable mutation analysis for java," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 433–436.
- [124] L. Deng, N. Mirzaei, P. Ammann, and J. Offutt, "Towards mutation analysis of android apps," in *Proc. IEEE 8th Int. Conf. Softw. Testing Verification Validation Workshops*, 2015, pp. 1–10.
- [125] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation operators for testing android apps," *Inf. Softw. Technol.*, vol. 81, pp. 154–168, 2017.
- [126] L. Deng, J. Offutt, and D. Samudio, "Is mutation analysis effective at testing android apps?" in *Proc. IEEE Int. Conf. Softw. Quality Rel. Secur.*, 2017, pp. 86–93.
- [127] M. Linares-Vásquez *et al.*, "Enabling mutation testing for android apps," in *Proc. 11th Joint Meeting Foundations Softw. Eng.*, 2017, pp. 233–244.
- [128] K. Moran *et al.*, "Mdroid+: A mutation testing framework for android," in *Proc. 40th Int. Conf. Softw. Eng.: Companion Proc.*, 2019, pp. 33–36.
- [129] L. Clapp, O. Bastani, S. Anand, and A. Aiken, "Minimizing GUI event traces," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 422–434.
- [130] W. Choi, K. Sen, G. C. Necula, and W. Wang, "Detreduce: minimizing android GUI test suites for regression testing," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.*, 2019, pp. 445–455.
- [131] B. Jiang, Y. Wu, T. Li, and W. K. Chan, "Simplydroid: efficient event sequence simplification for android application," in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 297–307.
- [132] C. Hu and I. Neamtiu, "Automating GUI testing for android applications," in *Proc. 6th Int. Workshop Autom. Softw. Test.*, 2011, pp. 77–83.
- [133] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. D. Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2012, pp. 258–261.
- [134] R. N. Zaem, M. R. Prasad, and S. Khurshid, "Automated generation of oracles for testing user-interaction features of mobile apps," in *Proc. IEEE 7th Int. Conf. Softw. Testing Verification Validation*, 2014, pp. 183–192.
- [135] L. Fan, S. Chen, L. Xu, Z. Yang, and H. Zhu, "Model-based continuous verification," in *Proc. 23rd Asia-Pacific Softw. Eng. Conf.*, 2016, pp. 81–88.

- [136] A. Banerjee, H.-F. Guo, and A. Roychoudhury, "Debugging energy-efficiency related field failures in mobile apps," in *Proc. IEEE/ACM Int. Conf. Mobile Softw. Eng. Syst.*, 2016, pp. 127–138.
- [137] H. Huang, L. Wei, Y. Liu, and S. Cheung, "Understanding and detecting callback compatibility issues for android applications," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2019, pp. 532–542.
- [138] A. Sadeghi, R. Jabbarvand, and S. Malek, "Patdroid: Permission-aware GUI testing of android," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 220–232.
- [139] H. Shahriar, S. North, and E. Mawangi, "Testing of memory leak in android applications," in *Proc. IEEE 15th Int. Symp. High-Assurance Syst. Eng.*, 2014, pp. 176–183.
- [140] G. Santhanakrishnan, C. Cargile, and A. Olmsted, "Memory leak detection in android applications based on code patterns," in *Proc. Int. Conf. Inf. Soc.*, 2016, pp. 133–134.
- [141] J. Hu, L. Wei, Y. Liu, S. Cheung, and H. Huang, "A tale of two cities: How webview induces bugs to android applications," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2019, pp. 702–713.
- [142] W. Enck, D. Oeteau, P. McDaniel, and S. Chaudhuri, "A study of android application security," in *Proc. 20th USENIX Conf. Secur.*, 2011, pp. 21–21.
- [143] S. Chen, L. Fan, C. Chen, M. Xue, Y. Liu, and L. Xu, "GUI-squatting attack: Automated generation of android phishing apps," *IEEE Trans. Dependable Secure Comput.*, to be published.
- [144] C. Tang S. et al., "A large-scale empirical study on industrial fake apps," in *Proc. 41st Int. Conf. Softw. Eng.: Softw. Eng. Practice*, 2019, pp. 183–192.
- [145] "AndroTest," 2019. [Online]. Available: <http://bear.cc.gatech.edu/shauvik/androtest/>
- [146] "DroidBugs," 2020. [Online]. Available: <https://github.com/I4Soft/DroidBugs>
- [147] S. Amann, S. Nadi, H. A. Nguyen, T. N. Nguyen, and M. Mezini, "MUBench: A benchmark for API-misuse detectors," in *Proc. IEEE/ACM 13th Working Conf. Mining Softw. Repositories*, 2016, pp. 464–467.
- [148] O. Riganelli, D. Micucci, and L. Mariani, "From source code to test cases: A comprehensive benchmark for resource leak detection in android apps," *Softw.: Practice Experience*, vol. 49, no. 3, pp. 540–548, 2019.
- [149] S. Mostafa, R. Rodriguez, and X. Wang, "Experience paper: A study on behavioral backward incompatibilities of java software libraries," in *Proc. 26th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2017, pp. 215–225.



Ting Su received the BS degree in software engineering and PhD degree in computer science from the School of Software Engineering, East China Normal University, Shanghai, China, in 2011 and 2016, respectively. He is a postdoc scholar with the Department of Computer Science, ETH Zurich, Switzerland, and will join East China Normal University as a professor in Fall 2020. Previously, he was a visiting scholar of UC Davis, USA from 2014 to 2015. His research focuses on software testing and validation, and was recognized with three ACM SIGSOFT distinguished paper awards (ICSE 2018, ASE 2018, ASE 2019). He has published broadly in top-tier programming language and software engineering venues, including PLDI, ICSE, FSE, ASE, and CSUR. For more information, please visit <http://tingsu.github.io/>.



Lingling Fan received the BEng and PhD degrees in computer science from East China Normal University, Shanghai, China, in 2014 and 2019, respectively. She is a research fellow with the School of Computer Science and Engineering, Nanyang Technological University, Singapore, and will join Nankai University in Fall 2020. She had been a research assistant in the Cyber Security Lab of NTU (2017–2019). Her research focuses on program analysis and testing, software security analysis and big data driven analysis, and has published in top-tier venues of software engineering and security (including ICSE, ASE, ESEC/FSE, S&P, TDSC, etc.) She got an ACM SIGSOFT Distinguished Paper Award at ICSE 2018. For more information, please visit <https://lingling-fan.github.io/>.



Sen Chen received the PhD degree in computer science from the School of Computer Science and Software Engineering, East China Normal University, Shanghai, China, in Jun. 2019. Currently, he is a research assistant professor with the School of Computer Science and Engineering, Nanyang Technological University, Singapore, and will join College of Intelligence and Computing, Tianjin University as a tenured associate professor. Previously, he was a research assistant of NTU from 2016 to 2019 and a research fellow from 2019–

2020. His research focuses on security and software engineering. He has published broadly in top-tier security and software engineering venues, including ICSE, ESEC/FSE, ASE, TSE, S&P, TDSC, etc. For more information, please visit <https://sen-chen.github.io/>.



Liu Yang received the bachelors and PhD degrees in computer science from the National University of Singapore (NUS), in 2005 and 2010, and continued with his postdoctoral research in NUS. He is now an associate professor with Nanyang Technological University. His current research focuses on software engineering, formal methods and security, and particularly specializes in software verification using model checking techniques, which led to the development of a state-of-the-art model checker, Process Analysis Toolkit. For more information, please visit <http://www.ntu.edu.sg/home/yangliu/>.



Lihua Xu received the master's and PhD degrees in computer science from the University of California, Irvine. She is associate professor of Practice in computer science at NYU Shanghai. Her research interests include software engineering and mobile security, with a focus on improving software quality via program analysis. She has published in top-tier venues such as ICSE, FSE, ASE, CCS, and MobiCom. Her recent work in software analysis received the 2018 ACM SIGSOFT Distinguished Paper Award. She is a recipient of the "Best New Investigator" Award at the 2006 Grace Hopper Women in Computing conference.



Geguang Pu received the BS degree in mathematics from Wuhan University, in 2000, and the PhD degree in mathematics from Peking University, in 2005. He is a professor with the School of Software Engineering, East China Normal University. His research interests include program testing, analysis and verification. He served as PC members for international conferences such as SEFM, ATVA, TASE etc. He was a co-chair of ATVA 2015. He has published more than 70 publications on the topics of software engineering and system verification (including ICSE, FSE, IJCAI, FM, ECAI, CONCUR etc.).



Zhendong Su received the BS degree in computer science and BA degree in mathematics from the University of Texas at Austin, Austin, TX, and the MS and PhD degrees in computer science from the University of California at Berkeley, Berkeley, CA. He is a professor in computer science at ETH Zurich, where he specializes in programming languages, software engineering, and computer security. For more information, please visit <https://people.inf.ethz.ch/suz/>.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

Inputs From Hell:

Learning Input Distributions for Grammar-Based Test Generation

Ezekiel Soremekun^{ID}, Esteban Pavese, Nikolas Havrikov, Lars Grunske^{ID}, and Andreas Zeller

Abstract—Grammars can serve as *producers* for structured test inputs that are syntactically correct by construction. A probabilistic grammar assigns probabilities to individual productions, thus controlling the distribution of input elements. Using the grammars as input parsers, we show how to *learn input distributions from input samples*, allowing to create inputs that are *similar* to the sample; by *inverting* the probabilities, we can create inputs that are *dissimilar* to the sample. This allows for three *test generation strategies*: 1) “Common inputs”—by learning from common inputs, we can create inputs that are *similar* to the sample; this is useful for regression testing. 2) “Uncommon inputs”—learning from common inputs and inverting probabilities yields inputs that are *strongly dissimilar* to the sample; this is useful for completing a test suite with “inputs from hell” that test uncommon features, yet are syntactically valid. 3) “Failure-inducing inputs”—learning from inputs that caused failures in the past gives us inputs that share similar features and thus also have a *high chance of triggering bugs*; this is useful for testing the completeness of fixes. Our evaluation on three common input formats (JSON, JavaScript, CSS) shows the effectiveness of these approaches. Results show that “common inputs” reproduced 96 percent of the methods induced by the samples. In contrast, for almost all subjects (95 percent), the “uncommon inputs” covered significantly different methods from the samples. Learning from failure-inducing samples reproduced all exceptions (100 percent) triggered by the failure-inducing samples and discovered new exceptions not found in any of the samples learned from.

Index Terms—Test case generation, probabilistic grammars, input samples

1 INTRODUCTION

DURING the process of software testing, software engineers typically attempt to satisfy three goals:

- 1) First, the software should work well on *common* inputs, such that the software delivers its promise on the vast majority of cases that will be seen in typical operation. To cover such behavior, one typically has a set of dedicated tests (manually written or generated).
- 2) Second, the software should work well on *uncommon* inputs. The rationale for this is that such inputs would exercise code that is less frequently used in production, possibly less tested, and possibly less understood [1].
- 3) Third, the software should work well on inputs that *previously caused failures*, such that it is clear that previous bugs have been fixed. Again, these would be covered via specific tests.

- Ezekiel Soremekun is with the Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, L-1855, Luxembourg. This work was conducted while working at CISPA Helmholtz Center for Information Security, 66123 Saarbrücken, Germany. E-mail: ezekiel.soremekun@uni.lu.
- Esteban Pavese and Lars Grunske are with the Department of Computer Science, Humboldt-Universität zu Berlin, 10117 Berlin, Germany. E-mail: {pavesees, grunske}@informatik.hu-berlin.de.
- Nikolas Havrikov and Andreas Zeller are with CISPA Helmholtz Center for Information Security, 66123 Saarbrücken, Germany. E-mail: {nikolas.havrikov, zeller}@cipsa.saarland.

Manuscript received 2 Oct. 2019; revised 19 July 2020; accepted 29 July 2020. Date of publication 3 Aug. 2020; date of current version 18 Apr. 2022. (Corresponding author: Ezekiel Soremekun.) Recommended for acceptance by A. Mesbah. Digital Object Identifier no. 10.1109/TSE.2020.3013716

How can engineers obtain such inputs? In this paper, we introduce a novel test generation method that *learns from a set of sample inputs* to produce additional inputs that are markedly *similar* or *dissimilar* to the sample. By learning from past failure-inducing inputs, we can create inputs with similar features; by learning from common inputs, we can create uncommon inputs with dissimilar features not seen in the sample.

The key ingredient to our approach is a *context-free grammar* that describes the input language to a program. Using such a grammar, we can *parse* existing input samples and *count* how frequently specific elements occur in these samples. Armed with these numbers, we can enrich the grammar to become a *probabilistic grammar*, in which production alternatives carry different likelihoods. Since these probabilities come from the samples used for the quantification, such a grammar captures properties of these samples, and producing from such a grammar should produce inputs that are *similar* to the sample. Furthermore, we can *invert* the learned probabilities in order to obtain a second probabilistic grammar, whose production would produce inputs that are *dissimilar* to the sample. We thus can produce three kinds of inputs, covering the three testing goals listed above:

- 1) “*Common inputs*”. By learning from *common* samples, we obtain a “*common*” probability distribution, which allows us to produce more “*common*” inputs. This is useful for regression testing.
- 2) “*Uncommon inputs*”. Learning from common samples, the resulting *inverted* grammar describes in turn the distribution of legal, but *uncommon* inputs. This is useful for completing test suites by testing uncommon features.

- 3) “Failure-inducing inputs”. By learning from samples that caused failures in the past, we can produce similar inputs that test the *surroundings* of the original inputs. This is useful for testing the completeness of fixes.

Both the “uncommon inputs” and “failure-inducing inputs” strategies have high chances of triggering failures. Since they combine features rarely seen or having caused issues in the past, we gave them the nickname “inputs from hell”. As an example, consider the following JavaScript input generated by focusing on uncommon features:

```
var { a: {} } = 'b' } = {};
```

This snippet is valid JavaScript code, but causes the Mozilla Rhino 1.7.7.2 JavaScript engine to crash during interpretation.¹ This input makes use of so-called *destructuring assignments*: In JavaScript, one can have several variables on the left hand side of an assignment or initialization. In such a case, each gets assigned a part of the structure on the right hand side, as in

```
var [one, two, three] = [1, 2, 3];
```

where the variable *one* is assigned a value of 1, *two* a value of 2, and so on. Such destructuring assignments, although useful in some contexts, are rarely found in JavaScript programs and tests. It is thus precisely the aim of our approach to generate such uncommon “inputs from hell”.

This article makes the following contributions:

- 1) We use context-free grammars to determine production probabilities from a given set of input samples.
- 2) We use mined probabilities to produce inputs that are *similar to a set of given samples*. This is useful for thoroughly testing commonly used features (regression testing), or to test the surroundings of previously failure-inducing inputs. Our approach thus leverages probabilistic grammars for both mining and test case generation. In our evaluation using the JSON, CSS and JavaScript formats, we show that our approach repeatedly covers the same code as the original sample inputs; learning from failure-inducing samples, we produce the same exceptions as the samples as well as new exceptions.
- 3) We use mined probabilities to produce inputs that are *markedly dissimilar* to a set of given samples, yet still valid according to the grammar. This is useful for robustness testing, as well as for exploring program behavior not triggered by the sample inputs. We are not aware of any other technique that achieves this objective. In our evaluation using the same subjects, we show that our approach is successful in repeatedly covering code not covered in the original samples.

The remainder of this paper is organized as follows. After giving a motivational example in Section 2, we detail our approach in Section 3. Section 4 evaluates our three strategies (“common inputs”, “uncommon inputs”, and “failure-inducing inputs”) on various subjects. After discussing the limitations (Section 5) and related work (Section 6), Section 7 concludes and presents future work.

1. We have reported this snippet as Rhino issue #385 and it has been fixed by the developers.

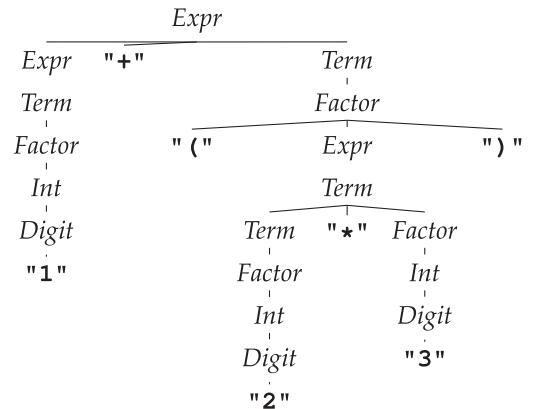


Fig. 1. Derivation tree representing $1 + (2 * 3)$.

2 INPUTS FROM HELL IN A NUTSHELL

To demonstrate how we produce both common and uncommon inputs, let us illustrate our approach using a simple example grammar. Let us assume we have a program P that processes *arithmetic expressions*; its inputs follow the standard syntax given by the grammar G below.

```

Expr → Term | Expr "+" Term | Expr "-" Term ;
Term → Factor | Term "*" Factor
      | Term "/" Factor ;
Factor → Int | " +" Factor
      | " - " Factor | "(" Expr ")";
Int → Digit Int | Digit ;
Digit → "0" | "1" | "2" | "3" | ... | "9" ;
  
```

Let us further assume we have discovered a bug in P : The input $I = 1 * (2 + 3)$ is not evaluated properly. We have fixed the bug in P , but want to ensure that similar inputs would also be handled in a proper manner.

To obtain inputs that are *similar* to I , we first use the grammar G to parse I and determine the *distribution* of the individual choices in productions. This makes G a *probabilistic grammar* G_p in which the productions’ choices are tagged with their probabilities. For the input I above, for instance, we obtain the probabilistic rule

```

Digit → 0% "0" | 33.3% "1" | 33.3% "2"
      | 33.3% "3" | 0% "4" | 0% "5"
      | 0% "6" | 0% "7" | 0% "8" | 0% "9" ;
  
```

which indicates the distribution of digits in I . Using this rule for production, we would obtain ones, twos, and threes at equal probabilities, but none of the other digits. Fig. 2 shows the grammar G_p as extension of G with all probabilities as extracted from the derivation tree of I (Fig. 1). In this derivation tree we see, for instance, that the nonterminal `Factor` occurs 4 times in total. 75 percent of the time it produces integers (`Int`), while in the remaining 25 percent, it produces a parenthesis expression `(" Expr ")`. Expressions using unary operators like `"+"` `Factor` and `" - "` `Factor` do not occur.

If we use G_p from Fig. 2 as a probabilistic production grammar, we obtain inputs according to these probabilities. As listed in Fig. 3, these inputs uniquely consist of the digits and operators seen in our sample $1 * (2 + 3)$. All of these inputs are likely to cover the same code in P as the original

```

 $Expr \rightarrow 66.7\% Term \mid 33.3\% Expr "-" Term$ 
 $\mid 0\% Expr "-" Term;$ 
 $Term \rightarrow 75\% Factor \mid 25\% Term "*" Factor$ 
 $\mid 0\% Term "/" Factor;$ 
 $Factor \rightarrow 75\% Int \mid 0\% "+" Factor$ 
 $\mid 0\% "-" Factor \mid 25\% "(" Expr ")" ;$ 
 $Int \rightarrow 0\% Digit Int \mid 100\% Digit;$ 
 $Digit \rightarrow 0\% "0" \mid 33.3\% "1" \mid 33.3\% "2"$ 
 $\mid 33.3\% "3" \mid 0\% "4" \mid 0\% "5"$ 
 $\mid 0\% "6" \mid 0\% "7" \mid 0\% "8" \mid 0\% "9";$ 

```

Fig. 2. Probabilistic grammar G_p , expanding G .

sample input, yet with different input structures that trigger the same functionality in P in several new ways.

When would one want to replicate the features of sample inputs? In the “common inputs” strategy, one would create test cases that are similar to a set of common inputs; this is helpful for regression testing. In the more interesting “failure-inducing inputs” strategy, one would learn from a set of failure-inducing samples to replicate their features; this is useful for testing the surroundings of past bugs.

If one only has sample inputs that work just fine, one would typically be interested in inputs that are *different* from our samples—the “uncommon inputs” strategy. We can easily obtain such inputs by *inverting* the mined probabilities: if a rule previously had a weight of p , we now assign it a weight of $1/p$, normalized across all production alternatives. For our $Digit$ rule, this gives the digits not seen so far a weight of $1/0 = \infty$, which is still distributed equally across all seven alternatives, yielding individual probabilities of $1/7 = 14.3\%$. Proportionally, the weights for the digits already seen in I are infinitely small, yielding a probability of effectively zero. The “inverted” rule reads now:

```

 $Digit \rightarrow 14.3\% "0" \mid 0\% "1" \mid 0\% "2" \mid 0\% "3"$ 
 $\mid 14.3\% "4" \mid 14.3\% "5" \mid 14.3\% "6"$ 
 $\mid 14.3\% "7" \mid 14.3\% "8" \mid 14.3\% "9";$ 

```

Applying this inversion to rules with non-terminal symbols is equally straightforward. The resulting probabilistic grammar $G_{p^{-1}}$ is given in Fig. 4.

This inversion can lead to infinite derivations, for example, the production rule in $G_{p^{-1}}$ for generating $Expr$ is recursive 100 percent of the time, expanding only to $Expr "-" Term$, without chance of hitting the base case. As a result, we take special measures to avoid such infinite productions during input generation (see Section 3.3).

If we use $G_{p^{-1}}$ as a production grammar—and avoiding infinite production—we obtain inputs as shown in Fig. 5. These inputs now focus on operators like subtraction or division or unary operators not seen in our input samples. Likewise, the newly generated digits cover the complement of those digits previously seen. Yet, all inputs are syntactically valid according to the grammar.

In summary, with common inputs as produced by G_p , we can expect to have a good set of regression tests—or a set replicating the features of failure-inducing inputs when learning from failure-inducing samples. In contrast, uncommon inputs as produced by $G_{p^{-1}}$ would produce features rarely found in samples, and thus cover complementary functionality.

```

 $(2 * 3)$ 
 $2 + 2 + 1 * (1) + 2$ 
 $((3 * 3))$ 
 $3 * (((3 + 3 + 3) * (2 * 3 + 3))) * (3)$ 
 $3 * 1 * 3$ 
 $((3) + 2 + 2 * 1) * (1)$ 
 $1$ 
 $((2)) + 3$ 

```

Fig. 3. Inputs generated from G_p in Fig. 2.

3 APPROACH

In order to explain our approach in detail, we start with introducing basic notions of probabilistic grammars.

3.1 Probabilistic Grammars

The probabilistic grammars that we employ in this paper are based on the well-known context-free grammars (CFGs) [2].

Definition 1 (Context-free grammar). A context-free grammar is a 4-tuple (V, T, P, S_0) , where V is the set of non-terminal symbols, T the terminals, $P : V \rightarrow (V \cup T)^*$ the set of productions, and $S_0 \in V$ the start symbol.

In a *non-probabilistic grammar*, rules for a non-terminal symbol S provide n alternatives A_i for expansion

$$S \rightarrow A_1 | A_2 | \dots | A_n. \quad (1)$$

In a *probabilistic context-free grammar* (PCFG), each of the alternatives A_i in Equation (1) is augmented with a probability p_i , where $\sum_{i=1}^n p_i = 1$ holds

$$S \rightarrow p_1 A_1 | p_2 A_2 | \dots | p_n A_n. \quad (2)$$

If we are using these grammars for generation of a sentence of the language described by the grammar, each alternative A_i has a probability of p_i to be selected when expanding S .

By convention, if one or more p_i are not specified in a rule, we assume that their value is the complement probability, distributed equally over all alternatives with these unspecified probabilities. Consider the rule

$$Letter \rightarrow 40.0\% "a" | "b" | "c"$$

Here, the probabilities for “**b**” and “**c**” are not specified; we assume that the complement of “**a**”, namely 60 percent, is equally distributed over them, yielding effectively

$$Letter \rightarrow 40.0\% "a" | 30.0\% "b" | 30.0\% "c"$$

Formally, to assign a probability to an unspecified p_i , we use

$$p_i = \frac{1 - \sum\{p_j | p_j \text{ is specified for } A_j\}}{\text{number of alternatives } A_k \text{ with unspecified } p_k}. \quad (3)$$

Again, this causes the invariant $\sum_{i=1}^n p_i = 1$ to hold. If no p_i is specified for a rule with n alternatives, as in Equation (1), then Equation (3) makes each $p_i = 1/n$, as intended.

3.2 Learning Probabilities

Our aim is to turn a classical context-free grammar G into a probabilistic grammar G_p capturing the probabilities from a set of samples—that is, to determine the necessary p_i values

```

Expr → 0% Term | 0% Expr "+" Term
| 100% Expr "-" Term;
Term → 0% Factor | 0% Term "*" Factor
| 100% Term "/" Factor;
Factor → 0% Int | 50% "+" Factor
| 50% "-" Factor | 0% "(" Expr ")";
Int → 100% Digit Int | 0% Digit;
Digit → 14.3% "0" | 0% "1" | 0% "2" | 0% "3"
| 14.3% "4" | 14.3% "5" | 14.3% "6"
| 14.3% "7" | 14.3% "8" | 14.3% "9";

```

Fig. 4. Grammar $G_{p^{-1}}$ inverted from G_p in Fig. 2.

as defined in Equation (2) from these samples. This is achieved by *counting* how frequently individual alternatives occur during parsing in each production context, and then to determine appropriate probabilities.

In language theory, the result of parsing a sample input I using G is a *derivation tree* [3], representing the structure of a sentence according to G . As an example, consider Fig. 1, representing the input " $1 + (2 * 3)$ " according to the example arithmetic expression grammar in Section 2. In this derivation tree, we can now *count* how frequently a particular alternative A_i was chosen in the grammar G during parsing. In Fig. 1, the rule for *Expr* is invoked three times during parsing. This rule expands once (33.3 percent) into *Expr* "+" *Term* (at the root); and twice (66.7 percent) into *Term* in the subtrees. Likewise, the *Term* symbol expands once (25 percent) into *Term* "*" *Factor* and three times (75 percent) into *Factor*. Formally, given a set T of derivation trees from a grammar G applied on sample inputs, we determine the probabilities p_i for each alternative A_i of a symbol $S \rightarrow A_1|\dots|A_n$ as

$$p_i = \frac{\text{Expansions of } S \rightarrow A_i \text{ in } T}{\text{Expansions of } S \text{ in } T}. \quad (4)$$

If a symbol S does not occur in T , then Equation (4) makes $p_i = 0/0$ for all alternatives A_i ; in this case, we treat all p_i for S as *unspecified*, assigning them a value of $p_i = 1/n$ in line with Equation (3). In our example, Equation (4) yields the probabilistic grammar G_p in Fig. 2.

3.3 Inverting Probabilities

We turn our attention now to the converse approach; namely producing inputs that *deviate* from the sample inputs that were used to learn the probabilities described above. This "uncommon input" approach promises to be useful if we accept that our samples are not able to cover all the possible system behavior, and if we want to find bugs in behaviors that are either not exercised by our samples, or do so rarely.

The key idea is to *invert* the probability distributions as learned from the samples, such that the input generation focuses on the complement section of the language (w.r.t. the samples and those inputs generated by the probabilistic grammar). If some symbol occurs frequently in the parse trees corresponding to the samples, this approach should generate the symbol less frequently, and vice versa: if the symbol seldom occurs, then the approach should definitely generate it often.

```

+5 / -5 / 7 - +0 / 6 / 6 - 6 / 8 - 5 - 4
-4 / +7 / 5 - 4 / 7 / 4 - 6 / 0 - 5 - 0
+5 / ++4 / 4 - 8 / 8 - 4 / 8 / 7 - 8 - 9
-6 / 9 / 5 / 8 - +7 / -9 / 6 - 4 - 4 - 6
+8 / ++8 / 5 / 4 / 0 - 5 - 4 / 8 - 8 - 8
-9 / -5 / 9 / 4 - -9 / 0 / 5 - 8 / 4 - 6
++7 / 9 / 5 - +8 / +9 / 7 / 7 - 6 - 8 - 4
-+6 / -8 / 9 / 6 - 5 / 0 - 5 - 8 - 0 - 5

```

Fig. 5. Inputs generated from $G_{p^{-1}}$ from Fig. 4.

For a moment, let us ignore probabilities and focus on *weights* instead. That is, the absolute (rather than relative) number of occurrences of a symbol in the parse tree of a sample. We start by determining the occurrences of a symbol A during a production S found in a derivation tree T

$$w_{A,S} = \frac{\text{Occurrence count of } A \text{ in the}}{\text{expansions of symbol } S \text{ in } T}. \quad (5)$$

To obtain *inverted weights* $w'_{A,S}$, a simple way is to make each $w'_{A,S}$ based on the reciprocal value of $w_{A,S}$, that is

$$w'_{A,S} = w_{A,S}^{-1} = \frac{1}{w_{A,S}}. \quad (6)$$

If the set of samples is small enough, or focuses only on a section of the language of the grammar, it might be the case that some production or symbol never appears in the parsing trees. If this is the case, then the previous equations end up yielding $w_{A,S} = 0$. We can compute $w_{A,S}^{-1} = \infty$, assigning the elements not seen an infinite weight. Consequently, all symbols B that were indeed seen before (with $w_{B,S} > 0$) are assigned an infinitesimally small weight, leading to $w'_{B,S} = 0$. The remaining infinite weight is then distributed over all of the originally "unseen" elements with original weight $w_{A,S} = 0$. Recall the arithmetic expression grammar in Section 2; such a situation arises when we consider the rule for the symbol *Digit*: the inverted probabilities for the rule focus exclusively on the complement of the digits seen in the sample.

All that remains in order to obtain actual probabilities is to *normalize* the weights back into a probability measure, ensuring for each rule that its invariant $\sum_{i=1}^n p'_i = 1$ holds

$$p'_i = \frac{w'_i}{\sum_{i=1}^n w'_i}. \quad (7)$$

3.4 Producing Inputs From a Grammar

Given a probabilistic grammar G_p for some language (irrespective of whether it was obtained by learning from samples, by inverting, or simply written that way in the first place), our next step in the approach is to generate inputs following the specified productions. This generation process is actually very simple, since it reduces to produce instances by traversing the grammar, as if it were a Markov chain. However, this generation runs the serious risk of probabilistically choosing productions that lead to an excessively large parsing tree. Even worse, the risk of generating an *unbounded* tree is very real, as can be seen in the rule for the symbol *Int* in the arithmetic expression grammar in Section 2. The production

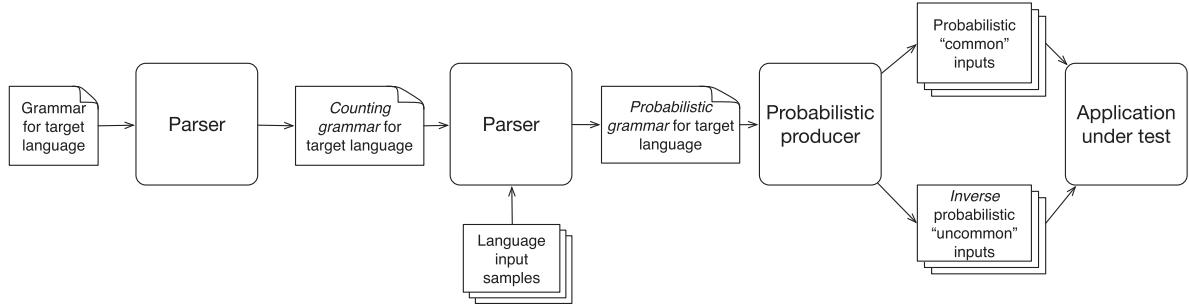


Fig. 6. Workflow for the generation of “common inputs” and “uncommon inputs”.

rule for said symbol triggers, with probability 1.0, a recursion with no base case, and will never terminate.

Our inspiration for constraining the growth of the tree during input generation comes from the PTC2 algorithm [4]. The main idea of this algorithm is to allow the expansion of not-yet-expanded productions, while ensuring that the number of productions does not exceed a certain threshold of performed expansions. This threshold would be set as parameter of the input generation process. Once this threshold is exceeded, the partially generated instance cannot be truncated, as that would result in an illegal input. Alternatively, we choose to allow further expansion of the necessary non-terminal symbols. However, from this point on, expansions are not chosen probabilistically. Rather, the choice is constrained to those expansions that generate the *shortest* possible expansion tree. This ensures both termination of the generation procedure, as well as trying to keep the input size close to the threshold parameter. This choice, however, does introduce a bias that may constitute a threat to the validity of our experiments that we discuss in Section 4.3.

3.5 Implementation

As a prerequisite for carrying out our approach, we only assume we have the context-free grammar of the language available for which we are interested in generating inputs, and a collection (no matter the size) of inputs that we will assume are *common* inputs. Armed with these elements, we perform the workflow detailed in Fig. 6.

The first step of the approach is to obtain a *counting grammar* from the original grammar. This counting grammar is, from the parsing point of view, completely equivalent to the original grammar. However, it is augmented with *actions* during parsing which perform all necessary counting of symbol occurrences parallel to the parsing phase. Finally, it outputs the probabilistic grammar. Note that this first phase requires not only the grammar of the target language, but also the grammar of the *language in which the grammar itself is written*. That is, generating the probabilistic grammar not only requires parsing sample inputs, but also the grammar itself. In the particular case of our implementation, we make use of the well-known parser generator ANTLR [5].

Once the probabilistic grammar is obtained, we derive the probabilistically-inverted grammar as described in this section. Armed with both probabilistically annotated grammars, we can continue with the input generation procedure.

4 EXPERIMENTAL EVALUATION

In this section we evaluate our approach by applying the technique in several case studies. In particular, we ask the following research questions:

- **RQ1 (“Common inputs”).** Can a learned grammar be used to generate inputs that resemble those that were employed during the grammar training?
- **RQ2 (“Uncommon inputs”).** Can a learned grammar be modified so it can generate inputs that, opposed to RQ1, are *in contrast* to those employed during the grammar training?
- **RQ3 (“Sensitivity to training set variance”).** Is our approach sensitive to variance in the initial samples?
- **RQ4 (“Sensitivity to size of training set”).** Is our approach sensitive to the size of the initial samples?
- **RQ5 (“Bugs found”).** What kind of crashes (exceptions) do we trigger in RQ1 and RQ2?
- **RQ6 (“Failure-inducing inputs”).** Can a learned grammar be used to generate inputs that reproduce failure-inducing behavior?

To answer RQ1 and RQ2, we need to compare inputs in order to decide whether these inputs are “similar” or “contrasting”. In the scope of this evaluation, we will use the *method coverage* and *call sequences* as measures of input similarity. We will define these measures later in this section, and we will discuss their usefulness. We address RQ3 by comparing the method calls and call sequences induced for three randomly selected training sets, each containing five inputs. Likewise, we evaluate RQ4 by comparing the method calls and call sequences induced for four randomly selected training sets, each containing N sample inputs, where $N \in \{1, 5, 10, 50\}$. We assess RQ5 by categorizing, inspecting and reporting all exceptions triggered by our test suites in RQ1 and RQ2. Finally, we address RQ6 by investigating if the “(un)common inputs” strategy can reproduce (or avoid) a failure and explore the surroundings of the buggy behavior.

4.1 Evaluation Setup

4.1.1 Generated Inputs

Once a probabilistic grammar is learned from the training instances, we generate several inputs that are fed to each subject. Our evaluation involves the generation of three types of test suites:

- a) *Probabilistic* - choice between productions is governed by the distribution specified by the learned probabilities in the grammar.

TABLE 1
Depth and Size of Derivation Trees for “Common Inputs”
(PROB) and “Uncommon Inputs” (INV)

Grammar	Mode	Depth of derivation tree				Nodes avg.
		min	max	avg.	median	
JSON	PROB	14	2867	96	63	3058
	INV	5	37	23	37	68
JavaScript	PROB	1	79	19	8	400
	INV	1	38	19	1	11,061
CSS	PROB	3	44	41	44	19,380
	INV	9	30	29	30	11,269

- b) *Inverse* - choice is governed by the distribution obtained by the inversion process described in Section 3.3.
- c) *Random* - choice between productions is governed by a uniform distribution (see RQ6).

Expansion size control is carried out in order to avoid unbounded expansion as described in Section 3.4. Table 1 reports the details of the produced inputs, i.e., the depth and average number of nodes in the derivation trees for the “common inputs” (i.e., probabilistic/PROB) and “uncommon inputs” (i.e., inverse/INV).

4.1.2 Research Protocol

In our evaluation, we generate test suites and measure the frequency of method calls, the frequency of call sequences and the number of failures induced in our subject programs. For each input language, the experimental protocol proceeds as follows:

- a) We randomly selected five files from a pool of thousands of sample files crawled from GitHub code repositories, and through our approach produced a probabilistic grammar out of them.² Since one of the main use cases of our tool is to complete a test suite, we perform grammar training with few (i.e., five) initial sample tests.
- b) We feed the sampled input files into the subject program and record the triggered failures, the induced call sequences and the frequency of method calls using the HPROF [6] profiler for Java.
- c) Using the probabilistic grammar, we generate test suites, each one containing 100 input files. We generate a total of 1,000 test suites, in order to control for variance in the input files. Overall, each experiment contains 100,000 input files (100 files × 1,000 runs). We perform this step for both probabilistic and inverse generations. Hence, the total number of inputs generated for each grammar is 200,000 (1,000 suites of 100 inputs each, a set of suites for each experiment).
- d) We test each subject program by feeding the input files into the subject program and recording the induced failures, the induced call sequences and the frequency of method calls using HPROF.

All experiments were conducted on a server with 64 cores and 126 GB of RAM; more specifically an Intel Xeon

2. To evaluate RQ6, we learned a PCFG from at most five random failure-inducing inputs.

TABLE 2
Subject Details

Input Format	Subject	Version	#Methods	LOC
JSON	Argo	5.4	523	8,265
	Genson	1.4	1,182	18,780
	Gson	2.8.5	793	25,172
	JSONJava	20180130	202	3,742
	Jackson	2.9.0	5,378	117,108
	JsonToJava	1880978	294	5,131
	MinimalJson	0.9.5	224	6,350
	Pojo	0.5.1	445	18,492
	json-simple	a8b94b7	63	2,432
	cliftonlabs	3.0.2	183	2,668
	fastjson	1.2.51	2,294	166,761
JavaScript	json2flat	1.0.3	37	659
	json-flattener	0.6.0	138	1,522
CSS3	Rhino	1.7.7	4873	100,234
	rhino-sandbox	0.0.10	49	529
CSS3	CSSValidator	1.0.4	7774	120,838
	flute	1.3	368	8,250
	jstyleparser	3.2	2,589	26,287
	csparser	0.9.27	2,014	18,465
CSS3	closure-style	0.9.27	3,029	35,401

CPU E5-2683 v4 @ 2.10 GHz with 64 virtual cores (Intel Hyperthreading), running Debian 9.5 Linux.

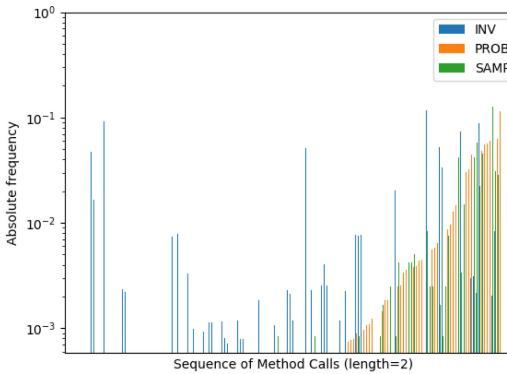
4.1.3 Subject Programs

We evaluated our approach by generating inputs and feeding them to a variety of Java applications. All these applications are open source programs using three different input formats, namely JSON, JavaScript and CSS3. Table 2 summarizes the subjects to be analyzed, their input format and the number of methods in each implementation.

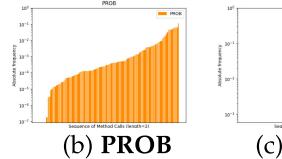
The initial, unquantified grammars for the input subjects were adapted from those in the repository of the well-known parser generator ANTLR [5]. We handle grammar ambiguity that may affect learning probabilities by ensuring every input has only one parse tree. Specifically, we adapt the input grammars by (re-)writing lexer modes for the grammars, shortening lexer tokens and re-writing parser rules. Training samples were obtained by scraping GitHub repositories for the required format files. The probabilistic grammars developed from the original ones, as well as the obtained training samples can be found in our replication package.

4.1.4 Measuring (Dis)similarity

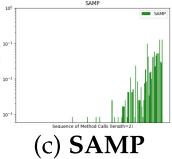
Questions RQ1 and RQ2 refer to a notion of similarity between inputs. Although white-box approaches exist that aim to measure test-case similarity and dissimilarity [7], [8], applying them to complex grammar-based inputs is not straightforward. However, in this paper, since we are dealing with evaluating the behavior of a certain piece of software, it makes sense to aim for a notion of *semantic* similarity. In this sense, two inputs are semantically similar if they incite similar behaviors in the software that processes them. In order to achieve this, we define two measures of input similarity based on structural and non-structural program coverage metrics. The *non-structural measure* of input similarity is the *frequency of method calls* induced in the programs. The *structural measure* is the *frequency of call sequences* induced in the program, a similar measure was used in [9]. Thus, we will say two inputs are similar if they induce



(a) PROB vs. SAMP vs. INV



(b) PROB

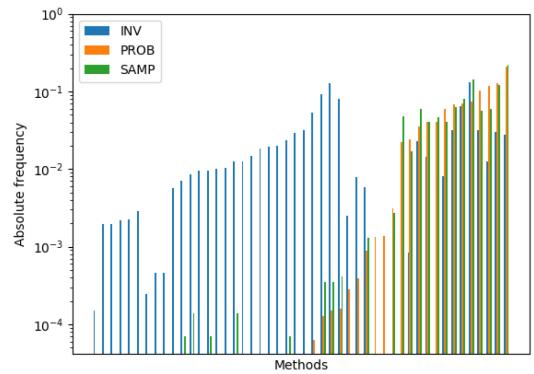


(c) SAMP

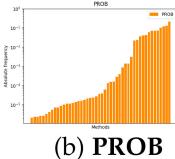


(d) INV

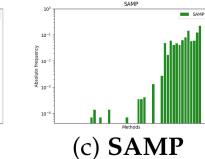
Fig. 7. Frequency analysis of call sequences for json-flattener (length=2).



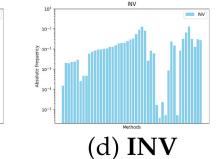
(a) PROB vs. SAMP vs. INV



(b) PROB



(c) SAMP



(d) INV

Fig. 9. Call frequency analysis for json-simple-cliftonlabs.

program, using grouped and single bar charts. These comparisons are in line with the visual tests described in [10].

For instance, Fig. 7 shows the frequency analysis of the call sequences induced in json-flattener by our test suites. The grouped bar chart compares the frequency distribution of call sequences for all three test suites, (i.e., (a) PROB versus SAMP versus INV) and the single bar chart shows the frequency distribution of call sequences for each test suite (i.e., (b.) PROB, (c) SAMP and (d) INV). Frequency analysis (in (a.)) shows that the (distribution of) call sequences of PROB and SAMP align (see rightmost part of bar chart), and INV often induces a different distribution of call sequences from the initial samples (see leftmost part of bar chart). The single bar chart for a test suite shows the frequency distribution of the call sequences triggered by the test suite. For instance, Figs. 7b and 7c show the call sequence distribution triggered by the “common inputs” and initial samples respectively. The comparison of both charts shows that all call sequences covered by the samples, were also frequently covered by the “common inputs”.

Likewise, Figs. 9, 10, and 11 show the call frequency analysis of the test suites using a grouped bar chart for comparison (i.e., (a) PROB versus SAMP versus INV) and a single bar chart to show the call frequency distribution of each test suite (i.e., (b.) PROB, (c) SAMP and (d) INV). The grouped bar chart shows the call frequency for each test suite grouped together by method, with bars for each test suite appearing side by side per method. For instance, analysing Fig. 9a shows that the call frequencies of PROB and SAMP align (see rightmost part of bar chart), and INV often induces a different call frequency for most methods (see leftmost part of bar chart). Moreover, the single bar chart for a test suite shows the call frequency distribution of the test suite. For instance, Figs. 9b and 9c show the call frequency distribution of the “common inputs” and initial samples respectively, their comparison shows that all methods covered by the samples, were also frequently covered by the “common inputs”.

Fig. 8. Call sequences covered by Sample for “common inputs” (PROB) and “uncommon inputs” (INV).

similar (distribution of) method call frequencies for the same program. The *frequency of call sequences* refers to the number of times a specific method call sequence is triggered by an input, for a program. For this measure, we say two inputs are similar if they trigger a similar distribution in the frequency with which the method sequences are called, for the same program. These notions allow for a great variance drift if we were to compare only two inputs. Therefore, we perform these comparisons on test suites as a whole to dampen the effect of this variance.

Using these measures, we aim at answering RQ1 and RQ2. RQ1 will be answered satisfactorily if the (distribution of) call frequencies and sequences induced by the “common inputs” strategy is *similar* to the call frequency and sequences obtained when running the software on the *training samples*. Likewise, RQ2 will be answered positively if the (distribution of) call frequencies and sequences for suites generated with the “uncommon inputs” strategy are markedly *dissimilar*.

4.1.5 Visual Test

For each test suite, we compare the frequency distribution of the call sequences and method calls triggered in a

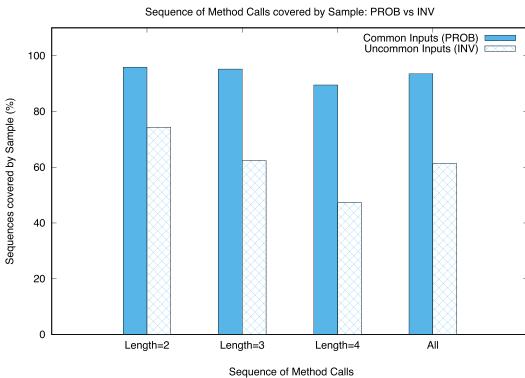
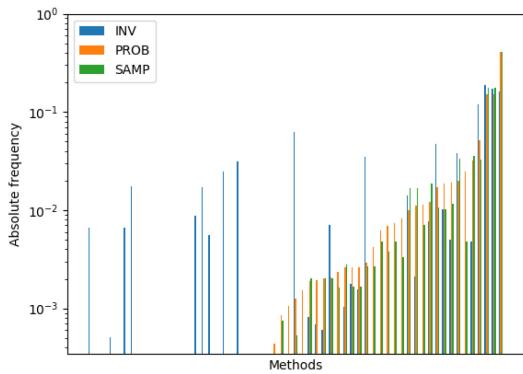
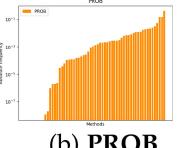


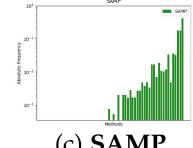
Fig. 8. Call sequences covered by Sample for “common inputs” (PROB) and “uncommon inputs” (INV).



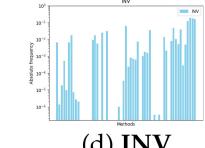
(a) PROB vs. SAMP vs. INV



(b) PROB



(c) SAMP



(d) INV

Fig. 10. Call frequency analysis for JSONJava.

4.1.6 Collecting Failure-Inducing Inputs

For each input file in our Github corpus, we fed it to every subject program of the input language and observe if the subject program crashes, i.e., the output status code is non-zero after execution. Then, we collect such inputs as failure-inducing inputs for the subject program and parse the standard output for the raised exception. In total, we fed 10,853 files to the subject programs, 1,000 each for CSS and JavaScript, and 8,853 for JSON. Exceptions were triggered for two input languages, namely JavaScript and JSON, no exception was triggered for CSS. In total, we collected 15 exceptions in seven subject programs (see Table 10).

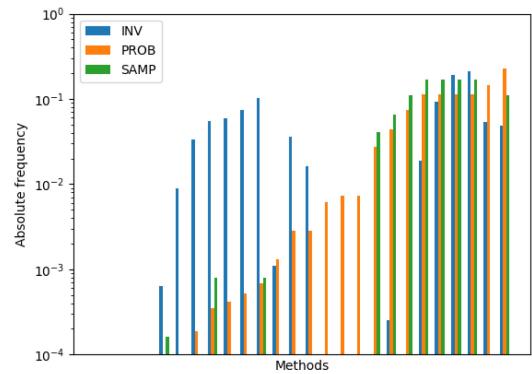
4.2 Experimental Results

In Figs. 9, 10, and 11, we show a representative selection of our results.³ For each subject, we constructed a chart that represents the absolute call frequency of each method in the subject. The horizontal axis (which is otherwise unlabelled) represents the set of methods in the subject, ordered by the frequency of calls in the experiment on probabilistic inputs.

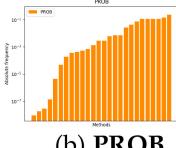
RQ1 (“Common inputs”): *Can a learned grammar be used to generate inputs that resemble those that were employed during the grammar training?*

To answer **RQ1**, we compare the methods covered by the sample inputs and the “common inputs” strategy (Table 4 and Figs. 9, 10, 11). We also examine the call sequences covered by the sample inputs and the “common inputs”, for consecutive call sequences of length two, three and four (Table 3 and Fig. 8). In particular, we investigate if the “common inputs” strategy covered at least the same methods or the same call sequences as the initial sample inputs.

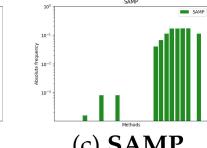
3. The full range of charts is omitted for space reasons. However, all charts, as well as the raw data, are available as part of the artifact package. Moreover, the charts shown here have been selected so that they are representative of the whole set; that is, the omitted charts do not deviate significantly.



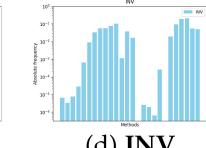
(a) PROB vs. SAMP vs. INV



(b) PROB



(c) SAMP



(d) INV

Fig. 11. Call frequency analysis for json-simple.

TABLE 3
Call Sequence Analysis for “Common Inputs” (PROB) and
“Uncommon Inputs” (INV) for all Subject Programs

Length	#	Call Sequences covered by Sample		Call sequences covered by PROB INV	
		also by PROB	also by INV	PROB	INV
2	1210	1157 (96%)	937 (74%)	6348	5196
3	1152	1099 (95%)	782 (62%)	7946	5930
4	849	803 (90%)	479 (47%)	9236	5825
Total	3211	3059 (94%)	2198 (61%)	23 530	16 951

Do the “common inputs” trigger similar non-structural program behavior (i.e., method calls) as the initial samples? For all subjects, the “common inputs” strategy covered almost all (96 percent) of the methods covered by the sample (see Table 4). This result shows that the “common inputs” strategy learned the input properties in the samples and reproduced the same (non-structural) program behavior as the initial samples. Besides, this strategy also covered other methods that were not covered by the samples.

The “common inputs” strategy triggered almost all methods (96 percent) called by the initial sample inputs.

Do the “common inputs” also trigger similar structural program behavior (i.e., sequences of method calls)? In our evaluation, the “common inputs” strategy covered most of the call sequences that were covered by the initial samples. For instance, Fig. 7 shows that the call sequences covered by the samples were also frequently covered by the “common inputs”, for json-flattener. Overall, the “common inputs” strategy covered 94 percent of the method call sequences induced by the sample (see Table 3 and Fig. 8). For all call sequences, the “common inputs” strategy also covered 90 to 96 percent of the method call sequences covered by the samples. This result shows that the “common inputs” strategy triggers the same structural program behavior as the initial samples.

TABLE 4
Method Coverage for “Common Inputs” (PROB) and “Uncommon Inputs” (INV)

Subject	#	Methods covered by sample also by PROB	Methods covered by sample also by INV	Methods covered by PROB	Methods covered by INV	Kolmogorov-Smirnov (KS) test of sample vs. PROB D-statistic(p-value)	Kolmogorov-Smirnov (KS) test of sample vs. INV D-statistic(p-value)
Argo	52	52 (100%)	32 (62%)	256	165	0.28 (1.11E-9)	0.50 (1.84E-30)
Genson	12	12 (100%)	10 (83%)	218	188	0.25 (3.46E-7)	0.73 (5.88E-64)
Gson	24	24 (100%)	14 (58%)	287	239	0.52 (1.09E-40)	0.25 (1.94E-9)
JSONJava	29	29 (100%)	23 (79%)	51	42	0.08 (0.99)	0.63 (3.45E-11)
Jackson	2	2 (100%)	1 (50%)	957	732	N/A	N/A
JsonToJava	29	29 (100%)	9 (31%)	82	33	0.25 (8.48E-3)	0.24 (1.38E-2)
MinimalJson	24	24 (100%)	18 (75%)	110	100	0.34 (2.36E-6)	0.83 (5.39E-42)
Pojo	23	23 (100%)	7 (30%)	159	93	0.19 (1.81E-3)	0.29 (1.04E-7)
json-simple	11	11 (100%)	10 (91%)	26	24	0.35 (0.09)	0.46 (7.13E-3)
cliftonlabs	23	23 (100%)	23 (100%)	48	48	0.21 (0.25)	0.54 (8.29E-7)
fastjson	70	70 (100%)	62 (89%)	245	231	0.37 (3.07E-15)	0.41 (8.84E-19)
json2flat	6	6 (100%)	5 (83%)	17	14	0.35 (0.24)	0.65 (1.15E-3)
json-flattener	36	36 (100%)	32 (89%)	83	81	0.15 (0.29)	0.15 (0.29)
Rhino	23	6 (26%)	23 (100%)	107	201	0.34 (3.18E-12)	0.45 (6.48E-21)
rhino-sandbox	3	3 (100%)	3 (100%)	17	17	0.47 (0.04)	0.53 (0.02)
CSSValidator	10	10 (100%)	10 (100%)	97	124	0.42 (1.20E-10)	0.38 (7.95E-9)
flute	58	57 (98%)	51 (88%)	148	131	0.29 (3.35E-6)	0.50 (7.34E-18)
jstyleparser	75	74 (99%)	59 (79%)	183	169	0.34 (1.53E-13)	0.38 (1.83E-17)
cssparser	71	71 (100%)	66 (93%)	177	152	0.36 (2.91E-12)	0.62 (4.21E-37)
closure-style	104	95 (91%)	103 (99%)	229	238	0.16 (2.03E-6)	0.09 (3.34E-2)
Total	685	657 (96%)	561 (82%)	3,497	3,022		

The “common inputs” strategy triggered most call sequences (94 percent) covered by the initial sample inputs.

Additionally, we compare the statistical distributions resulting from our strategies. We need to be able to see a pattern in frequency calls such that the frequency curves for the *sample* runs and the *probabilistic* runs match as described in the visual test (see Section 4.1.5). Figs. 9, 10, and 11 show that this match does hold for all subjects.

For all subjects, the method call frequency curves for the sample runs and the probabilistic runs match.

We also perform a statistical analysis on the distributions to increase the confidence in our conclusion. We performed a distribution fitness test (KS - Kolmogorov-Smirnov) on the sample versus the probabilistic call distribution; and on the sample versus the inverse probabilistic distribution. It must be noted that the KS test aims at determining whether the distributions are *exactly* the same, whereas we want to ascertain if they are *similar* or *dissimilar*. KS tests are *very sensitive* to small variations in data, which makes it, in principle, inadequate for this objective. In this work, we employ the approach used by Fan [11]—we first estimate the kernel density functions of the data distributions, which smoothen the estimated distribution. Then, we bootstrap and resample new data on the kernel density estimates, and perform the KS test on the bootstrapped data.

The KS test confirms the results from the visual inspection, the distribution of the method call frequency of “common inputs” matches the distribution in the sample (see Table 4), for some subjects. However, there are also subjects, where the hypothesis is rejected ($p < 0.05$) that method call frequency distributions (sample and “common inputs”) come from the same distribution, which is indicated by the blue entries. In the case of the Jackson subject, frequencies for the sample calls are all close to zero, which makes the data inadequate for the KS test.

RQ2 (“Uncommon inputs”): Can a learned grammar be modified such it can generate inputs that, opposed to RQ1, are in contrast to those employed during the grammar training?

For all subjects, the “uncommon inputs” produced by inverting probabilities covered markedly fewer (82 percent) of the methods covered by the sample (see Table 4). This result shows that the “uncommon inputs” strategy learned the input properties in the samples and produced inputs that avoid several methods covered by the samples.

The “uncommon inputs” strategy triggered markedly fewer methods (82 percent) called by the initial sample inputs.

Do the “uncommon inputs” trigger fewer of the call sequences covered by the initial samples? Table 3 shows that the “uncommon inputs” strategy triggered significantly fewer (61 percent) of the call sequences covered by the samples. The number of call sequences induced by the uncommon inputs decreases significantly as the length of the call sequence increases (see Fig. 8). For instance, comparing frequency charts of call sequences in Figs. 7a, 7c and 7d also show that “uncommon inputs” frequently avoided inducing the call sequences triggered by the initial samples. Notably, for sequences of four consecutive method calls, the “uncommon inputs” strategy covered only 47 percent of the sequences covered by the initial samples (see Table 3). Overall, the “uncommon inputs” avoided inducing the call sequences that were triggered by the initial samples.

The “uncommon inputs” strategy induced significantly fewer call sequences (61 percent) covered by the initial samples.

Do the “uncommon inputs” only cover *fewer*, or also *different* methods? We perform a visual test to examine if we see a *markedly different* call frequency between the samples and the inputs generated by the “uncommon inputs”

TABLE 5
Sensitivity to Training Set Variance Using Three Different Sets of Initial Samples Containing Five Inputs Each

Set#	Methods covered by sample			Methods covered by PROB		Call Sequences covered by sample			Call Sequence covered by PROB	
	#	also by PROB	also by INV	by PROB	by INV	#	also by PROB	also by INV	by PROB	by INV
1	685	657 (96%)	561 (82%)	3,497	3,022	3,211	3,059 (94%)	2,198 (61%)	23,530	16,951
2	2,963	2,924 (97%)	2,764 (85%)	8,623	8,246	6,044	5,643 (93%)	4,110 (68%)	22,531	19,896
3	2,656	2,639 (100%)	2,516 (87%)	8,655	8,165	5,005	4,915 (98%)	3,306 (66%)	20,792	19,892

TABLE 6
Sensitivity to the Size of the Training Set Using Initial Sample Size $N \in \{1, 5, 10, 50\}$

Size	Methods covered by sample			Methods covered by PROB		Call Sequences covered by sample			Call Sequence covered by PROB	
	#	also by PROB	also by INV	by PROB	by INV	#	also by PROB	also by INV	by PROB	by INV
1	1,496	1,490 (100%)	1,352 (79%)	8,715	7,954	1,955	1,942 (99%)	1,135 (58%)	21,279	15,341
5	685	657 (96%)	561 (82%)	3,497	3,022	3,211	3,059 (94%)	2,198 (61%)	23,530	16,951
10	3,546	3,517 (100%)	3,339 (89%)	9,388	11,497	6,297	6,105 (97%)	4,474 (71%)	26,575	21,214
50	5,347	5,313 (100%)	4,961 (89%)	8,950	8,217	9,389	9,076 (97%)	7,421 (79%)	23,512	18,391

strategy. In almost all charts this is the case (see Figs. 9, 10, and 11). The only exception is the CSSValidator subject.

For all subjects (except CSSValidator), the method call frequency curves for the sample runs and “uncommon inputs” runs are markedly different.

Besides, we examine if the frequency of distribution of method calls for the samples and the “uncommon inputs” are significantly dissimilar. In particular, the KS tests shows that for all subjects (except json-flattener) the distributions of method calls in the sample and the “uncommon inputs” are significantly different ($p < 0.05$, see *sample versus INV* in Table 4).

RQ3 (“Sensitivity to training set variance”): Is our approach sensitive to variance in the initial samples?

We examine the sensitivity of our approach to the variance in the training set. We randomly selected three distinct training sets, each containing five inputs. Then, for each set, we compare the methods and call sequences covered by the samples to those induced by the generated inputs (Table 5).

Our evaluation showed that *our approach is not sensitive to training set variance*. In particular, for all training sets, the “common inputs” strategy covered most of the methods and call sequences induced by the initial sample inputs. Table 5 shows that the “common inputs” (PROB) consistently covered almost all call sequences (93 to 98 percent) covered by the initial samples, while “uncommon inputs” (INV) covered significantly fewer call sequences (61 to 68 percent). Likewise, the “common inputs” consistently covered almost all methods (96 to 100 percent) covered by the initial samples, while “uncommon inputs” covered fewer methods (82 to 87 percent) (cf. Table 5).

Both strategies, the “common inputs” and the “uncommon inputs”, are insensitive to training set variance.

RQ4 (“Sensitivity to the size of training set”): Is our approach sensitive to the size of the initial samples?

We evaluate the sensitivity of our approach to the size of the training set. For each input format, we randomly

selected four distinct training sets containing N sample inputs, where $N \in \{1, 5, 10, 50\}$. Then, for each set, we compare the methods and call sequences induced by the samples to those induced by the generated inputs (Table 6).

Regardless of the size of the training set, the “common inputs” strategy consistently covered most of the methods and call sequences covered by the initial samples. Specifically, for all sizes, the “common inputs” covered almost all (94 to 99 percent) of the call sequences covered by the initial samples, while “uncommon inputs” covered significantly fewer call sequences (58 to 79 percent). In the same vein, the “common inputs” consistently covered almost all methods (96 to 100 percent) covered by the initial samples, while “uncommon inputs” covered fewer methods (79 to 89 percent) (cf. Table 6). These results demonstrates that the effectiveness of our approach is independent of the size of the training set.

The effectiveness of our approach is independent of the size of the training set used for grammar training.

RQ5 (“Bugs found”): What kind of crashes (exceptions) do we trigger?

To address RQ5, we examine all of the exceptions triggered by our test suites. We inspect the exceptions triggered during our evaluation of the “common inputs” strategy (in RQ1) and the “uncommon inputs” strategy (in RQ2). To evaluate if our approach is capable of finding real-world bugs, we compare the exceptions triggered in both RQ1 and RQ2 to the exceptions triggered by the input files crawled from *Github* (using the setup described in Section 4.1.6).

Both of our strategies triggered 40 percent of the exceptions triggered by the crawled files, i.e., six (out of 15) exceptions causing thousands of crashes in four subjects (cf. Tables 7 and 8). Half (three) of these exceptions had no samples of failure-inducing inputs in their grammar training. This indicates that, even without failure-inducing input samples during grammar training, our approach is able to trigger common buggy behaviors in the wild, i.e., bugs triggered by the crawled input samples. Exceptions were triggered for JSON and JavaScript input formats, however, no exception was triggered for CSS.

TABLE 7
Exceptions Induced by “Common Inputs” (PROB), and “Uncommon Inputs” (INV)

Input Format	Subject	Exception	#Failure-inducing samples	PROB	Occurrence rate in INV	Crawled Files
CSS		No exceptions triggered				
JSON	Gson	java.lang.NullPointerException	0	0	0.0001	0
	Pojo	java.lang.StringIndexOutOfBoundsException	0	0.0259	0	0.0024
	Pojo	java.lang.NumberFormatException	0	0	0.0001	0
	Argo	argo.saj.InvalidSyntax	0	0	0.0023	0.0225
	JSONToJava	org.json.JSON	0	0.0200	0.0200	0.0223
	json2flat	com.fasterxml.jackson.core.JsonParseException	0	0	0.0013	0
	json-flattener	com.eclipsesource.json.Parse	0	0	0.0013	0
	json-flattener	java.lang.UnsupportedOperationException	0	0.2981	0	0
	json-flattener	java.lang.IllegalArgumentException	0	0.2554	0	0
	json-flattener	java.lang.NullPointerException	0	0.0398	0	0
	json-flattener	java.lang.NumberFormatException	0	0.0028	0.0745	0
JavaScript	rhino-sandbox	org.mozilla.javascript.EvaluatorException	3	0.4905	0.4469	0.5290
	rhino-sandbox	java.lang.IllegalStateException	1	0.0016	0	0.0010
	rhino-sandbox	org.mozilla.javascript.EcmaError	1	0.0058	0.0500	0.3740

Probabilistic grammar-based testing induced two fifths of all exceptions triggered by the crawled files.

Our strategies were able to trigger eight other exceptions that could not be found by the crawled files (*cf. Fig. 12*). This result shows the utility of our approach in finding *rare buggy behaviors*, i.e., uncommon bugs in the crawled input samples. Besides, all of these exceptions were triggered despite a lack of “failure-inducing input” samples in the grammar training. In particular, both strategies triggered nine exceptions each, three and four of which were triggered only by the “common inputs” and only by the “uncommon inputs”, respectively.

Probabilistic grammar-based testing induced eight new exceptions that were not triggered by the crawled files.

The “common inputs” strategy triggered all of the exceptions triggered by the original sample inputs used in grammar training. Three exceptions were triggered by the sample inputs and all three exceptions were triggered by the “common inputs” strategy, while “uncommon inputs” triggered only two of these exceptions (*cf. Tables 7 and 8*). Again, this result confirms that our grammar training approach can learn the input properties that characterize specific program behaviors.

The “common inputs” induced all of the exceptions triggered by the original sample inputs.

Overall, 14 exceptions in seven subject programs were found in our experiments (*see Tables 7 and 8*). On inspection, six of these exceptions affecting five subject programs have

been reported to developers as severe bugs. These exceptions have been extensively discussed in the bug repository of each subject program. This result reveals that our approach can generate inputs that reveal real-world buggy behaviors. Additionally, from the evaluation of crawled files, 15 exceptions in five subjects were found. In particular, one exception (Rhino issue #385, which is also reproducible with our approach) has been confirmed and fixed by the developers.

RQ6 (“Failure-inducing inputs”): Can a learned grammar be used to generate inputs that reproduce failure-inducing behavior?

Let us now investigate if our approach can learn a PCFG from failure-inducing samples and reproduce the failure.

For each exception triggered by the crawled files (in Section 4.1.6), we learned a PCFG from at most five failure-inducing inputs that trigger the exception. Then, we run our PROB approach on the PCFG, using the protocol setting in Section 4.1.2. The goal is to demonstrate that the PCFG learns the input properties of the “failure-inducing inputs”, i.e. inputs generated via PROB should trigger *the same exception* as the failure-inducing samples. This is useful for exploring the surroundings of a bug.

In addition, for each exception, we run the inverse of “failure-inducing inputs” (i.e., INV), in order to evaluate if the “uncommon inputs” avoid reproducing the failures. In contrast, for each exception, we run the random generator (RAND) on the CFG (according to Section 4.1.2), in order to evaluate the

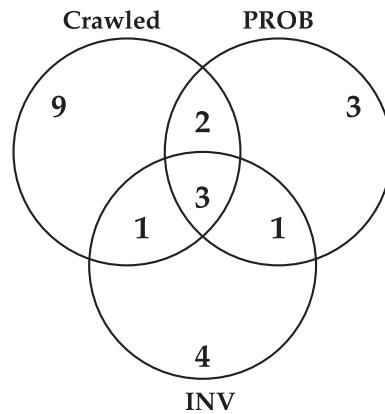


Fig. 12. Number of exceptions triggered by the test suites.

TABLE 8
Exception Details

Subject	#Exceptions	#Subjects	Average subject crash rate (All)	(Crashed)
SAMP	3	1	0.05263	1
PROB	9	4	0.05999	0.28493
INV	9	7	0.03139	0.08521

TABLE 9

Reproduced Exceptions by Sample Inputs (SAMP), “Failure-Inducing Inputs” (PROB), Inverse of “Failure-Inducing Inputs” (INV) and Random Grammar-Based Generation (RAND)

	#Exceptions Reproduced	Other	Average # Failure-inducing inputs
SAMP	15	0	87
PROB	15	21	18,429
INV	5	6	18,080
RAND	0	0	0

probability of randomly triggering (these) exceptions without a (learned) PCFG. In the random configuration (RAND), production choices have equal probability of generation.

In Table 9, we have summarized the number of reproduced exceptions. We see that probabilistic generation (PROB) reproduced all (15) failure-inducing inputs collected in our corpus. This shows that the grammar training approach effectively captured the distribution of input properties in the failure-inducing inputs. Moreso, it reproduced the program behavior using the learned PCFG.

Learning probabilities from failure-inducing inputs strategy reproduces 100 percent of the exceptions in our corpus.

For the inverse of “failure-inducing inputs”, our evaluation showed that the “uncommon inputs” strategy could avoid reproducing the learned failure-inducing behavior for most (10 out of 15) of the exceptions (*cf. Tables 9 and 10*).

The “uncommon inputs” strategy could avoid reproducing the learned program behavior for two-thirds of the exceptions.

However, this strategy reproduced a third (five out of 15) of the exceptions in our corpus (*cf. Tables 9 and 10*). On inspection, we found that “uncommon inputs” reproduced these five exceptions by generating new counter-examples, i.e., new inputs that are different from the initial samples but trigger the same exception. This is because the initial sample of failure-inducing inputs was not general enough to fully characterize all input properties triggering the crash. This result demonstrates that the inverse of “failure-inducing inputs” can explore the boundaries of the learned behavior in the PCFG, hence, it is useful for generating counter-examples.

The “uncommon inputs” strategy generated new counter-examples for one-third of the exceptions in our corpus.

In contrast, the random test suite (RAND) could not trigger *any* of the exceptions in our corpus, as shown in Table 9. This is expected, since a random traversal of the input grammar would need to explore numerous path combinations to find the specific paths that trigger an exception. This result demonstrates the effectiveness of the grammar training and the importance of the PCFG in capturing input properties.

Random input generation could not reproduce any of the exceptions in our corpus.

Furthermore, we examined the proportion of the generated inputs that trigger an exception. In total, for each test configuration and each exception we generated 100,000 inputs. We investigate the proportion of these inputs that trigger the exception.

Our results for this analysis are summarized in Table 10. We see that about 18 percent of the inputs generated by the “failure-inducing inputs” strategy (PROB) trigger the learned exception, on average. This rate is three times as high as the exception occurrence rate in our corpus (SAMP; 6 percent).

About one in five inputs generated by the “failure-inducing inputs” strategy reproduced the failure-inducing exception.

Finally, the “failure-inducing inputs” strategy also produced *new exceptions* not produced by the original sample of failure-inducing inputs. As shown in the “Other” column in Table 9, “failure-inducing inputs” triggered at least one new exception for each exception in our corpus. This result suggests that the PCFG is also useful for exploring the boundaries of the learned behavior, in order to trigger other program behaviors different from the learned program behavior. This is possible because “failure-inducing inputs” not only reproduces the exact features found in the samples, but also their variations and combinations.

The “failure-inducing inputs” strategy discovered new exceptions not triggered by the samples or random generation.

4.3 Threats to Validity

4.3.1 Internal Validity

The main threat to internal validity is the correctness of our implementation. Namely, whether our implementation does indeed learn a probabilistic grammar corresponding to the distribution of the real world samples used as training set. Unfortunately, this problem is not a simple one to resolve. The probabilistic grammar can be seen as a Markov chain, and the aforementioned problem is equivalent to verifying that its equilibrium distribution corresponds to the posterior distribution of the real world samples. The problem is two-fold: first, the number of samples necessary in order to ascertain the posterior distribution is inordinate. Second, even if we had a chance to process such a number of inputs, or if the posterior distribution were otherwise known, it might well be the case that the probabilistic grammar actually has no equilibrium distribution. However, our tests on smaller and simpler grammars suggest that this is not an issue.

A second internal validity threat is present in the technique we use for controlling the size of the generated

TABLE 10
**Reproducing Exceptions by “Failure-Inducing Inputs” (PROB), Inverse of “Failure-Inducing Inputs” (INV),
and Random Grammar-Based Test Generation (RAND)**

Input Format (#Crawled files)	Subject	Exception	#Failure-inducing inputs	Occurrence rate in crawled files	Reproduction rate in RAND		
					PROB	RAND	INV
JSON (8853)	Gson	java.lang.ClassCast	6	0.0007	0.0090	0	0
	Gson	java.lang.IllegalState	22	0.0025	1	0	1
	JSONToJava	java.lang.ArrayIndexOutOfBoundsException	38	0.0043	0.0025	0	0
	JSONToJava	java.lang.IllegalArgumentException	1	0.0001	0.0003	0	0
	JSONToJava	org.json.JSON_1	167	0.0189	0.1811	0	0.1811
	JSONToJava	org.json.JSON_2	30	0.0034	1	0	1
	Pojo	java.lang.IllegalArgumentException	88	0.0099	0.0002	0	0
	Pojo	java.lang.StringIndexOutOfBoundsException	21	0.0024	0.0471	0	0
AVERAGE				0.0646	0.1842	0	0.1808

samples. As described before, a sample’s size is defined in terms of the number of expansions in its parsing tree. In order to control the size, we keep track of the number of expansions generated. Once this number crosses a certain threshold (if it actually crosses it at all), all open derivations are closed via their shortest path. This does introduce a bias in the generation that does not exactly correspond to the distribution described by the probabilistic grammar. The effects of such a bias are difficult to determine, and merit further and deeper study. However, not performing this termination procedure would render useless any approach based on probabilistic grammars.

4.3.2 External Validity

Threats to external validity relate to the generalizability of the experimental results. In our case, this is specifically related to the subjects used in the experiments. We acknowledge that we have only experimented with a limited number of input grammars. However, we have selected the subjects with the intention to test our approach on practically relevant input grammars with different complexities, from small to medium size grammars like JSON; and rather complex grammars like JavaScript and CSS. As a result, we are confident that our approach will also work on inputs that can be characterized by context-free grammars with a wide range of complexity. However, we do have evidence that the approach does not seem to be generalizable to combinations of grammars and samples such that they induce the learning of an almost-uniform probabilistic grammar.

4.3.3 Construct Validity

The main threat to construct validity is the metric we use to evaluate the similarity between test suites, namely method call frequency. While the uses of coverage metrics as adequacy criteria is extensively discussed by the community [12], [13], [14], their binary nature (that is, we can either report *covered* or *not covered*) makes them too shallow to differentiate for behavior. The variance intrinsic to the probabilistic generation makes it very likely that at least one sample will cover parts of the code unrelated to those covered by

the rest of the suite. Besides, method call frequency is considered a non-structural coverage metric. To mitigate this threat, we also evaluate our test suites using a structural metric, in particular, (frequency of) call sequences.

5 LIMITATIONS

Context Sensitivity. Although, our probabilistic grammar learning approach captures the distribution of input properties, the learned input distribution is limited to production choices at the syntactic level. This approach does not handle context-sensitive dependencies such as the order, sequences or repetitions of specific input elements. However, our approach can be extended to learn contextual dependencies, e.g., by learning sequences of elements using N-grams [15] or hierarchies of elements using k-paths [16].

Input Constraints. Beyond lexical and syntactical validity, structured inputs often contain input semantics such as checksums, hashes, encryption, or references. Context-free grammars, as applied in this work, do not capture such complex input constraints. Automatically learning such input constraints for test generation is a challenging task [17]. In the future, we plan to automatically learn input constraints to drive test generation, e.g., using attribute grammars.

6 RELATED WORK

Generating Software Tests. The aim of *software test generation* is to find a sample of inputs that induce executions that sufficiently cover the possible behaviors of the program—including undesired behavior. Modern software test generation relies, as surveyed by Anand *et al.* [12] on *symbolic code analysis* to solve the path conditions leading to uncovered code [1], [18], [19], [20], [21], [22], [23], [24], *search-based approaches* to systematically evolve a population of inputs towards the desired goal [25], [26], [27], [28], random inputs to programs and functions [29], [30] or a combination of these techniques [31], [32], [33], [34], [35]. Additionally, machine learning techniques can also be applied to create test sequences [36], [37]. All these approaches have in common that they do not require an additional model or annotations to constrain the set of generated inputs; this makes them very versatile, but brings the risk of producing false

alarms—failing executions that cannot be obtained through legal inputs.

Grammar-Based Test Generation. The usage of grammars as *producers* was introduced in 1970 by Hanford in his *syntax machine* [38]. Such producers are mainly used for testing compilers and interpreters: CSmith [39] produces syntactically correct C programs, and LANGFUZZ [40] uses a JavaScript grammar to parse, recombine, and mutate existing inputs while maintaining most of the syntactic validity. GENA [41], [42] uses standard symbolic grammars to produce test cases and only applies stochastic annotation during the derivation process to distribute the test cases and to limit recursions and derivation depth. Grammar-based white-box fuzzing [43] combines grammar-based fuzzing with symbolic testing and is now available as a service from Microsoft. As these techniques operate with system inputs, any failure reported is a true failure—there are no false alarms. None of the above approaches use probabilistic grammars, though.

Probabilistic Grammars. The foundations of probabilistic grammars date back to the earliest works of Chomsky [44]. The concept has seen several interactions and generalizations with physics and statistics; we recommend the very nice article by Geman and Johnson [45] as an introduction. Probabilistic grammars are frequently used to analyze ambiguous data sequences—in computational linguistics [46] to analyze natural language, and in biochemistry [47] to model and parse macromolecules such as DNA, RNA, or protein sequences. Probabilistic grammars have been used also to model and produce input data for specific domains, such as 3D scenes [48] or processor instructions [49].

The usage of probabilistic grammars for test generation seems rather straightforward, but is still uncommon. The *Geno* test generator for .NET programs by Lämmel and Schulte [50] allowed users to specify probabilities for individual production rules. Swarm testing [51], [52] uses statistics and a variation of random testing to generate tests that deliberately targets or omits features of interest. These approaches, in contrast to the one we present in this paper, does not use existing samples to learn or estimate probabilities. The approach by Poupling *et al.* [53], [54] uses a stochastic context-free grammar for statistical testing. The goal of this work is to correctly imitate the operational profile and consequently the generated test cases are similar to what one would expect during normal operation of the system. The test case generation [55], [56] and failure reproduction [57] approaches by Kifetew *et al.* combine probabilistic grammars with a search-based testing approach. In particular, like our work, StGP [55] also learns stochastic grammars from sample inputs.

Our approach aims to generate inputs that induce (dis)similar program behaviors as the sample inputs. In contrast to our paper, StGP [55] is focused on evolving and mutating the learned grammars to *improve code coverage*. Although, StGP’s goal of generating realistic inputs is very similar to our “common inputs” strategy (see RQ1), our approach can further generate realistic inputs that are dissimilar to the sample inputs (see RQ2). Meanwhile, StGP is not capable of generating dissimilar inputs.

Mining Probabilities. Related to our work are approaches that mine grammar rules and probabilities from existing

samples. Patra and Pradel [58] use a given parser to mine probabilities for subsequent fuzz testing and to reduce tree-based inputs for debugging [59]. Their aim, however, is not to produce inputs that would be similar or dissimilar to existing inputs, but rather to produce inputs that have a higher likelihood to be syntactically correct. This aim is also shared by two *mining* approaches: GLADE [60] and Learn&Fuzz [61], which learn producers from large sets of input samples even without a given grammar.

All these approaches, however, share the problem of producing only “common inputs”—they can only focus on common features rather than uncommon features, creating a general “tension between conflicting learning and fuzzing goals” [61]. In contrast, our work can specifically focus on “uncommon inputs”—that is, the complement of what has been learned.

Like us, the Skyfire approach [62] aims at also leveraging “uncommon inputs” for probabilistic fuzzing. Their idea is to learn a probabilistic distribution from a set of samples and use this distribution to generate seeds for a standard fuzzing tool, namely AFL [63]. Here, favoring low probability rules is one of many heuristics applied besides low frequency, low complexity, or production limits. Although the tool has shown good results for XML-like languages, results for other, general grammar formats such as JavaScript are marked as “preliminary” only, though.

Mining Grammars. Our approach requires a grammar that can be used both for parsing and producing inputs. While engineering such a grammar may well pay off in terms of better testing, it is still a significant investment in the case of specific domain inputs where such a grammar might not be immediately available. Mining input structures [64], as exemplified using the above GLADE [60] and Learn&Fuzz [61] approaches, may assist in this task. AUTOGRAM [65] and MIMID [66] mine human-readable input grammars exploiting structure and identifiers of a program processing the input, which makes them particularly promising.

7 CONCLUSION AND FUTURE WORK

In this paper we have presented an approach that allows engineers, using a grammar and a set of input samples, to generate instances that are either similar or dissimilar to these samples. Similar samples are useful, for instance, when learning from failure-inducing inputs; while dissimilar samples could be used to leverage the testing approach to explore previously uncovered code. Our approach provides a simple, general, and cost-effective means to generate test cases that can then be targeted to the commonly used portions of the software, or to the rarely used features.

Despite their usefulness for test case generation, grammars—including probabilistic grammars—still have a lot of potential to explore in research, and a lot of ground to cover in practice. Our future work will focus on the following topics:

Deep Models. At this point, our approach captures probabilistic distributions only at the level of individual rules. However, probabilistic distributions could also capture the occurrence of elements in particular *contexts*, and differentiate between them. For instance, if a “+” symbol rarely occurs within parentheses, yet frequently outside of them, this difference would, depending on how the grammar is structured,

not be caught by our approach. The domain of computational linguistics [46] has introduced a number of models that take context into account. In our future work, we shall experiment with deeper context models, and determining their effect on capturing common and uncommon input features.

Grammar Learning. The big cost of our approach is the necessity of a formal grammar for both parsing and producing—a cost that can boil down to 1–2 programmer days if a formal grammar is already part of the system (say, as an input file for parser generators), but also extend to weeks if it is not. In the future, we will be experimenting with approaches that *mine grammars* from input samples and programs [65], [66] with the goal of extending the resulting grammars with probabilities for probabilistic fuzzing.

Debugging. Mined probabilistic grammars could be used to characterize the features of failure-inducing inputs, separating them from those of passing inputs. Statistical fault localization techniques [67], for instance, could then identify input elements most likely associated with a failure. Generating “common inputs”, as in this paper, and testing whether they cause failures, could further strengthen correlations between input patterns and failures, as well as narrow down the circumstances under which the failure occurs.

We are committed to making our research accessible for replication and extension. The source code of our parsers and production tools, the raw input samples, as well as all raw obtained data and processed charts is available as a replication package:

<https://tinyurl.com/inputs-from-hell>

ACKNOWLEDGMENTS

We thank the reviewers for their helpful comments. This work was (partially) funded by Deutsche Forschungsgemeinschaft, Project “Extracting and Mining of Probabilistic Event Structures from Software Systems (EMPRESS)”.

REFERENCES

- [1] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler, “EXE: Automatically generating inputs of death,” *ACM Trans. Inf. Syst. Secur.*, vol. 12, no. 2, pp. 10:1–10:38, 2008.
- [2] J. E. Hopcroft, R. Motwani, and J. D. Ullman, “Introduction to automata theory, languages, and computation,” *ACM SIGACT News*, vol. 32, no. 1, pp. 60–65, 2001.
- [3] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2006.
- [4] S. Luke, “Two fast tree-creation algorithms for genetic programming,” *IEEE Trans. Evol. Comput.*, vol. 4, no. 3, pp. 274–283, Sep. 2000.
- [5] T. Parr, *The Definitive ANTLR 4 Reference*, 2nd ed. Raleigh, North Carolina, USA: Pragmatic Bookshelf, 2013.
- [6] K. O’Hair, “HPROF: A Heap/CPU profiling tool in J2SE 5.0,” *Sun Developer Network, Developer Technical Articles & Tips*, vol. 28, 2004.
- [7] R. Feldt, R. Torkar, T. Gorscak, and W. Afzal, “Searching for cognitively diverse tests: Towards universal test diversity metrics,” in *Proc. 1st Int. Conf. Softw. Testing Verification Validation*, 2008, pp. 178–186.
- [8] Q. Shi, Z. Chen, C. Fang, Y. Feng, and B. Xu, “Measuring the diversity of a test set with distance entropy,” *IEEE Trans. Rel.*, vol. 65, no. 1, pp. 19–27, Mar. 2016.
- [9] W. Jin and A. Orso, “BugRedux: Reproducing field failures for in-house debugging,” in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 474–484.
- [10] A. Buja *et al.*, “Statistical inference for exploratory data analysis and model diagnostics,” *Philos. Trans. Roy. Soc. London A: Math. Phys. Eng. Sci.*, vol. 367, no. 1906, pp. 4361–4383, 2009.
- [11] Y. Fan, “Testing the goodness of fit of a parametric density function by kernel method,” *Econom. Theory*, vol. 10, no. 2, pp. 316–356, 1994.
- [12] S. Anand *et al.*, “An orchestrated survey of methodologies for automated software test case generation,” *J. Syst. Softw.*, vol. 86, no. 8, pp. 1978–2001, 2013.
- [13] A. Bertolino, “Software testing research: Achievements, challenges, dreams,” in *Proc. Int. Conf. Softw. Eng.*, 2007, pp. 85–103.
- [14] H. Zhu, P. A. V. Hall, and J. H. R. May, “Software unit test coverage and adequacy,” *ACM Comput. Surv.*, vol. 29, no. 4, pp. 366–427, 1997.
- [15] M. Damashek, “Gauging similarity with n-grams: Language-independent categorization of text,” *Science*, vol. 267, no. 5199, pp. 843–848, 1995.
- [16] N. Havrikov and A. Zeller, “Systematically covering input structure,” in *Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2019, pp. 189–199.
- [17] M. Mera, “Mining constraints for grammar fuzzing,” in *Proc. 28th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2019, pp. 415–418.
- [18] W. Visser, C. S. Pasareanu, and S. Khurshid, “Test input generation with Java PathFinder,” in *Proc. ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2004, pp. 97–107.
- [19] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury, “Directed greybox fuzzing,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2329–2344.
- [20] C. Cadar, D. Dunbar, and D. Engler, “KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs,” in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, pp. 209–224.
- [21] S. Khurshid, C. S. Pasareanu, and W. Visser, “Generalized symbolic execution for model checking and testing,” in *Proc. 9th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2003, pp. 553–568.
- [22] Y. Li, Z. Su, L. Wang, and X. Li, “Steering symbolic execution to less traveled paths,” in *Proc. ACM SIGPLAN Int. Conf. Object Oriented Program. Syst. Lang. Appl.*, 2013, pp. 19–32.
- [23] M. Christakis, P. Müller, and V. Wüstholtz, “Guiding dynamic symbolic execution toward unverified program executions,” in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 144–155.
- [24] N. Tillmann and J. de Halleux, “Pex—white box test generation for .NET,” in *Proc. 2nd Int. Conf. Tests Proofs*, 2008, pp. 134–153.
- [25] P. McMinn, “Search-based software testing: Past, present and future,” in *Proc. IEEE 4th Int. Conf. Softw. Testing Verification Validation Workshops*, 2011, pp. 153–163.
- [26] G. Fraser and A. Arcuri, “EvoSuite: Automatic test suite generation for object-oriented software,” in *Proc. 19th ACM SIGSOFT Symp./13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 416–419.
- [27] A. Panichella, F. M. Kifetew, and P. Tonella, “Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets,” *IEEE Trans. Softw. Eng.*, vol. 44, no. 2, pp. 122–158, Feb. 2018.
- [28] K. Mao, M. Harman, and Y. Jia, “Sapienz: Multi-objective automated testing for Android applications,” in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 94–105.
- [29] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball, “Feedback-directed random test generation,” in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 75–84.
- [30] B. P. Miller, L. Fredriksen, and B. So, “An empirical study of the reliability of UNIX utilities,” *Commun. ACM*, vol. 33, no. 12, pp. 32–44, Dec. 1990.
- [31] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed automated random testing,” in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2005, pp. 213–223.
- [32] K. Sen, D. Marinov, and G. Agha, “CUTE: A concolic unit testing engine for C,” in *Proc. 10th Eur. Softw. Eng. Conf. Held Jointly 13th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2005, pp. 263–272.
- [33] M. Böhme, V. Pham, M. Nguyen, and A. Roychoudhury, “Directed greybox fuzzing,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2017, pp. 2329–2344.
- [34] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, “VUZZer: Application-aware evolutionary fuzzing,” in *Proc. 24th Annu. Netw. Distrib. Syst. Secur. Symp.*, 2017, pp. 1–14.
- [35] L. D. Toffola, C. Staicu, and M. Pradel, “Saying ‘hi!’ is not enough: Mining inputs for effective test generation,” in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 44–49.

- [36] P. Liu, X. Zhang, M. Pistoia, Y. Zheng, M. Marques, and L. Zeng, "Automatic text input generation for mobile testing," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 643–653.
- [37] T. Su *et al.*, "Guided, stochastic model-based GUI testing of Android apps," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 245–256.
- [38] K. V. Hanford, "Automatic generation of test cases," *IBM Syst. J.*, vol. 9, no. 4, pp. 242–257, 1970.
- [39] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in C compilers," in *Proc. 32nd ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2011, pp. 283–294.
- [40] C. Holler, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proc. Presented as part 21st USENIX Secur. Symp.*, 2012, pp. 445–458.
- [41] H. Guo and Z. Qiu, "Automatic grammar-based test generation," in *Proc. 25th IFIP WG 6.1 Int. Conf. Testing Softw. Syst.*, 2013, pp. 17–32.
- [42] H. Guo and Z. Qiu, "A dynamic stochastic model for automatic grammar-based test generation," *Softw., Pract. Experience*, vol. 45, no. 11, pp. 1519–1547, 2015.
- [43] P. Godefroid, A. Kiezun, and M. Y. Levin, "Grammar-based whitebox fuzzing," in *Proc. 29th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2008, pp. 206–215.
- [44] N. Chomsky, *Syntactic Structures*. Berlin, Germany: Mouton, 1957.
- [45] S. Geman and M. Johnson, "Probabilistic grammars and their applications," in *Proc. Int. Encyclopedia Soc. Behavioral Sci.*, 2000, pp. 12 075–12 082.
- [46] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.
- [47] Y. Sakakibara *et al.*, "Stochastic context-free grammars for tRNA modeling," *Nucleic Acids Res.*, vol. 22, no. 23, pp. 5112–5120, 1994.
- [48] T. Liu, S. Chaudhuri, V. G. Kim, Q. Huang, N. J. Mitra, and T. Funkhouser, "Creating consistent scene graphs using a probabilistic grammar," *ACM Trans. Graph.*, vol. 33, no. 6, pp. 211:1–211:12, Nov. 2014.
- [49] O. Cekan and Z. Kotasek, "A probabilistic context-free grammar based random test program generation," in *Proc. Euromicro Conf. Digit. Syst. Des.*, 2017, pp. 356–359.
- [50] R. Lämmel and W. Schulte, "Controllable combinatorial coverage in grammar-based testing," in *Proc. IFIP Int. Conf. Testing Communicating Syst.*, 2006, pp. 19–38.
- [51] A. Groce, C. Zhang, E. Eide, Y. Chen, and J. Regehr, "Swarm testing," in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 78–88.
- [52] M. A. Alipour, A. Groce, R. Gopinath, and A. Christi, "Generating focused random tests using directed swarm testing," in *Proc. 25th Int. Symp. Softw. Testing Anal.*, 2016, pp. 70–81.
- [53] R. Feldt and S. M. Poulding, "Finding test data with specific properties via metaheuristic search," in *Proc. IEEE 24th Int. Symp. Softw. Rel. Eng.*, 2013, pp. 350–359.
- [54] S. M. Poulding, R. Alexander, J. A. Clark, and M. J. Hadley, "The optimisation of stochastic grammars to enable cost-effective probabilistic structural testing," *J. Syst. Softw.*, vol. 103, pp. 296–310, 2015.
- [55] F. M. Kifetew, R. Tiella, and P. Tonella, "Combining stochastic grammars and genetic programming for coverage testing at the system level," in *Proc. 6th Int. Symp. Search-Based Softw. Eng.*, 2014, pp. 138–152.
- [56] F. M. Kifetew, R. Tiella, and P. Tonella, "Generating valid grammar-based test inputs by means of genetic programming and annotated grammars," *Empir. Softw. Eng.*, vol. 22, no. 2, pp. 928–961, 2017.
- [57] F. M. Kifetew, W. Jin, R. Tiella, A. Orso, and P. Tonella, "Reproducing field failures for programs with complex grammar-based input," in *Proc. 7th IEEE Int. Conf. Softw. Testing Verification Validation*, 2014, pp. 163–172.
- [58] J. Patra and M. Pradel, "Learning to fuzz: Application-independent fuzz testing with probabilistic, generative models of input data," Technical University of Darmstadt, Darmstadt, Germany, Tech. Rep. TUD-CS-2016-14664, Nov. 2016.
- [59] S. Herfert, J. Patra, and M. Pradel, "Automatically reducing tree-structured test inputs," in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 861–871.
- [60] O. Bastani, R. Sharma, A. Aiken, and P. Liang, "Synthesizing program input grammars," in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2017, pp. 95–110.
- [61] P. Godefroid, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 50–59.
- [62] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," in *Proc. IEEE Symp. Secur. Privacy*, 2017, pp. 579–594.
- [63] M. Zalewski, "American fuzzy lop," 2018. Accessed: Jan. 28, 2018. [Online]. Available: <http://lcamtuf.coredump.cx/afl/>
- [64] Z. Lin and X. Zhang, "Deriving input syntactic structure from execution," in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2008, pp. 83–93.
- [65] M. Höschele and A. Zeller, "Mining input grammars from dynamic taints," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 720–725.
- [66] R. Gopinath, B. Mathis, and A. Zeller, "Mining input grammars from dynamic control flow," in *Proc. ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2020.
- [67] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa, "A survey on software fault localization," *IEEE Trans. Softw. Eng.*, vol. 42, no. 8, pp. 707–740, Aug. 2016.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csl.

On How Bit-Vector Logic Can Help Verify LTL-Based Specifications

Mohammad Mehdi Pourhashem Kallehbasti^{ID}, Matteo Rossi^{ID}, and Luciano Baresi^{ID}

Abstract—This paper studies how bit-vector logic (bv logic) can help improve the efficiency of verifying specifications expressed in Linear Temporal Logic (LTL). First, it exploits the notion of Bounded Satisfiability Checking to propose an improved encoding of LTL formulae into formulae of bv logic, which can be formally verified by means of Satisfiability Modulo Theories (SMT) solvers. To assess the gain in efficiency, we compare the proposed encoding, implemented in our tool *Zot*, against three well-known encodings available in the literature: the classic bounded encoding and the optimized, incremental one, as implemented in both NuSMV and nuXmv, and the encoding optimized for metric temporal logic, which was the “standard” implementation provided by *Zot*. We also compared the newly proposed solution against five additional efficient algorithms proposed by nuXmv, which is the state-of-the-art tool for verifying LTL specifications. The experiments show that the new encoding provides significant benefits with respect to existing tools. Since the first set of experiments only used Z3 as SMT solver, we also wanted to assess whether the benefits were induced by the specific solver or were more general. This is why we also embedded different SMT solvers in *Zot*. Besides Z3, we also carried out experiments with CVC4, Mathsat, Yices2, and Boolector, and compared the results against the first and second best solutions provided by either NuSMV or nuXmv. Obtained results witness that the benefits of the bv logic encoding are independent of the specific solver. Bv logic-based solutions are better than traditional ones with only a few exceptions. It is also true that there is no particular SMT solver that outperformed the others. Boolector is often the best as for memory usage, while Yices2 and Z3 are often the fastest ones.

Index Terms—Formal methods, linear temporal logic, bounded satisfiability checking, bit-vector logic

1 INTRODUCTION

LINEAR Temporal Logic [1] (LTL) plays a key role in computer science. It has been used for the specification and verification of (possibly safety-critical) programs [2], the generation of test cases [3], the synthesis of controllers [4], the formalization of notations (e.g., UML) [5], the run-time verification of systems [6], and as planning formalism [7]. However, one of the key factors that still hamper the widespread adoption of this formalism in practice is the limited efficiency and scalability of verification tools.

While various techniques have used automata in the past to formally verify LTL models [8], this work exploits the notion of *Bounded Satisfiability Checking* (BSC) [9], a variant of Bounded Model Checking (BMC) [10]. BSC requires that LTL formulae be suitably translated into formulae of another decidable logic, such as propositional logic, that precisely capture ultimately periodic models of the original formulae of length up to a bound k . Produced formulae are

then fed to a solver for the target logic (e.g., a SAT or SMT solver) for verification (up to bound k).

To tackle efficiency, this article presents *bit-vector logic* (bv logic) as means to encode LTL formulae and speed-up their verification. This logic allows SMT solvers to exploit the representation of the different temporal values of variables as vectors and to carry out simplifications and optimizations at word (vector) level. Our initial work [11] demonstrated the feasibility of the approach, proposed an initial encoding, and demonstrated it was able to scale better than the “usual” Boolean-based ones by exploiting Z3 [12] as SMT solver.

This paper moves a step forward and generalizes the outcome. It first proposes a new bv logic-based encoding, which significantly improves the original one [11]. Besides highlighting the novel aspects, we implemented it as additional plug-in of our bounded satisfiability checker *Zot* [13]. Its architecture helped us implement different encodings as independent plug-ins and carry out the experiments more easily. To assess the efficiency gain we carried out a first set of experiments, reported in Section 4, to compare the new encoding against solutions already proposed by *Zot* and by NuSMV [14] and nuXmv¹ [16], which are the de-facto standard for bounded verification of LTL specifications (we did not consider tools like SPIN [17] because they employ other, different verification techniques).

We used *Zot* for reusing the old bv logic-based encoding [11] and the “standard” LTL encoding [9]. We also used

• Mohammad Mehdi Pourhashem Kallehbasti is with the Department of Computer Engineering, University of Science and Technology of Mazandaran, Behshahr, Iran. E-mail: pourhashem@mazust.ac.ir.
 • Matteo Rossi is with the Dipartimento di Meccanica, Politecnico di Milano, 20133 Milano, Italy. E-mail: matteo.rossi@polimi.it.
 • Luciano Baresi is with the Dipartimento di Elettronica Informazione e Bio-ingegneria, Politecnico di Milano, 20133 Milano, Italy.
 E-mail: luciano.baresi@polimi.it.

Manuscript received 2 Nov. 2019; revised 29 Apr. 2020; accepted 22 July 2020. Date of publication 5 Aug. 2020; date of current version 18 Apr. 2022.

(Corresponding author: Mohammad Mehdi Pourhashem Kallehbasti.)

Recommended for acceptance by C. Wang.

Digital Object Identifier no. 10.1109/TSE.2020.3014394

1. nuXmv is an extension of NuSMV that comes with strong SAT-based algorithms as well as SMT-based verification techniques integrated with MathSAT5 [15].

both NuSMV and nuXmv to try with three “classical”, Boolean logic-based encodings available in the literature: (i) the classic bounded encoding [18]; (ii) the optimized encoding [19], and (iii) the improved and incremental version [12], [20]. We also exploited nuXmv for five additional verification algorithms that both adopt diverse verification techniques and exploit specific optimizations to solve particular problems.

Obtained results show that the new solution, implemented as *Zot* plug-in and based on *Z3*, is almost always the fastest option and consumes less memory. The most significant exception is the verification of the Fischer protocol, where the *k-live* solution proposed by nuXmv is the best because it is able to subsume the UNSAT result without necessarily iterating up to the maximum bound. Our experiments also suggest that this solution (*k-live*) only works well with a few small models.

The second set of experiments we carried out aimed to assess whether efficiency benefits were independent of the particular SMT solver used —*Z3* in the initial set of experiments. This is why we exploited *Zot* one more time to implement plug-ins and compare the top five solvers in recent SMT competitions [21]: Boolector [22], Yices2 [23], Mathsat [15], CVC4 [24], and *Z3*.

In this paper we focus on the verification of LTL specifications, which are finite-state models. The bv logic-based encoding presented here has also been used to improve the efficiency of the verification technique of infinite-state models presented in [25]. We do not present this work in this paper for the sake of brevity.

All these experiments helped us reject the claim that the gain was mainly due to the efficiency of *Z3*, and clearly highlight the benefits of the bv logic encoding. Obtained results witness that the benefits are independent of the specific solver. Bv logic-based solutions are better than traditional ones with only a few exceptions. There is however no specific solver that outperformed the others. Boolector is often the best as for memory usage, while Yices2 and *Z3* are often the fastest options.

To summarize, this article extends the work initially presented in [11] with: (i) an improved, and more efficient, bv logic encoding of LTL formulae; (ii) a new and more thorough set of experiments to compare the efficiency of our *Zot*- and *Z3*-based solution against the best Boolean logic-based approaches and additional algorithms (provided by nuXmv); and (iii) a wider comparison to assess the impact of different SMT solvers on the efficiency of the proposed solution.

The rest of this article is organized as follows. Section 2 introduces LTL, briefly sketches logic-based system verification, and describes the existing bounded Boolean-based encoding for LTL. Section 3 explains the improved bv logic-based encoding for LTL and highlights the differences with respect to the original one [11]. Section 4 describes the tools we used for evaluation, the experiments we carried out, and the results we obtained. Section 5 surveys related approaches and Section 6 concludes the article.

2 PRELIMINARIES

2.1 Linear Temporal Logic

LTL [1] is a widely-used specification logic. In this article, we focus on the version with both future and past temporal operators: although past operators do not increase the

expressiveness of the logic, they are advantageous for compositional reasoning [26]. In addition, LTL with past operators is exponentially more concise than its future-only counterpart [27].

An LTL formula ϕ is defined over a set of atomic propositions AP by means of the following grammar:

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid \mathbf{X}\phi \mid \mathbf{Y}\phi \mid \phi\mathbf{U}\phi \mid \phi\mathbf{S}\phi,$$

where $p \in AP$, \neg and \wedge have the usual meaning, \mathbf{X} and \mathbf{U} are the “next” and “until” future operators, and \mathbf{Y} (“yesterday”) and \mathbf{S} (“since”) are their past counterparts. Complex formulae are composed of sub-formulae: for example, $p\mathbf{UX}p \wedge q$ comprises p , q , $p \wedge q$, and $\mathbf{X}p \wedge q$.

The semantics of LTL is given in terms of infinite sequences of sets of atomic propositions, or *words*. A word $\pi : \mathbb{N} \rightarrow 2^{AP}$ assigns to every instant of the temporal domain \mathbb{N} the (possibly empty) set of atomic propositions that hold in that instant. We can think of a word as an infinite sequence of states $\pi = s_0s_1s_2\dots$, where each state is labeled with the atomic propositions that hold in it. We say that a word π satisfies formula ϕ at instant i , written $\pi, i \models \phi$, if ϕ holds when evaluated starting from instant i of π . The following is the usual formal semantics of the satisfiability relation for LTL

$$\begin{aligned} \pi, i \models p &\Leftrightarrow p \in \pi(i) \text{ for } p \in AP \\ \pi, i \models \neg\phi &\Leftrightarrow \pi, i \not\models \phi \\ \pi, i \models \phi_1 \wedge \phi_2 &\Leftrightarrow \pi, i \models \phi_1 \text{ and } \pi, i \models \phi_2 \\ \pi, i \models \mathbf{X}\phi &\Leftrightarrow \pi, i+1 \models \phi \\ \pi, i \models \mathbf{Y}\phi &\Leftrightarrow i > 0 \text{ and } \pi, i-1 \models \phi \\ \pi, i \models \phi_1 \mathbf{U} \phi_2 &\Leftrightarrow \exists j \geq i \text{ s.t. } \pi, j \models \phi_2 \\ &\quad \text{and } \forall n \text{ s.t. } i \leq n < j : \pi, n \models \phi_1 \\ \pi, i \models \phi_1 \mathbf{S} \phi_2 &\Leftrightarrow \exists j \leq i \text{ s.t. } \pi, j \models \phi_2 \end{aligned}$$

We say that a word π satisfies formula ϕ when it holds at the first instant of the temporal domain, i.e., when $\pi, 0 \models \phi$ holds. In this case we will sometimes write $\pi \models \phi$. A word π that satisfies ϕ is a *model* for ϕ .

Starting from the basic connectives and operators, it is customary to introduce the other traditional Boolean connectives (\vee , \Rightarrow , ...), and temporal operators as abbreviations. In particular the “eventually in the future” (\mathbf{F}), “globally in the future” (\mathbf{G}) and “release” (\mathbf{R}) operators (and their past counterparts “eventually in the past” \mathbf{P} , “historically” \mathbf{H} and “trigger” \mathbf{T}) are defined as follows: $\mathbf{F}\phi = \mathbf{T}\mathbf{U}\phi$, $\mathbf{G}\phi = \neg\mathbf{F}\neg\phi$, $\mathbf{F}_1\mathbf{R}\phi_2 = \neg(\neg\phi_1\mathbf{U}\neg\phi_2)$, $\mathbf{P}\phi = \mathbf{T}\mathbf{S}\phi$, $\mathbf{H}\phi = \neg\mathbf{P}\neg\phi$, and $\mathbf{F}_1\mathbf{T}\phi_2 = \neg(\neg\phi_1\mathbf{S}\neg\phi_2)$.

LTL is then often used to model (complex) systems and the properties they must comply with, in a so-called *descriptive* approach [28]. If formulae S and ϕ describe system and property to be checked, respectively, satisfiability checking can help prove if ϕ holds (or fails) for S , since a formula is *valid* iff its negation is unsatisfiable [28]. $S \Rightarrow \phi$, which captures the fact that property ϕ holds for S , can be proven valid if its negation ($S \wedge \neg\phi$) is shown to be unsatisfiable, otherwise a trace that satisfies $S \wedge \neg\phi$ would witness the failure of property ϕ for system S .

For the sake of simplicity, let us introduce a simple running example used throughout the paper to materialize the

main concepts. A synchronous shift-register returns every received bit after a delay of two time instants. This system can be specified by the LTL formula $S : \mathbf{G}(in \Leftrightarrow \mathbf{XX}out)$, which states that in holds at the current time instant iff out will hold at the second time instant from now. Consider property $P_1 : \mathbf{FG}\neg in$, which asserts that there is a time instant in the future at which in stops occurring; one can easily show that P_1 does not hold for S by producing a counterexample in which in occurs infinitely often. This can be proven by checking the satisfiability of formula $S \wedge \neg P_1$, which leads to a counterexample. On the other hand, property $P_2 : \mathbf{FG}\neg in \Rightarrow \mathbf{FG}\neg out$, which states that, if in ceases to occur after a certain point in time, then out eventually ceases to occur, holds for S . Indeed, there is not a single trace of S in which P_2 is falsified, which means that $S \wedge \neg P_2$ is unsatisfiable.

2.2 Bounded Satisfiability Checking

Bounded Satisfiability Checking is a well-known satisfiability checking technique. It is based on the idea of translating a temporal logic formula ψ into a formula of propositional logic that represents infinite, *ultimately periodic* models of ψ —i.e., sequences of states of the form $\pi = s_0s_1\dots s_{l-1}(s_ls_{l+1}\dots s_k)^\omega$, where k is a parameter called the *bound* of the model. As discussed in Section 2.1, then, if one wants to validate the specification of a system S against property ϕ using a BSC approach, the formula to be translated is $S \wedge \neg\phi$, and one must look for an ultimately periodic sequence of states $\pi = s_0s_1\dots s_{l-1}(s_ls_{l+1}\dots s_k)^\omega$ of S that violates ϕ . If a counterexample that witnesses the violation of the property exists, then the property does not hold for S . If no counterexample of length up to k is found, then the property holds for S provided that k is big enough. For example, back to the running example, property P_1 does not hold for S because of the counterexample $\pi = \{\} \{in\} \{\} (\{in, out\})^\omega$, where we have an in at the second time instant and from the forth time instant onwards both in and out occur every other time instant forever.

BSC can be easily carried out by an SMT solver by translating LTL formulae properly. The classic encoding technique into propositional logic [18] represents states $s_0\dots s_l\dots s_k$, and then the fact that the state after s_k , say s_{k+1} , is in fact s_l again. Hence, the bounded encoding captures finite sequences of states of the form $\alpha s \beta s$, where $\alpha = s_0s_1\dots s_{l-1}$, $\beta = s_{l+1}\dots s_k$, and $s = s_l = s_{k+1}$.

The encoding is defined as Boolean constraints over so-called *formula variables* $[[\psi]]_i$. These are Boolean variables that are used to represent the values of all subformulae of the LTL formula to be checked for satisfiability at instants $0, 1, \dots, k+1$. More precisely, given an LTL formula ϕ and a bound k , the encoding introduces $k+2$ formula variables $[[\psi]]_0, [[\psi]]_1, \dots, [[\psi]]_{k+1}$ for each subformula ψ of ϕ to capture whether ψ is true or not at the various instants in $[0, k+1]$.

In addition, the encoding introduces $k+1$ *loop selector variables* l_0, l_1, \dots, l_k , which are fresh Boolean variables such that l_i is true iff the loop starts at position i (hence, if l_i is true, then $s_l = s_{k+1}$); at most one of l_0, l_1, \dots, l_k can be true. Other Boolean variables are introduced for convenience: the $k+1$ variables $InLoop_i$, with $0 \leq i \leq k$, are such that $InLoop_i$ is true iff position i is in the loop (i.e., $l \leq i \leq k$). Finally, variable $LoopExists$ is true iff the desired loop exists.

TABLE 1
Constraints Defined to Capture the Semantics of LTL Formulae

$ \text{LoopConstraints} _k$	
Base	$\neg l_0 \wedge \neg InLoop_0$
$1 \leq i \leq k$	$(l_i \Rightarrow s_{i-1} = s_k) \wedge (InLoop_i \Leftrightarrow InLoop_{i-1} \vee l_i)$ $\wedge (InLoop_{i-1} \Rightarrow \neg l_i) \wedge (LoopExists \Leftrightarrow InLoop_k)$
$ \text{LastStateConstraints} _k$	
Base	$\neg LoopExists \Rightarrow \neg [[\phi]]_{k+1}$
$1 \leq i \leq k$	$l_i \Rightarrow ([[\phi]]_{k+1} \Leftrightarrow [[\phi]]_i)$
$ \text{PropConstraints} _k$	
ϕ	$0 \leq i \leq k+1$
p	$[[p]]_i \Leftrightarrow p \in \pi(i)$
$\neg p$	$[[\neg p]]_i \Leftrightarrow p \notin \pi(i)$
$\psi_1 \wedge \psi_2$	$[[\psi_1 \wedge \psi_2]]_i \Leftrightarrow [[\psi_1]]_i \wedge [[\psi_2]]_i$
$\psi_1 \vee \psi_2$	$[[\psi_1 \vee \psi_2]]_i \Leftrightarrow [[\psi_1]]_i \vee [[\psi_2]]_i$
$ \text{TempConstraints} _k$ for future operators	
ϕ	$0 \leq i \leq k$
$\mathbf{X}\psi$	$[[\mathbf{X}\psi]]_i \Leftrightarrow [[\psi]]_{i+1}$
$\psi_1 \mathbf{U} \psi_2$	$[[\psi_1 \mathbf{U} \psi_2]]_i \Leftrightarrow [[\psi_2]]_i \vee ([[[\psi_1]]_i \wedge [[\psi_1 \mathbf{U} \psi_2]]_{i+1}})$
$\psi_1 \mathbf{R} \psi_2$	$[[\psi_1 \mathbf{R} \psi_2]]_i \Leftrightarrow [[\psi_2]]_i \wedge ([[[\psi_1]]_i \vee [[\psi_1 \mathbf{R} \psi_2]]_{i+1}})$
$ \text{Eventualities} _k$	
Base	$LoopExists \Rightarrow ([[[\psi_1 \mathbf{U} \psi_2]]_k \Rightarrow \langle\langle F\psi_2\rangle\rangle_k)}$ $LoopExists \Rightarrow ([[[\psi_1 \mathbf{R} \psi_2]]_k \Leftarrow \langle\langle G\psi_2\rangle\rangle_k)}$
$\psi_1 \mathbf{U} \psi_2$	$\langle\langle F\psi_2\rangle\rangle_0 \Leftrightarrow \perp$
$\psi_1 \mathbf{R} \psi_2$	$\langle\langle G\psi_2\rangle\rangle_0 \Leftrightarrow \top$
$1 \leq i \leq k$	$[[\psi_1 \mathbf{U} \psi_2]]_i \Leftrightarrow \langle\langle F\psi_2\rangle\rangle_{i-1} \vee (InLoop_i \wedge [[\psi_2]]_i)$ $[[\psi_1 \mathbf{R} \psi_2]]_i \Leftrightarrow \langle\langle G\psi_2\rangle\rangle_{i-1} \wedge (\neg InLoop_i \vee [[\psi_2]]_i)$
$ \text{TempConstraints} _k$ for past operators	
ϕ	$0 < i \leq k+1$
$\mathbf{Y}\psi$	$[[\mathbf{Y}\psi]]_i \Leftrightarrow [[\psi]]_{i-1}$
$\mathbf{Z}\psi$	$[[\mathbf{Z}\psi]]_i \Leftrightarrow [[\psi]]_{i-1}$
$\psi_1 \mathbf{S} \psi_2$	$[[\psi_1 \mathbf{S} \psi_2]]_i \Leftrightarrow [[\psi_2]]_i \vee ([[[\psi_1]]_i \wedge [[\psi_1 \mathbf{S} \psi_2]]_{i-1}})$
$\psi_1 \mathbf{T} \psi_2$	$[[\psi_1 \mathbf{T} \psi_2]]_i \Leftrightarrow [[\psi_2]]_i \wedge ([[[\psi_1]]_i \vee [[\psi_1 \mathbf{T} \psi_2]]_{i-1}})$
$ \text{TempConstraints} _k$ in the origin.	
ϕ	Base
$\mathbf{Y}\psi$	$\neg [[\mathbf{Y}\psi]]_0$
$\mathbf{Z}\psi$	$[[\mathbf{Z}\psi]]_0$
$\psi_1 \mathbf{S} \psi_2$	$[[\psi_1 \mathbf{S} \psi_2]]_0 \Leftrightarrow [[\psi_2]]_0$
$\psi_1 \mathbf{T} \psi_2$	$[[\psi_1 \mathbf{T} \psi_2]]_0 \Leftrightarrow [[\psi_2]]_0$

Table 1 introduces the constraints that are imposed on the Boolean variables introduced above to capture the semantics of LTL formulae. Constraints $|\text{LoopConstraints}|_k$ formalize the semantics of Boolean variables $\{l_i\}_{i \in [0,k]}$, $\{InLoop_i\}_{i \in [0,k]}$ and $LoopExists$ (e.g., the existence of at most one loop). In addition, as mentioned in [18], they impose that the same atomic propositions that hold in state s_k also hold in state s_{l-1} , which has been shown to improve the efficiency of the satisfiability checking.

Constraints $|\text{LastStateConstraints}|_k$ define that the subformulae of ϕ that hold in s_{k+1} are the same as those that hold in state s_l . This effectively defines that after state s_k the bounded trace loops back to state s_l .

The subsequent constraints define the semantics of the propositional connectives and of the temporal operators. Constraints $|\text{PropConstraints}|_k$ capture the semantics of propositional connectives. For example, they state that the

value of $||[p]||_i$ and $||[\neg p]||_i$ capture whether propositional letter p holds at instant i or not. The definitions of $||[\psi_1 \wedge \psi_2]||$ and of $||[\psi_1 \vee \psi_2]||$ are straightforward. Note that the Boolean encoding was defined for LTL formulae in Positive Normal Form (PNF), that is, negations can only appear next to atomic propositions. This can save some formula variables, but the encoding can be easily generalized to formulae that are not in PNF.

Constraints $|TempConstraints|_k$ define the semantics of the temporal operators, both future (\mathbf{X} , \mathbf{U} and \mathbf{R}) and past ones (\mathbf{Y} , \mathbf{S} and \mathbf{T}). The semantics of \mathbf{U} and \mathbf{R} is defined through their standard fixpoint characterization and through the introduction of the set of constraints $|Eventualities|_k$.

The latter constraints are used to ensure that, if $\psi_1 \mathbf{U} \psi_2$ holds in s_k , then ψ_2 occurs infinitely often, that is, it occurs somewhere in the loop. Similarly, if $\psi_1 \mathbf{R} \psi_2$ occurs in s_k , then either ψ_2 holds throughout the loop, or at some point of the loop ψ_1 holds. $\langle\langle F\psi_2 \rangle\rangle_i$ and $\langle\langle G\psi_2 \rangle\rangle_i$ are auxiliary variables required for capturing these constraints. $\langle\langle F\psi_2 \rangle\rangle_i$ holds if position i belongs to the loop and ψ_2 holds in at least one position between l and i . Accordingly, $\langle\langle F\psi_2 \rangle\rangle_k$ means that ψ_2 holds somewhere in the loop. Therefore, constraint $LoopExists \Rightarrow (||[\psi_1 \mathbf{U} \psi_2]||_k \Rightarrow \langle\langle F\psi_2 \rangle\rangle_k)$ does not allow $\psi_1 \mathbf{U} \psi_2$ to hold at k , if ψ_2 does not occur infinitely often. Similarly, $\langle\langle G\psi_2 \rangle\rangle_k$ holds iff ψ_2 holds everywhere in the loop. Then, constraint $LoopExists \Rightarrow (||[\psi_1 \mathbf{R} \psi_2]||_k \Leftarrow \langle\langle G\psi_2 \rangle\rangle_k)$ forces $||[\psi_1 \mathbf{R} \psi_2]||_k$ to hold if ψ_2 holds from position l on.

Similar constraints define the semantics of the past operators \mathbf{Y} , \mathbf{S} and \mathbf{T} , which is symmetrical to their future counterparts. We also define operator \mathbf{Z} , which is necessary for formulae in PNF, which is simply a variant of \mathbf{Y} such that $\mathbf{Z}\psi$ holds in 0 no matter ψ . Since the temporal domain is mono-infinite (i.e., it is infinite only towards the future), there is no need to impose eventuality constraints over past operators. However, we must define the value of past operators in the origin 0 (constraints $|TempConstraints|_k$ in the origin).

Finally, given an LTL formula ϕ , its Boolean encoding ϕ_B is given by the conjunction of the constraints in sets $|LoopConstraints|_k$, $|LastStateConstraints|_k$, $|PropConstraints|_k$, $|TempConstraints|_k$, and $|Eventualities|_k$, plus the statement that ϕ holds in the origin, i.e., $||[\phi]||_0$.

2.3 Bit-Vector Logic

A bit-vector is an array whose elements are bits (Booleans). In bit-vector logic, the size of a bit-vector (number of bits) is finite, and can be any nonzero number in \mathbb{N} . We use the notation $\overleftarrow{x}_{[n]}$ for the bit-vector \overleftarrow{x} with size n , or simply \overleftarrow{x} when the size is not important or can be inferred from the context. Furthermore, $\overleftarrow{x}_{[n]}^{[i]}$ stands for the i th bit in the bit-vector \overleftarrow{x} , where bits are indexed from right to left. Accordingly, $\overleftarrow{x}_{[n]}^{[n-1]}$ is the leftmost and most significant bit, and $\overleftarrow{x}_{[n]}^{[0]}$ is the rightmost and least significant bit. For constants we use the notation $\overleftarrow{c}_{[n]}$, which is the two's complement representation of integer c over n bits. For example, $\overleftarrow{-2}_{[4]}$ is 1110.

Bv logic offers a wide range of operators. The two core operators are *concatenation* and *extraction*. *Concatenation*: $\overleftarrow{x}_{[n]} :: \overleftarrow{y}_{[m]}$ is a bit-vector $\overleftarrow{z}_{[n+m]}$, such that $\overleftarrow{z}_{[0]} = \overleftarrow{y}_{[0]}$ and $\overleftarrow{z}_{[m+n-1]} = \overleftarrow{x}_{[n-1]}$. For example, $111 :: 0 = 1110$. *Extraction*:

$\overleftarrow{x}_{[j:i]}$ is a bit-vector $\overleftarrow{z}_{[j-i+1]}$, where $\overleftarrow{z}_{[0]} = \overleftarrow{x}_{[i]}$ and $\overleftarrow{z}_{[j-i]} = \overleftarrow{x}_{[j]}$, which can be defined through concatenation as $\overleftarrow{x}_{[j:i]} = ::_{k=j}^i x^{[k]}$. For example, $1100^{[2:0]} = 100$.

Arithmetic operators *addition* (+) and *subtraction* (-) throw away the final carry bit and the resulting bit-vector has the same size as the operands. *Unsigned shift to the right/left* (\gg/\ll) throws away the rightmost/leftmost bit and inserts zero from the left/right. For example, $\gg 1100 = 0110$ and $\ll 1100 = 1000$. In general, $\ll^n \overleftarrow{x}$ (resp., $\gg^n \overleftarrow{x}$) is the operation that applies \ll (resp., \gg) to \overleftarrow{x} n times.

We also use bitwise operators like *negation* (!), *conjunction* (&), *disjunction* (!), *reduction or* (\uparrow), and *reduction and* (\downarrow). The *reduction and* operator is defined as $\downarrow \overleftarrow{x}_{[n]} = \&_{i=0}^{n-1} \overleftarrow{x}_{[n]}^{[i]}$ (i.e., it is the “and” of all the bits in \overleftarrow{x}). The size of the resulting bit-vector is one. The bit corresponds to the minimum value in \overleftarrow{x} ; in other words, it is equal to *one* if all the bits of the bit-vector \overleftarrow{x} are *one*, *zero* otherwise.

Bit-vectors (or parts thereof) can be compared using the usual relational operators =, <, and formulae of bv logic can be built using the usual Boolean connectives \neg , \wedge .

3 BIT-VECTOR-BASED ENCODING

Before introducing our new bv logic-based encoding, we want to motivate the choice of this logic.

The truth values of an LTL formula at the time instants from 0 to k are a series of *trues* or *falses*, and the value at a particular time instant is logically related to the values at the other instants. If one adopted a Boolean encoding, each value would be stored in an independent variable and the broader view is disregarded. While a bit-vector is a collection of Boolean values, the key difference lies in the way constraints are managed. If they are asserted on a set of (independent) Boolean values, the solver is blind to their interrelations and no simplifications can be carried out at word level. In contrast, when these values are stored in a single vector (word), SMT solvers can apply simplifications and optimizations (more) efficiently. Essentially, more information is provided to the solver in the latter case.

While a thorough assessment of the impact of these simplifications is out of the scope of this paper [29] (see also Section 4 for our empirical results), we invite the reader to focus on the trivially unsatisfiable LTL formula $((a\mathbf{Ub} \vee \neg a\mathbf{R}\neg b)\mathbf{U}c) \wedge \neg Fc$. By definition, $a\mathbf{Ub}$ is equivalent to $\neg(\neg a\mathbf{R}\neg b)$, which reduces $a\mathbf{Ub} \vee \neg a\mathbf{R}\neg b$ to \top . Besides, $\top\mathbf{U}c$ is another form of Fc , which reduces the LTL formula to $Fc \wedge \neg Fc$, that is \perp . These simplifications are not easy for a solver, especially when the whole formula is asserted at the Boolean level. Since only Z3 shows its intermediate steps, we can report its behavior, but we argue it can be generalized. Z3 simplifies the Boolean formula produced by the classic Boolean encoding into another Boolean formula that then must be solved. In contrast, the bv logic formula produced by *sbvzot* is simplified and reduced to \perp , and thus the result is UNSAT, without solving any formula. With the Boolean encoding, the solver computes the Boolean variables for time instants i and $i + 1$, which are false, by resolving different constraints. It is not aware that they both represent the same sub-formula (\perp) at various time instants. In a bv logic-based encoding, the solver knows that bit i and $i + 1$ are zero, not by solving constraints at bit level (Boolean

TABLE 2
A Counterexample That Falsifies Property
 P_1 of the Running Example

subformula	bit-vector	5 4 3 2 1 0
in	$\langle \overleftarrow{in} \rangle$	1 0 1 0 1 0
out	$\langle \overleftarrow{out} \rangle$	1 0 1 0 0 0
$f_1 : Xout$	$\langle \overleftarrow{Xout} \rangle$	0 1 0 1 0 0
$f_2 : XXout$	$\langle \overleftarrow{Xf_1} \rangle$	1 0 1 0 1 0
$in \Leftrightarrow XXout$	$\langle \overleftarrow{in \Leftrightarrow f_2} \rangle$	1 1 1 1 1 1
	$\langle \overleftarrow{lpos} \rangle$	0 0 0 0 1 1
	$\langle \overleftarrow{inloop} \rangle$	1 1 1 0 0 0

values), but by simplifying the formula at vector level since both bits are parts of the same bit vector (\perp).

This example shows that bv logic can indeed enable simplifications that Boolean logic does not. However, in this specific example, since the formula is quite small, the solving time is quite small. Section 4 witnesses that the bigger formulae become, the higher the gain is.

3.1 sbvzot

bvzot is the first bv logic-based encoding for LTL we developed [11], *sbvzot* (simple *bvzot*) is the new encoding presented in this paper. *sbvzot*: (i) does not use binary arithmetic operations (addition and subtraction), (ii) introduces as many bit-vectors as the number of subformulae in a formula (not only for its propositional letters), (iii) and adds “last state constraints” for all operators (not only for past ones). This encoding—which, from a purely syntactic point of view, is usually more concise than *bvzot*—is the result of diverse experiments that explored different tweaks and solutions. *sbvzot* is overall the best one in terms of efficiency.

Similarly to the classic Boolean encoding of Section 2.2, *sbvzot* uses bit-vectors to represent the truth value of each subformula in time instants $[0, k+1]$. More precisely, to encode an LTL formula ϕ , for each subformula ψ of ϕ we introduce a bit-vector, $\langle \psi \rangle_{[k+2]}$ (i.e., of size $k+2$), such that $\langle \psi \rangle_{[i]}_{[k+2]}$, with $i \in [0, k+1]$, captures the value of subformula ψ at instant i .²

In addition to a bit-vector for each subformula ψ , we also introduce a bit-vector, $\langle lpos \rangle_{[k+2]}$, that contains (encoded in binary) position *pos* of the loop in interval $[0, k+1]$ and a bit-vector, $\langle inloop \rangle_{[k+2]}$, where the bit at position i is 1 iff the position i is inside the periodic part. For the sake of uniformity, we encode \perp (false) as $\overleftarrow{0}_{[k+2]}$ (i.e., a sequence of zeros) and \top (true) as $\overleftarrow{1}_{[k+2]}$ (i.e., a sequence of ones), so the size of all bit-vectors used in the encoding is $k+2$. Note that, given a formula ϕ , and its vector $\langle \phi \rangle$, $\langle \phi \rangle \& \overleftarrow{!} \langle \phi \rangle = \perp$ and $\langle \phi \rangle \overleftarrow{!} \langle \phi \rangle = \top$.

To define the value of bit-vector $\langle inloop \rangle_{[k+2]}$ we introduce constraint $\langle inloop \rangle_{[k+2]} = \overleftarrow{\ll^{pos}} \overleftarrow{1}_{[k+2]}$.

For example, Table 2 shows an exemplar trace, along with $\langle lpos \rangle$, and $\langle inloop \rangle$, where we assume that k is 4 and

2. Recall that $\overleftarrow{\psi}^{[0]}$ is the right-most (least significant) bit in $\overleftarrow{\psi}$, and $\overleftarrow{\psi}^{[k+1]}$ is the left-most (most significant) one.

TABLE 3
Constraints in bv logic That Define the Value of ϕ

$ SBVPropConstraints _k$	
	bit-vector encoding
ϕ	$\overleftarrow{\neg \psi} = !\langle \psi \rangle$
$\psi_1 \wedge \psi_2$	$\overleftarrow{\psi_1 \wedge \psi_2} = \langle \psi_1 \rangle \& \langle \psi_2 \rangle$
$\psi_1 \vee \psi_2$	$\overleftarrow{\psi_1 \vee \psi_2} = \langle \psi_1 \rangle \mid \langle \psi_2 \rangle$

$ SBVTempConstraints _k$	
	bit-vector encoding
ϕ	$\overleftarrow{\mathbf{Y} \psi} = \ll \langle \psi \rangle$
$\psi_1 \mathbf{S} \psi_2$	$(\overleftarrow{\psi_1 \mathbf{S} \psi_2}^{[k+1:1]} = (\overleftarrow{\psi_2}^{[k+1:1]} \mid \langle \psi_1 \rangle^{[k+1:1]} \& \langle \psi_1 \mathbf{S} \psi_2 \rangle^{[k:0]})) \wedge (\overleftarrow{\psi_1 \mathbf{S} \psi_2}^{[0]} = \langle \psi_2 \rangle^{[0]})$
$\mathbf{X} \psi$	$\overleftarrow{\mathbf{X} \psi}^{[k:0]} = \langle \psi \rangle^{[k+1:1]}$
$\psi_1 \mathbf{U} \psi_2$	$(\overleftarrow{\psi_1 \mathbf{U} \psi_2}^{[k:0]} = \langle \psi_2 \rangle^{[k:0]} \mid \langle \psi_1 \rangle^{[k:0]} \& \langle \psi_1 \mathbf{U} \psi_2 \rangle^{[k+1:1]}) \wedge (((\langle \psi_1 \rangle^{[k+1]} \mid \langle \psi_2 \rangle^{[k+1]} \mid !\langle \psi_1 \mathbf{U} \psi_2 \rangle^{[k+1]}) \& (\overleftarrow{\psi_2}^{[k+1]} \mid \langle \psi_1 \mathbf{U} \psi_2 \rangle^{[k+1]})) = 1) \wedge (\langle \psi_1 \mathbf{U} \psi_2 \rangle^{[k+1]} \Rightarrow \uparrow (\langle \psi_2 \rangle \& \langle inloop \rangle) = 1)$

thus all bit-vectors have length 6 ($k+2$). This trace comes from a counterexample that shows P_1 does not hold for S in the running example. P_1 states that, for all executions of the system, at some point *in* stops occurring. This property can be trivially falsified by the shown counterexample, in which *in* occurs infinitely often, to be precise, every other time instant from time instant 3. The first two rows are the actual trace, and the rest shows how bit-vectors represent their corresponding subformulae. $\langle lpos \rangle$ equal to 000011 means that the solver was able to find a loop at position 3. Consequently, $\langle inloop \rangle$ is 111000, that corresponds to 111111 shifted to the left 3 (*lpos*) times. The table shows that in all bit-vectors that represent a subformula, the bit at position 3 (loop position, *lpos*) is equal to the one at position 5 ($k+1$), because of the last state constraint.

As mentioned in Section 2.2, constraints $|LoopConstraints|_k$, which impose the equality of states s_{i-1} and s_k , are introduced for optimization purposes, but they do not affect the correctness of the encoding. Since in our new encoding we assessed empirically they do not have beneficial effects on the efficiency of the verification, we did not use them, and $|SBVLoopConstraints|_k$ reduce to the definition of bit-vector $\langle inloop \rangle$.

For every subformula ϕ being replaced by a fresh bit-vector, Table 3 introduces the sets of constraints in bv logic that define the value of ϕ . $|SBVPropConstraints|_k$ assume that the main connective in ϕ is a Boolean one. $|SBVTempConstraints|_k$ capture the semantics of temporal operators.

Yesterday. Given the semantics of formula $\mathbf{Y} \psi$, where $\mathbf{Y} \psi$ holds at i iff ψ holds at $i-1$, the bit-vector for $\mathbf{Y} \psi$ is the one for ψ , but shifted “to the left” (from $i-1$ to i , recall that position 0 in bit-vectors is the rightmost one). Consistent with the origin semantics of $\mathbf{Y} \psi$, the rightmost bit of $\ll \langle \psi \rangle$ is 0.

Since. The encoding of \mathbf{S} is recursively defined based on the fact that $\psi_1 \mathbf{S} \psi_2$ holds in i iff either ψ_2 holds in i or ψ_1 holds in i and $\psi_1 \mathbf{S} \psi_2$ holds in $i-1$. This recursive definition can be captured by $\bigwedge_{i=1}^k (\langle \phi \rangle^{[i]} \Leftrightarrow (\overleftarrow{\psi_2}^{[i]} \vee \langle \psi_1 \rangle^{[i]} \wedge \langle \phi \rangle^{[i-1]}))$, that is equivalent to $\overleftarrow{\langle \phi \rangle}^{[k+1:1]} = (\overleftarrow{\langle \psi_2 \rangle}^{[k+1:1]} \mid \overleftarrow{\langle \psi_1 \rangle}^{[k+1:1]} \& \overleftarrow{\langle \phi \rangle}^{[k:0]})$.

Along with this constraint, $\overleftarrow{\langle \phi \rangle}^{[0]} = \overleftarrow{\langle \psi_2 \rangle}^{[0]}$ is asserted to make the encoding compliant with the origin semantics of $\psi_1 \mathbf{S} \psi_2$.

Next, the encoding of formula $\mathbf{X}\psi$ is a bit-wise shift to the right of bit-vector $\langle \psi \rangle$, i.e., $\mathbf{X}\psi$ holds at i iff ψ holds at $i+1$. The constraint that bit $\overleftarrow{\langle \phi \rangle}^{[k+1]}$ must be equal to the one at the loop-back position is asserted in the “last state constraints” that are presented later in this section.

Until. Similar to \mathbf{S} , the encoding of \mathbf{U} is also defined recursively. $\psi_1 \mathbf{U} \psi_2$ holds in i iff either ψ_2 holds in i or ψ_1 holds in i and $\psi_1 \mathbf{U} \psi_2$ holds in $i+1$. This recursive definition can be captured by $\bigwedge_{i=1}^k (\overleftarrow{\langle \phi \rangle}^{[i]} \Leftrightarrow (\overleftarrow{\langle \psi_2 \rangle}^{[i]} \vee \overleftarrow{\langle \psi_1 \rangle}^{[i]} \wedge \overleftarrow{\langle \phi \rangle}^{[i+1]})$, that is equivalent to $\overleftarrow{\langle \phi \rangle}^{[k:0]} = (\overleftarrow{\langle \psi_2 \rangle}^{[k:0]} \mid \overleftarrow{\langle \psi_1 \rangle}^{[k:0]} \& \overleftarrow{\langle \phi \rangle}^{[k+1:1]})$.

Based on the recursive definition of \mathbf{U} at position $k+1$, two constraints should hold. First, if $\overleftarrow{\langle \psi_1 \mathbf{U} \psi_2 \rangle}^{[k+1]}$ holds, then either $\overleftarrow{\langle \psi_1 \rangle}^{[k+1]}$ or $\overleftarrow{\langle \psi_2 \rangle}^{[k+1]}$ hold; this constraint, which in the following we indicate as *Constraint₁*, can be represented in bv logic as $(!\overleftarrow{\langle \psi_1 \mathbf{U} \psi_2 \rangle}^{[k+1]} \mid (\overleftarrow{\langle \psi_1 \rangle}^{[k+1]} \mid \overleftarrow{\langle \psi_2 \rangle}^{[k+1]})) = 1$. Second, if $\overleftarrow{\langle \psi_2 \rangle}^{[k+1]}$ holds, then $\overleftarrow{\langle \psi_1 \mathbf{U} \psi_2 \rangle}^{[k+1]}$ also holds, i.e., $\overleftarrow{\langle \psi_2 \rangle}^{[k+1]} \Rightarrow \overleftarrow{\langle \psi_1 \mathbf{U} \psi_2 \rangle}^{[k+1]}$ holds. Therefore, a bv logic representation of this constraint (which we indicate in the following as *Constraint₂*) can be $(!\overleftarrow{\langle \psi_2 \rangle}^{[k+1]} \mid (\overleftarrow{\langle \psi_1 \mathbf{U} \psi_2 \rangle}^{[k+1]})) = 1$.

The second and third lines of the encoding are essentially a conjunction of *Constraint₁* and *Constraint₂* expressed in bv logic.

If no additional constraints are imposed on the semantics of operator \mathbf{U} , $\langle \phi \rangle$ can be true throughout the periodic part (i.e., $s\beta$ in $\alpha s\beta s$) without any position within it in which $\langle \psi_2 \rangle$ is true. For example, if we suppose that $k=2$, $\langle lpos \rangle = 0001$, $\langle inloop \rangle = 1110$, $\langle \psi \rangle_2 = 0001$, and $\langle \psi \rangle_1 = 1111$. According to the previous constraint (and the “last state constraint” introduced below), $\langle \phi \rangle = \psi_1 \mathbf{U} \psi_2$ can be either 0001 or 1111, but the latter value is not correct. In the classic encoding, this is fixed through the introduction of constraints $|Eventualities|_k$ (see Section 2.2). To avoid this problem, we add a constraint that asserts that $\langle \phi \rangle$ is true only if there is at least one position in the periodic part where ψ_2 is true, that is, ψ_2 holds infinitely often. More precisely, we add constraint $\overleftarrow{\langle \phi \rangle}^{[k+1]} \Rightarrow \uparrow(\overleftarrow{\langle \psi \rangle}_2 \& \overleftarrow{\langle inloop \rangle}) = 1$ to the encoding of operator \mathbf{U} . Consequently, incorrect values are ruled out, and in fact in the previous example $\langle \phi \rangle$ cannot be 1111, since $\uparrow(0001 \& 1110) = 0$.

The “last state constraints” ($|SBVLastStateConstraints|_k$), which must be added for all subformulae ψ of ϕ (including propositional letters), state that $\overleftarrow{\langle \psi \rangle}^{[lpos]} = \overleftarrow{\langle \psi \rangle}^{[k+1]}$.

Then, given an LTL formula ϕ , the complete bit-vector-based encoding, called ϕ_{sbv} , is given by:

- I $|SBVLastStateConstraints|_k$;
- II $|SBVLoopConstraints|_k$ to capture the definition of $\langle inloop \rangle$;
- III The constraints that define each subformula ($|SBVPropConstraints|_k$ and $|SBVTempConstraints|_k$);
- IV Constraint $\overleftarrow{\langle \phi \rangle}^{[0]} = 1$, where $\overleftarrow{\langle \phi \rangle}$ is the bit-vector defined based on its subformulae.

For example, if we consider formula $\neg \mathbf{X}p \vee (q \mathbf{U} \mathbf{Y}p)$, its complete encoding $(\neg \mathbf{X}p \vee (q \mathbf{U} \mathbf{Y}p))_{sbv}$ is given by the following formula:

I	$\overleftarrow{\langle p \rangle}^{[lpos]} = \overleftarrow{\langle p \rangle}^{[k+1]} \wedge \overleftarrow{\langle q \rangle}^{[lpos]} = \overleftarrow{\langle q \rangle}^{[k+1]} \wedge \overleftarrow{\langle \mathbf{Y}p \rangle}^{[lpos]} = \overleftarrow{\langle \mathbf{Y}p \rangle}^{[k+1]} \wedge \overleftarrow{\langle \mathbf{X}p \rangle}^{[lpos]} = \overleftarrow{\langle \mathbf{X}p \rangle}^{[k+1]} \wedge \overleftarrow{\langle q \mathbf{U} \mathbf{Y}p \rangle}^{[lpos]} = \overleftarrow{\langle q \mathbf{U} \mathbf{Y}p \rangle}^{[k+1]} \wedge$
II	$\overleftarrow{\langle inloop \rangle} = \ll\overleftarrow{lpos} - 1 \wedge$
III	$(\overleftarrow{\langle \mathbf{Y}p \rangle} = \ll\overleftarrow{\langle p \rangle}) \wedge (\overleftarrow{\langle q \mathbf{U} \mathbf{Y}p \rangle}^{[k:0]} = \overleftarrow{\langle \mathbf{Y}p \rangle}^{[k:0]} \mid (\overleftarrow{\langle q \rangle}^{[k:0]} \& \overleftarrow{\langle q \mathbf{U} \mathbf{Y}p \rangle}^{[k+1:1]})) \wedge (\overleftarrow{\langle q \mathbf{U} \mathbf{Y}p \rangle}^{[k+1]} \Rightarrow \uparrow(\overleftarrow{\langle \mathbf{Y}p \rangle} \& \overleftarrow{\langle inloop \rangle}) = 1) \wedge (\overleftarrow{\langle \mathbf{X}p \rangle}^{[k:0]} = \overleftarrow{\langle p \rangle}^{[k+1:1]}) \wedge ((\overleftarrow{\langle \neg \mathbf{X}p \rangle} = !\overleftarrow{\langle \mathbf{X}p \rangle})) \wedge (\overleftarrow{\langle \neg \mathbf{X}p \vee (q \mathbf{U} \mathbf{Y}p) \rangle} = \overleftarrow{\langle \neg \mathbf{X}p \rangle} \mid \overleftarrow{\langle q \mathbf{U} \mathbf{Y}p \rangle}) \wedge$
IV	$\overleftarrow{\langle \neg \mathbf{X}p \vee (q \mathbf{U} \mathbf{Y}p) \rangle}^{[0]} = 1$

Similar to the classic Boolean encoding, the semantics of the other temporal operators is defined from the basic ones as abbreviations. In fact, based on our experiments, in the case of *sbvzot*, introducing direct encodings for the derived temporal operators—as done in *bvzot*—does not impact on the efficiency of the encoding, therefore we simply define the following: $\mathbf{F}\phi = \top \mathbf{U}\phi$, $\mathbf{G}\phi = \neg \mathbf{F}\neg\phi$, $\phi_1 \mathbf{R}\phi_2 = \neg(\neg\phi_1 \mathbf{U} \neg\phi_2)$, $\mathbf{P}\phi = \top \mathbf{S}\phi$, $\mathbf{H}\phi = \neg \mathbf{P}\neg\phi$, and $\phi_1 \mathbf{T}\phi_2 = \neg(\neg\phi_1 \mathbf{S} \neg\phi_2)$.

As for *bvzot*, we also add constraint $\langle \phi \rangle = \ll\overleftarrow{\langle \psi \rangle} \mid 1$ to capture the semantics of $\phi = \mathbf{Z}\psi$, in order to support PNF formulae (see Section 2.2).

3.1.1 Correctness and Complexity

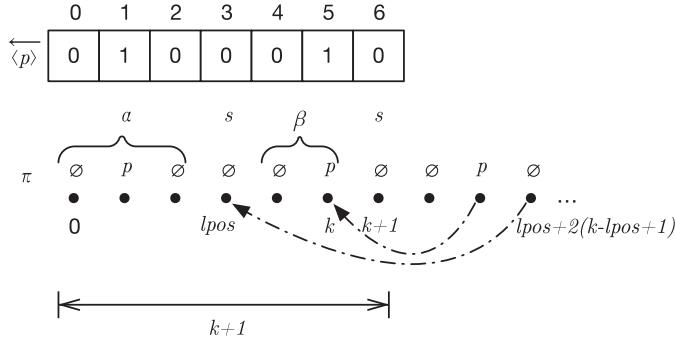
We show the correctness of the encoding by proving a pair of results. First, we show that, when the encoding of a formula ϕ is satisfiable, the original formula is also satisfiable (*soundness* of the encoding); then, we prove that, if an ultimately periodic model of ϕ exists, then the encoding is satisfiable, provided that a sufficiently long bound k has been defined (which shows, to a certain extent, the *completeness* of the encoding).

To help the reader follow the proofs presented in this section, we exemplify some relevant cases through pictures showing some example bit-vectors and corresponding LTL models.

Theorem 1. Let ϕ be an LTL formula, and let $k \in \mathbb{N}$ be the bound for the encoding ϕ_{sbv} . If formula ϕ_{sbv} is satisfiable, then there is a model $\pi = \alpha s(\beta s)^\omega$ of ϕ such that $k+1 = |\alpha s\beta|$.

Proof. To show the result, we first define how α , s and β are defined from the bit-vectors satisfying ϕ_{sbv} , and then we show that $\pi \models \phi$ holds.

Fig. 1 provides a graphical depiction of the correspondence between bit-vectors related to atomic propositions and words. Notice that, in all figures shown in this section, bit-vectors are depicted with the least significant bit on the left, instead of on the right, to facilitate the correspondence with words. Recall that $lpos$ is the loop-back position in π (where the first position in the bit-vector is 0), so we define $|\alpha| = lpos$ and $|\beta| = k - lpos$, and the length of the loop is $k - lpos + 1$. Word $\pi : \mathbb{N} \rightarrow 2^{AP}$ is defined in the following way: (i) for all $i \in \mathbb{N}$ such that $i \leq k$ holds, then $p \in \pi(i)$ (where $p \in AP$) if, and only if, $\overleftarrow{\langle p \rangle}^{[i]} = 1$ holds; (ii) for all i such that $i > k$, then $p \in \pi(i)$ holds if, and only if, $p \in \pi(j)$ also holds, where j is the unique value such that $lpos \leq j \leq k$ holds and there exists $m \in \mathbb{N}$ such that $i = j + m(k - lpos + 1)$ holds.

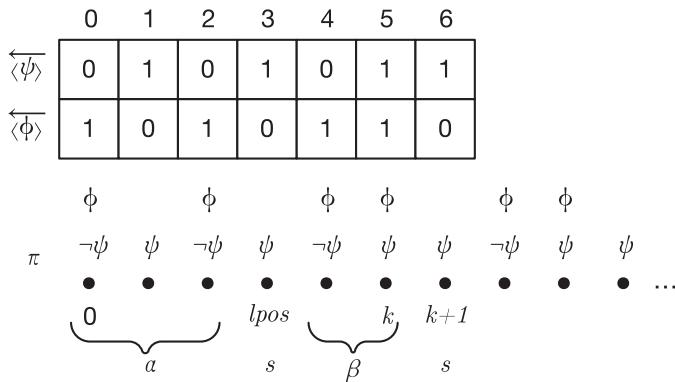
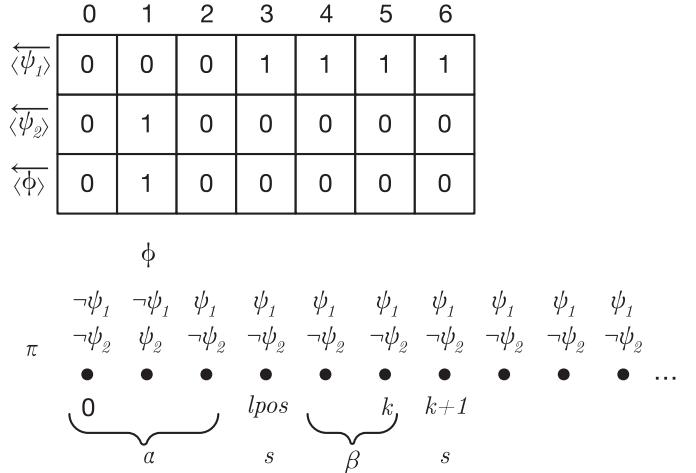
Fig. 1. Example of model π built from bit-vector $\leftarrow\langle p \rangle$.

To show that $\pi \models \phi$ holds we prove, by induction on the structure of formula ϕ , that: (i) for all $i \in \mathbb{N}$ such that $i \leq k$ holds, then $\pi, i \models \phi$ holds if, and only if, $\overleftarrow{\langle \phi \rangle}^{[i]} = 1$ holds; and (ii) for all i such that $i > k$, $\pi, i \models \phi$ holds if, and only if, $\overleftarrow{\langle \phi \rangle}^{[j]} = 1$ also holds, where, as above, j is the unique value $lpos \leq j \leq k$ such that there is $m \in \mathbb{N}$ such that $i = j + m(k - lpos + 1)$ holds.

The base case $\phi = p$, with $p \in AP$, is trivial from the definition of π .

If $\phi = \neg\psi$, by definition we have that, for all $i \leq k$, $\pi, i \models \phi$ holds if, and only if $\pi, i \not\models \psi$, which, by induction, holds if, and only if, $\overleftarrow{\langle \psi \rangle}^{[i]} = 0$; by the definitions of Table 3, this occurs if, and only if, $\overleftarrow{\langle \phi \rangle}^{[i]} = 1$ holds. The cases for $i > k$ and for the propositional connectives \wedge and \vee are similar.

If $\phi = X\psi$, then $\pi, i \models \phi$ if, and only if, $\pi, i + 1 \models \psi$. Fig. 2 exemplifies this case. If $i < k$ (say, $i = 2$ in Fig. 2), then by induction hypothesis $\overleftarrow{\langle \psi \rangle}^{[i+1]} = 1$ holds and, by the definitions of Table 3, $\overleftarrow{\langle \psi \rangle}^{[i+1]} = \overleftarrow{\langle \phi \rangle}^{[i]} = 1$ holds. If $i = k$, then $i + 1 = k + 1 = lpos + (k - lpos + 1)$ (that is, $j = lpos$ and $m = 1$); then, by induction hypothesis, $\overleftarrow{\langle \psi \rangle}^{[lpos]} = 1$ holds and, by constraints $|SBVLastStateConstraints|_k$ and Table 3, $\overleftarrow{\langle \phi \rangle}^{[k]} = \overleftarrow{\langle \psi \rangle}^{[k+1]} = \overleftarrow{\langle \psi \rangle}^{[lpos]} = 1$. If $i > k$, we separate the case where $i \neq k + m(k - lpos + 1)$ (e.g., $i = 7$ in Fig. 2, where $k = 5$ and $k - lpos + 1 = 3$) from the one where $i = k + m(k - lpos + 1)$ (e.g., $i = 8$ in Fig. 2), which are shown in a similar manner as cases $i < k$ and $i = k$ above.

Fig. 2. Exemplification of case $\phi = X\psi$.Fig. 3. Exemplification of case $\phi = \psi_1 U \psi_2$ when $\overleftarrow{\langle \phi \rangle}^{[k+1]} = 0$ holds.

If $\phi = Y\psi$, $\pi, i \models \phi$ holds if, and only if, $i > 0$ and $\pi, i - 1 \models \psi$. If $i = 0$, then by definition $\pi, 0 \not\models \phi$; by Table 3, $\overleftarrow{\langle \phi \rangle}^{[0]} = 0$ (recall that the bit of index 0 is the right-most one, and the unsigned left shift operation \ll inserts a 0 to the right), which shows the desired result. If $0 < i \leq k$ holds, then by induction hypothesis $\overleftarrow{\langle \psi \rangle}^{[i-1]} = 1$ holds and, by the definitions of Table 3, $\overleftarrow{\langle \psi \rangle}^{[i]} = \overleftarrow{\langle \phi \rangle}^{[i]} = 1$ holds. If $i > k$, we separate the cases $i = lpos + m(k - lpos + 1)$ and $i \neq lpos + m(k - lpos + 1)$. The latter is shown in a similar manner as case $0 < i \leq k$ above. If $i = lpos + m(k - lpos + 1)$, then $i - 1 = k + (m - 1)(k - lpos + 1)$, so, by induction hypothesis, $\overleftarrow{\langle \psi \rangle}^{[i-1]} = 1$ holds; then, by constraints $|SBVLastStateConstraints|_k$ and Table 3, $\overleftarrow{\langle \phi \rangle}^{[i]} = \overleftarrow{\langle \phi \rangle}^{[k]} = \overleftarrow{\langle \psi \rangle}^{[k]} = 1$ holds.

If $\phi = \psi_1 U \psi_2$, then $\pi, i \models \phi$ holds if, and only if, either $\pi, i \models \psi_2$ holds, or both $\pi, i \models \psi_1$ and $\pi, i + 1 \models \psi_1 U \psi_2$ hold. This case is exemplified in Fig. 3. Consider the case $i \leq k$. If $\pi, i \models \psi_2$ holds (in which case $\pi, i \models \phi$ also holds, as for $i = 1$ in Fig. 3), by induction hypothesis $\overleftarrow{\langle \psi_2 \rangle}^{[i]} = 1$ holds and, by the definitions of Table 3, $\overleftarrow{\langle \phi \rangle}^{[i]} = 1$ also holds. Otherwise, if $\pi, i \models \psi_1$ does not hold (in which case $\pi, i \models \phi$ does not hold, as for $i = 2$ in Fig. 3), by induction hypothesis $\overleftarrow{\langle \psi_1 \rangle}^{[i]} = 0$ holds and, by Table 3, $\overleftarrow{\langle \phi \rangle}^{[i]} = 0$ holds. If, instead, $\pi, i \models \psi_1$ holds (and $\pi, i \models \psi_2$ does not hold), then $\pi, i \models \phi$ holds if, and only if, $\pi, i + 1 \models \psi_1 U \psi_2$ holds; in addition, in this case, by Table 3 we have that $\overleftarrow{\langle \phi \rangle}^{[i]} = \overleftarrow{\langle \phi \rangle}^{[i+1]}$ holds. We separate two cases: $i < k$ and $i = k$. If $i < k$ (e.g., in position $i = 3$ in Fig. 3), the previous considerations apply also at position $i + 1$, and we iterate them (notice that $\pi, i' \not\models \psi_2$, $\pi, i' \models \psi_1$, $\overleftarrow{\langle \psi_2 \rangle}^{[i']} = 0$, $\overleftarrow{\langle \psi_1 \rangle}^{[i']} = 1$ and $\overleftarrow{\langle \phi \rangle}^{[i']} = \overleftarrow{\langle \phi \rangle}^{[i'+1]}$ all hold for all positions $i \leq i' < k$ in which we iterate the reasoning). If $i = k$, we have that $\pi, k \models \phi$ holds if, and only if, $\pi, k + 1 \models \psi_1 U \psi_2$ holds; also, $\overleftarrow{\langle \phi \rangle}^{[k]} = \overleftarrow{\langle \phi \rangle}^{[k+1]}$ holds by Table 3. We show that either $\overleftarrow{\langle \phi \rangle}^{[k+1]} = 0$ and $\pi, k \not\models \phi$ both hold, or $\overleftarrow{\langle \phi \rangle}^{[k+1]} = 1$ and $\pi, k \models \phi$ do.

- If $\overleftarrow{\langle \phi \rangle}^{[k+1]} = 0$ holds then, by constraints $|SBVLastStateConstraints|_k$, $\overleftarrow{\langle \phi \rangle}^{[lpos]} = 0$ also holds. Then, by Table 3, $\overleftarrow{\langle \psi_2 \rangle}^{[lpos]} = 0$ holds and at least one of $\overleftarrow{\langle \psi_1 \rangle}^{[lpos]}$ and $\overleftarrow{\langle \phi \rangle}^{[lpos+1]}$ is also 0. If $\overleftarrow{\langle \psi_1 \rangle}^{[lpos]} = 0$, then, by inductive hypothesis, $\pi, k+1 \not\models \psi_2$ and $\pi, k+1 \not\models \psi_1$ hold (notice that $k+1 = lpos + (k - lpos + 1)$), hence $\pi, k \not\models \phi$ also holds. If, instead, $\overleftarrow{\langle \psi_1 \rangle}^{[lpos]} = 1$, then $\overleftarrow{\langle \phi \rangle}^{[lpos]} = \overleftarrow{\langle \phi \rangle}^{[lpos+1]} = 0$, and we iterate the reasoning until either there is $lpos < i' \leq k$ such that $\overleftarrow{\langle \psi_1 \rangle}^{[i']}$ is 0, or we conclude that for all $lpos \leq i' \leq k$ both $\overleftarrow{\langle \psi_2 \rangle}^{[i']} = 0$ and $\overleftarrow{\langle \psi_1 \rangle}^{[i']} = 1$ hold (this is the case exemplified in Fig. 3). In both cases, by inductive hypothesis we conclude that $\pi, k \not\models \phi$ holds (notice that if, as in Fig. 3, throughout interval $[lpos, k]$ $\overleftarrow{\langle \psi_2 \rangle}$ is 0 and $\overleftarrow{\langle \psi_1 \rangle}$ is 1, then by inductive hypothesis ψ_1 holds forever after k , but ψ_2 never does, so ϕ does not hold).
- If, instead, $\overleftarrow{\langle \phi \rangle}^{[k+1]} = 1$ holds, then, by Table 3, $\overleftarrow{\langle \psi_1 \rangle}^{[k+1]} = 1$, or $\overleftarrow{\langle \psi_2 \rangle}^{[k+1]} = 1$ hold. In the latter case, by inductive hypothesis, $\pi, k+1 \models \psi_2$ holds, so $\pi, k \models \phi$ also holds. In the former case, by constraints $|SBVLastStateConstraints|_k$, both $\overleftarrow{\langle \psi_1 \rangle}^{[lpos]} = 1$ and $\overleftarrow{\langle \phi \rangle}^{[lpos]} = 1$ hold. By the constraints of Table 3, $\overleftarrow{\langle \psi_2 \rangle}^{[lpos]} = 1$ or $\overleftarrow{\langle \psi_1 \rangle}^{[lpos]} \& \overleftarrow{\langle \phi \rangle}^{[lpos+1]} = 1$ hold. The case $\overleftarrow{\langle \psi_2 \rangle}^{[lpos]} = 1$ (which is the same as $\overleftarrow{\langle \psi_2 \rangle}^{[k+1]} = 1$) was handled previously. If $\overleftarrow{\langle \psi_1 \rangle}^{[lpos]} \& \overleftarrow{\langle \phi \rangle}^{[lpos+1]} = 1$ holds, then we iterate the reasoning. By constraint $(\overleftarrow{\langle \psi_1 \rangle}^{[k+1]} \Rightarrow \uparrow(\overleftarrow{\langle \psi_2 \rangle} \& \overleftarrow{\langle \text{inloop} \rangle}) = 1)$ of Table 3, there must be an index $lpos \leq i' \leq k$ such that $\overleftarrow{\langle \psi_2 \rangle}^{[i']} = 1$ holds. Then, by inductive hypothesis $\pi, i' \models \psi_2$ and $\pi, i' + (k - lpos + 1) \models \psi_2$ hold (and $\pi, j \models \psi_1$ for all $k \leq j \leq i' + (k - lpos + 1)$), so $\pi, k \models \phi$ also holds.

Case $i > k$, with $i = j + m(k - lpos + 1)$ is similar to the previous one, when one considers index j (for which $lpos \leq j \leq k$ holds) in place of i .

If $\phi = \psi_1 S \psi_2$, then $\pi, i \models \phi$ holds if, and only if, either $\pi, i \models \psi_2$ holds, or both $\pi, i \models \psi_1$ and $\pi, i - 1 \models \psi_1 S \psi_2$ hold, provided that $i > 0$ holds. Notice that $\pi, 0 \models \phi$ holds if, and only if, $\pi, 0 \models \psi_2$ also holds. The proof for the case $i \leq k$ is similar to the one for subformula $\psi_1 U \psi_2$, with the simplification given by the fact that, at position 0, the truth of $\psi_1 S \psi_2$ is the same as that of ψ_2 . The proof for the case $i > k$, with $i = j + m(k - lpos + 1)$, is similar to the case $i \leq k$, using $lpos \leq j \leq k$ instead of i . One only needs to consider that, if $\overleftarrow{\langle \phi \rangle}^{[lpos]} = 1$ holds (which, by constraints $|SBVLastStateConstraints|_k$, entails that $\overleftarrow{\langle \phi \rangle}^{[k+1]} = 1$ also holds), and if $\overleftarrow{\langle \psi_1 \rangle}^{[i']} = 1$ and $\overleftarrow{\langle \psi_2 \rangle}^{[i']} = 0$ hold for all $lpos \leq i' \leq k$ then, by inductive hypothesis, $\pi, t \models \psi_1$ and $\pi, t \not\models \psi_2$ hold for all $lpos \leq t \leq i$. However, since $\overleftarrow{\langle \phi \rangle}^{[lpos]} = 1$ holds, using a similar reasoning as in

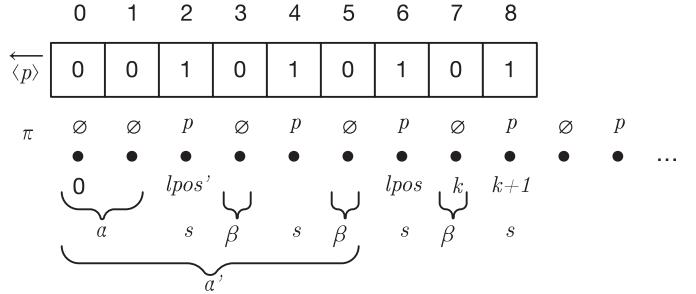


Fig. 4. Example of bit-vector \overleftarrow{p} built from word π in a case where the depth δ is 2.

the case of subformula $\psi_1 U \psi_2$, one can show that there must be a position $0 \leq j' < lpos$ such that $\overleftarrow{\langle \psi_2 \rangle}^{[j']} = 1$ holds, and for all $j' < t' < lpos$ also $\overleftarrow{\langle \psi_1 \rangle}^{[t']} = 1$ holds. Then, by inductive hypothesis, $\pi, t' \models \psi_1$ holds for all $j' < t' < lpos$, $\pi, j' \models \psi_2$ holds, and $\pi, i \models \phi$ finally holds.

Finally, from the fact that $\overleftarrow{\langle \phi \rangle}^{[0]} = 1$, we have that $\pi, 0 \models \phi$ holds, that is, formula ϕ is satisfiable. \square

In the following result, given a formula ϕ we indicate by $\delta(\phi)$ the nesting depth of past operators \mathbf{Y} and \mathbf{S} . More precisely, if $\phi = p$ (with $p \in AP$), then $\delta(\phi) = 0$; if $\phi = \neg(\psi)$ or $\phi = \mathbf{X}\psi$, then $\delta(\phi) = \delta(\psi)$; if $\phi = \psi_1 \wedge \psi_2$, $\phi = \psi_1 \vee \psi_2$, or $\phi = \psi_1 \mathbf{U} \psi_2$, then $\delta(\phi) = \max(\delta(\psi_1), \delta(\psi_2))$; if $\phi = \mathbf{Y}\psi$, then $\delta(\phi) = \delta(\psi) + 1$; finally, if $\phi = \psi_1 \mathbf{S} \psi_2$, then $\delta(\phi) = \max(\delta(\psi_1), \delta(\psi_2)) + 1$. For example $\delta(\mathbf{YY}p) = 2$. We have the following result.

Theorem 2. Let ϕ be an LTL formula, whose depth of past operators is $\delta(\phi)$. Let $\pi = \alpha s (\beta s)^\omega$ be a model of ϕ and $k+1 = |\alpha(s\beta)^{\delta(\phi)+1}|$; then, ϕ_{sbv} is satisfiable, with bound for the encoding ϕ_{sbv} equal to k .

Before proving the result let us remark that, in this case, we are considering a bound k that is long enough to encode a sufficient number of iterations of the loop $s\beta$ (as evidenced by the condition $k+1 = |\alpha(s\beta)^{\delta(\phi)+1}|$). This is due to the presence of past temporal operators \mathbf{Y} and \mathbf{S} , which entail that $\delta(\phi) > 0$ holds; for a formula ϕ that does not include past temporal operators (for which $\delta(\phi) = 0$ holds), the result could be proved with simply $k+1 = |\alpha s\beta|$. For example, consider formula $\bar{\phi} = \mathbf{GFYY}p$ whose depth is $\delta(\bar{\phi}) = 2$. Word $\pi = p^\omega$ is a model for $\bar{\phi}$, but we need to encode at least 3 iterations of the loop to make $\bar{\phi}_{\text{sbv}}$ satisfiable.

Proof. To prove the result, we first define the values of the bit-vectors that appear in formula ϕ_{sbv} , and then we show that they satisfy the formulae of the encoding. More precisely, for every subformula ψ of ϕ , for every position $0 \leq i \leq k+1$, we define that $\overleftarrow{\langle \psi \rangle}^{[i]} = 1$ if, and only if, $\pi, i \models \psi$. Notice that, since we are requiring that $k+1 = |\alpha(s\beta)^{\delta(\phi)+1}|$ holds, we are essentially considering model π to be $\pi = \alpha' s (\beta s)^\omega$, where $\alpha' = \alpha(s\beta)^{\delta(\phi)}$. Hence, we define $lpos = |\alpha'|$ (i.e., bit-vector $\overleftarrow{\langle lpos \rangle}$ is the binary encoding, over $k+2$ bits, of value $|\alpha'|$), so that position $lpos$ corresponds to the start of the $\delta(\phi) + 1$ th iteration of the loop in π . Finally, we define $\overleftarrow{\langle \text{inloop} \rangle}^{[i]} = 1$ if, and only if, $i \geq lpos$. Fig. 4 shows an example of bit-vector and

parameters $lpos$, k defined from a word $\pi = \alpha s(\beta s)^\omega$, in the case where subformula ψ is a propositional letter and the depth is 2. Notice that, in the shown example, word π is a model for formula $\bar{\phi} = \mathbf{GFY}\psi$.

First of all, constraints $|SBVLoopConstraints|_k$ trivially hold by construction. Similarly for constraint $\overleftarrow{[\cdot]}^{[0]} = 1$, since by definition $\pi, 0 \models \phi$ holds.

The constraints of Table 3 ($|SBVPropConstraints|_k$) also obviously hold. Consider, for example, a subformula $\psi = \neg\psi'$. By definition, $\pi, i \models \psi$ holds if, and only if, $\pi, i \models \psi'$ does not hold. By construction, then, for all $0 \leq i \leq k+1$, $\overleftarrow{[\psi]}^{[i]} = 1$ holds if, and only if, $\overleftarrow{[\psi']}^{[i]} = 0$.

Consider now the constraints $|SBVTempConstraints|_k$ of Table 3. It is easy to see that, if $\psi = X\psi'$ holds, constraint $\overleftarrow{[\psi]}^{[k:0]} = \overleftarrow{[\psi']}^{[k+1:1]}$ also holds. In fact, by definition, $\pi, i \models \psi$ holds if, and only if, $\pi, i+1 \models \psi'$ does. Then, by construction, for all $0 \leq i \leq k$, $\overleftarrow{[\psi]}^{[i]} = 1$ holds if, and only if, $\overleftarrow{[\psi']}^{[i+1]} = 1$ holds. Similarly if $\psi = Y\psi'$; in this case, by definition $\pi, 0 \not\models \psi$ holds, and in fact constraint $\overleftarrow{[\psi]} = \ll \overleftarrow{[\psi']}$ imposes that $\overleftarrow{[\psi]}^{[0]} = 0$ holds due to the \ll operator. The constraints of case $\psi = \psi_1 U \psi_2$ also hold. Indeed, by definition $\pi, i \models \psi$ holds if, and only if, either $\pi, i \models \psi_2$ holds, or both $\pi, i \models \psi_1$ and $\pi, i+1 \models \psi_1 U \psi_2$ hold. By construction, then, constraint $\overleftarrow{[\psi]} = \overleftarrow{[\psi_2]}^{[i]} \mid \overleftarrow{[\psi_1]}^{[i]} \& \overleftarrow{[\psi]}^{[i+1]}$ holds for all $0 \leq i \leq k$. At position $k+1$, either $\pi, k+1 \models \psi$ holds, or $\pi, k+1 \not\models \psi$ holds. If $\pi, k+1 \models \psi$ holds, by construction $\overleftarrow{[\psi]}^{[k+1]} = 1$ holds, which means that $(! \overleftarrow{[\psi_2]}^{[k+1]} \mid \overleftarrow{[\psi]}) = 1$ also holds. In addition, since $\pi, k+1 \models \psi$ holds, either $\pi, k+1 \models \psi_2$ holds, or $\pi, k+1 \models \psi_1$ does, which assures that $(\overleftarrow{[\psi_1]}^{[k+1]} \mid \overleftarrow{[\psi_2]}^{[k+1]} \mid ! \overleftarrow{[\psi]}) = 1$ holds by construction. In addition, since $\pi = \alpha' s(\beta s)^\omega$ and $k+1 = |\alpha' s\beta|$ (so $k+1$ is the position of the second s in $\alpha' s\beta s$), $\pi, i' \models \psi_2$ must hold for some $lpos \leq i' \leq k$, or ψ_2 would never be true throughout suffix $(\beta s)^\omega$, so ψ would not hold at position $k+1$. Then, constraint $\overleftarrow{[\psi]}^{[k+1]} \Rightarrow \uparrow (\overleftarrow{[\psi_2]} \& \overleftarrow{[\text{inloop}]}) = 1$ holds by construction. If $\pi, k+1 \not\models \psi$ holds, $(\overleftarrow{[\psi_1]}^{[k+1]} \mid \overleftarrow{[\psi_2]}^{[k+1]} \mid ! \overleftarrow{[\psi]}) = 1$ holds by construction. In addition, $\pi, k+1 \models \psi_2$ cannot hold, so constraint $(! \overleftarrow{[\psi_2]}^{[k+1]} \mid \overleftarrow{[\psi]}^{[k+1]}) = 1$ holds. The proof for the constraints of case $\psi = \psi_1 S \psi_2$ is similar (notice that $\pi, 0 \models \psi$ holds if, and only if, $\pi, 0 \models \psi_2$ does).

To conclude the proof, we need to show that constraints $|SBVLastStateConstraints|_k$ hold. To this end we first prove—by induction—something stronger. Let us call $lpos'$ the position of the first loop in $\pi = \alpha(s\beta)^\omega$, as depicted in Fig. 4—that is, $lpos' = |\alpha|$ (recall that, instead, by construction $lpos$ is the position of the $\delta(\phi) + 1$ th loop in π ; also, notice that $k - lpos + 1 = |s\beta|$ holds). We show that, for each subformula ψ of ϕ , whose depth of past operators is $\delta(\psi)$, for all position $lpos' + \delta(\psi)(k - lpos + 1) \leq i \leq lpos' + (\delta(\psi) + 1)(k - lpos + 1) - 1$, $\pi, i \models \psi$ holds if, and only if, $\pi, i + m(k - lpos + 1) \models \psi$, for all $m \in \mathbb{N}$. For example, with reference to Fig. 4 (where

$lpos' = 2$, $k - lpos + 1 = 2$), subformula $\mathbf{YY}\psi$, whose depth is 2, holds (resp., does not hold) at position 6 (resp., 7), and at all positions $6 + m2$ (resp., $7 + m2$); similarly, subformula $\mathbf{Y}\psi$, whose depth is instead 1, does not hold (resp., holds) at position 4 (resp., 5), and at all positions $4 + m2$ (resp., $5 + m2$).

The base case $\psi = p$ (with $p \in AP$) is trivial, since by definition $\pi(i) = \pi(i + m(k - lpos + 1))$ for all $i \leq lpos'$. The inductive cases for propositional connectives and for future temporal operators are straightforward. For example, if $\psi = \psi_1 \mathbf{U} \psi_2$, then there is $i' \geq i$ such that $\pi, i' \models \psi_2$ holds, and $\pi, i'' \models \psi_1$ holds for all $i \leq i'' < i'$. By inductive hypothesis, since $\delta(\psi) \geq \delta(\psi_1)$ and $\delta(\psi) \geq \delta(\psi_2)$ hold, this holds if $\pi, i' + m(k - lpos + 1) \models \psi_2$ holds, and $\pi, i'' + m(k - lpos + 1) \models \psi_1$ holds for all $i \leq i'' < i'$, which corresponds to $\pi, i + m(k - lpos + 1) \models \psi$ holding.

If $\psi = \mathbf{Y}\psi'$, then $\pi, i \models \psi$ holds if, and only if, $\pi, i - 1 \models \psi'$ holds. Since $\delta(\psi) > \delta(\psi')$ holds, then $i > lpos' + \delta(\psi')(k - lpos + 1)$ holds so, by inductive hypothesis, $\pi, i - 1 \models \psi'$ holds if, and only if, $\pi, i - 1 + m(k - lpos + 1) \models \psi'$ holds, which in turn corresponds to $\pi, i + m(k - lpos + 1) \models \psi$ holding.

If $\psi = \psi_1 \mathbf{S} \psi_2$, then there is $i' \leq i$ such that $\pi, i' \models \psi_2$ holds, and $\pi, i'' \models \psi_1$ holds for all $i' < i'' \leq i$. If $i' \geq lpos' + (\delta(\psi) - 1)(k - lpos + 1)$ holds then, since both $\delta(\psi) - 1 \geq \delta(\psi_1)$ and $\delta(\psi) - 1 \geq \delta(\psi_2)$ hold, by inductive hypothesis both $\pi, i' + m(k - lpos + 1) \models \psi_2$ and $\pi, i'' \models \psi_1$ hold for all $i' + m(k - lpos + 1) < i'' \leq i + m(k - lpos + 1)$, which entails that $\pi, i + m(k - lpos + 1) \models \psi$ holds. If, instead, $i' < lpos' + (\delta(\psi) - 1)(k - lpos + 1)$ holds, then $\pi, i'' \models \psi_1$ holds for all $lpos' + (\delta(\psi) - 1)(k - lpos + 1) \leq i'' < lpos' + \delta(\psi)(k - lpos + 1)$, which, by inductive hypothesis since $\delta(\psi) > \delta(\psi_1)$ holds, entails that $\pi, i \models \psi_1$ holds for all $i \geq i'$; hence, $\pi, i + m(k - lpos + 1) \models \psi$ holds for all $m \in \mathbb{N}$.

Since, obviously, $\delta(\phi) \geq \delta(\psi)$ for all subformulae ψ of ϕ , and since, by construction, $k + 1 > lpos' + (\delta(\phi) + 1)(k - lpos + 1) - 1$ holds, then, for all subformulae ψ of ϕ , $\pi, k + 1 \models \psi$ holds if, and only if, $\pi, lpos \models \psi$ holds, which by construction entails that $|SBVLastStateConstraints|_k$ hold. \square

Concerning the size of the encoding ϕ_{sbv} , it is easy to see that, since we introduce a bit-vector constraint of constant size for each subformula ψ of ϕ , the total size is $O(n)$, with n the number of subformulae of ϕ —notice that the number n of subformulae of ϕ is, in the worst case, $O(l)$, with l the length of the formula, defined for example as the number of connectives and temporal operators appearing in ϕ (at worst, each subformula appears only once in ϕ).

4 EXPERIMENTAL EVALUATION

This section summarizes how we evaluated the efficiency of the encoding presented in this paper by comparing it against different state-of-the-art tools. Most of the experiments exploit our checker *Zot*, which is an extensible Bounded Model/Satisfiability Checker written in Common Lisp. More precisely, *Zot* is capable of performing bounded satisfiability checking of formulae written both in LTL (with past operators) and in the propositional, discrete-time fragment of the metric temporal logic TRIO [30], which is

equivalent to LTL, but more concise. The user feeds *Zot* with the specification to be checked and selects the plugin and the time bound (i.e., the value of bound k) to be used to perform the verification. *Zot* encodes the received specification in a target logic (e.g., propositional logic, or bv logic) and provides the result to a solver that is capable of handling the target logic. The result obtained by the solver is parsed back and presented to the user in a textual representation.

To assess the new encoding, we selected five benchmark specifications, two from the literature and two from our previous work. We wanted to work with complex specifications to better highlight the strengths and weaknesses of each tool. What follows is a brief presentation of the five case studies, but we refer the reader to cited literature for more details. These studies employ a BSC approach, that is, they use temporal logic to describe both the system under verification and the properties to be checked (Section 2.2).

Kernel Railway Crossing (KRC). This problem is frequently used for comparing real-time notations and tools [31]. A railway crossing system prevents vehicles from crossing the railway while trains are passing through it by controlling a gate. A temporal logic-based version of the KRC problem was developed in [9] for benchmarking purposes. It only considers one track, trains can only move in a direction, and uses an interlocking system. We experimented with two sets of time constants that allow different degrees of non-determinism, denoted as $krc2$ and $krc3$ in our experiments. The level of non-determinism is increased by using bigger time constants—e.g., the time a train takes to go through the railway crossing—that increase the number of possible combinations of events in the system. We also carried out formal verification with two properties of interest: a safety property that says that as long as a train is in the critical region the gate is closed ($P1$); and a utility property that states that the gate must be open when it is safe to do so (i.e., the gate should not be closed when unnecessary), where the notion of “safe” is captured through suitable time constants ($P2$).

Fischer’s Protocol. It is a classic algorithm for granting exclusive access to a resource that is shared among many processes. Fischer’s protocol is a typical benchmark for verification tools capable of dealing with real-time constraints. The version we used is taken from [9]. It includes 4 processes, and the delay that a process waits after sending a request, which is the key parameter in the protocol, is 5 time instants. We then formally verified a safety property that states that it is never the case that two processes are simultaneously in their critical sections ($P1$). We identify the models of this case study through prefix *fischer*.

Ping Application. *Corretto*³ is the toolset we developed to perform formal verification of UML models [5]. *Corretto* takes as input a set of UML diagrams and produces their formal representation through temporal logic formulae. In our tests we used the example diagrams introduced in [5] (a Class Diagram, an Object Diagram, and a Sequence Diagram with various combined fragments), which describe the behavior of an ping application that pings two servers

and then sends queries to the server that responds first. The model comprises a loop, and we performed tests on two versions of the system, called *sdserver12*, and *sdserver13*, where the number of iterations in the loop is 2 and 3, respectively. Property $P1$ states that the search request is always sent to the server that replies earlier.

On Board Radar System. *Corretto* was also used in the EU-funded project MADES for the verification of two example Radar Systems, one on board the airplane and a ground-based one, provided by two industrial partners. In our tests, we used the on board system, and more precisely a component that carries out the delivery of the flight data from the environment to the User Interface (UI) of the pilot. Such a delivery is performed by a number of periodic tasks. The UML model (whose corresponding LTL formalization is identified by prefix *txt4* in our experiments) comprises a Class Diagram with five clocks, five Sequence Diagrams, and five State Machine Diagrams. The model identified by prefix *txt8* is similar, but larger, as it includes four more tasks—hence four more Sequence Diagrams and as many State Machine Diagrams. The different Sequence Diagrams illustrate how the data are read and processed by the different periodic tasks.

Human Robot Collaboration. This model (which is taken from [32]) formalizes the main elements a collaborative robotic system: a robot, a physical working area, a human operator, and a job executed by both the human and the robot. The model also includes definitions of hazardous physical contacts between the human and the robot based on the definitions of a few adopted ISO standards. Whenever the state of the model conforms with one of those definitions, a risk value that belongs to set {0,1,2} is assigned to the relevant hazard based on its attributes to estimate its harmfulness. Then, a risk reduction measure is activated when risk is 1 or 2 in order to reduce it to 0 in an acceptable amount of time. We use prefix *hrc* to identify the models of this case study.

4.1 Efficiency of the Encoding

To evaluate the efficiency of *sbvzot*, we implemented it as new *Zot* plugin and ran a first set of experiments to check the aforementioned benchmark by means of different tools. These first experiments exploit, in addition to *sbvzot*, the *meezot* and *bvzot* *Zot* plugins presented in [9] and [11], respectively: *meezot* implements an optimized encoding of LTL formulae into propositional logic, while *bvzot* implements our first bv logic-based encoding.

We also ran both NuSMV and nuXmv to test their implementations of the classic bounded encoding (*bmc*) [18], the corresponding optimized encoding (*sbmc*) [19], and its incremental version (*sbmcinc*⁴) [33]. We also used nuXmv for five additional, significant verification algorithms that mainly differ in the way they check LTL properties. *coisat* employs an incremental cone of influence reduction [34] to eliminate unrelated variables with respect to a given property. The flags used in the command specify that a SAT engine is used for both verification and trace execution.

4. While running this verification procedure we did not activate the completeness checking option since it often slows the verification down, as shown in [33].

TABLE 4
Time/Memory Comparison Over the Five Benchmarks

Tool Model	sbvzot		bvzot		meezot		S-bmcinc		S-sbmc		S-sbmcin		X-bmcinc		X-sbmc		X-sbmcin		X-coisat		X-coismt		X-klive		X-msatcoi		X-msat					
	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M				
krc2Sat_60	2	130	13	144	25	245	48	988	159	2108	24	906	147	2048	169	2750	17	777	58	108	18	777	178	1740								
krc2P1_60	2	143	40	176	349	498	282	4988	651	3903	158	4525	589	3823	805	7046	367	3226	788	264	392	3226	2541	4225								
krc2P1_90	95	175	649	213	1897	665	MO	2108	5665	482	9533	2160	5449	MO	TO	1118	264	TO	TO													
krc2P2_60	18	145	40	176	549	501	362	4979	739	3937	227	4500	641	3853	906	7194	457	3433			457	3384	2600	4329								
krc2P2_90	704	212	875	215	TO		MO	TO		2924	9757	TO	MO	TO					TO	TO			85	2364	1504	6317						
krc3Sat_60	7	153	35	177	134	506	292	4888	1319	6552	172	4498	1346	6347	1419	9330	85	2364			244	5660	MO	MO								
krc3P1_60	19	166	56	207	629	850			2927	9639			2814	9235	MO	MO					441	5904	MO	MO								
krc3P1_90	102	151	565	261	HE		MO	TO		2953	9369	MO	466	5904	TO	MO																
krc3P2_60	18	169	60	218	566	988																										
krc3P2_90	229	187	392	259	HE		MO	MO																								
fischerSat_30	4	128	12	155	19	182	77	1531	4	153	7	298	24	1467	2	159	7	302	12	406	11	462	4	73	12	462	9	219				
fischerP1_30	2	128	10	156	19	197	88	1656	5	160	4	306	24	1588	2	162	5	313	8	427	8	475	1	58	8	475	3	203				
fischerP1_60	8	140	38	186	145	378			66	557	13	539	MO	34	1076	11	536	43	1006	52	1825	0	58	52	1801	11	389					
fischerP1_90	19	155	86	217	457	554			58	1185	24	768	132	999	126	1924	204	1970	212	1503	721	1537	212	1503	162	1550						
hrcSat_20	14	462	116	855			235	1599	271	1015	265	1955	94	1591	1053	5202	231	4047	415	5015	1432	7390	930	1934	1434	7463	342	3907				
hrcP1_30	144	1110	995	2066					1235	5607	260	921									1801	2536	MO	MO	1646	7856						
hrcP1_60	596	1931	TO							921	7651	MO	MO											30	886	56	929					
hrcP1_90	1832	2809	TO						55	1543	62	1445	1826	9750	26	1189	58	1416	65	1621	30	886			112	2299	77	1527				
txt4Sat_20	5	186	17	303	89	524			194	4650	137	2293			101	3612	128	2204	158	2885	112	2293			MO	MO	289	3440				
txt4P1_30	18	236	85	425	373	1078					435	3993	MO		411	3880	618	6824	MO							618	5769					
txt4P1_60	114	338	623	737	HE					931	5840				52	1859	88	1784	116	2041	68	1271			66	1271	115	1215				
txt4P1_90	325	455	TO							431	7269	210	2844			189	5555	226	2777	245	3658	237	3545			239	3463	143	2080			
txt8Sat_20	9	226	29	416	197	762					665	5125	MO		630	4862	1044	8741	MO							532	4768					
txt8P1_30	26	294	121	561	418	1078					1421	7591				1352	7068	MO								1229	8460					
txt8P1_60	150	460	1868	1015	HE																											
txt8P1_90	528	661	TO																													
sdserver1Sat_50	11	212	87	547	115	603	145	2130	8	367	32	1104	104	1794	4	310	31	1085	32	1156	60	823	848	344	60	823	45	655				
sdserver2P1_60	169	649	1204	1244					572	9444	420	3910				256	6899	396	3816							621	4464					
sdserver2P1_90	407	840	3059	1797							945	5598	MO		883	5320	TO		TO									1806	8160			
sdserver2P1_120	791	1136	TO							1929	7521				1790	7097																
sdserver3Sat_50	16	255	135	558			269	832	1119	6254	17	639	55	1332	819	4984	9	517	50	1324	56	1466	102	1340			108	1340	79	924		
sdserver3P1_60	203	692	1401	1516							551	4414	MO		324	7631	726	4190	TO								829	4968				
sdserver3P1_90	474	924	TO								1216	6442				2232	8388	MO		1128	6070	TO					1682	9014				
sdserver3P1_120	896	1249																	2081	8049												
Solved Instances	100%	79%	50%	14%	50%	85%	17%	58%	88%	52%	52%	52%	32%	52%	73%																	

coismt is the same as *coisat*, but it uses an SMT engine. *klive* performs a K-Liveness algorithm with the *bmc* algorithm. Note that this algorithm also checks the completeness bound. For example, at a given point it may conclude that the LTL formula is UNSAT and there is no need to check for larger bounds. *msatcoi* employs an SMT-based incremental cone of influence. *msat* is an SMT-based incremental *sbmc*.

Note that NuSMV, nuXmv and Zot also support other encodings for LTL/TRIO; we have chosen to show the results for the ones above because further experiments, not reported here for the sake of brevity, shown them to be, on average, the fastest ones for the tools. We also use S and X before the labels identified above to distinguish between NuSMV and nuXmv. To compare the performance of the different algorithms, we built a simple translator to convert specifications written in the Zot input language—such as those used in [9] and [5]—into the SMV language (the input language of NuSMV and nuXmv).

All experiments⁵ were carried out on a Linux desktop machine with a 3.4 GHz Intel Core™ i7-4770 CPU and 16 GB RAM. In all cases we performed two kinds of checks. First, we took the temporal logic formula ϕ_S describing the system, and we simply checked for its satisfiability. This allowed us to determine whether the specification is realizable or not. As a second type of check, we also considered the logic formula ϕ_P that captures the property of interest,

and we fed the verification tool with formula $\phi_S \wedge \neg\phi_P$ to determine whether the property holds for the system or not. We also experimented with different bounds k to analyze how the tools behave when k is increased.

Since NuSMV and nuXmv adopt a Bounded Model Checking approach, we fed them with an empty system model (for which any trace is possible), together with either $\neg\phi_S$ or $\neg(\phi_S \wedge \neg\phi_P)$ as property to check [35]. Indeed, a BMC tool that receives a property ψ to be verified, first builds $\neg\psi$, then looks for a trace that satisfies $\neg\psi$. As a consequence, by feeding it $\neg\phi_S$ (resp., $\neg(\phi_S \wedge \neg\phi_P)$) as a property, the tool looks for a trace satisfying ϕ_S (resp., $\phi_S \wedge \neg\phi_P$), just like our tool does.

Table 4 shows the time (T) in seconds and memory (M) in MBs consumed in each of the experiments we performed.⁶ Column *Model* concatenates the name of the particular model with the verification type (either SAT or property checking) and the maximum bound. For example, the first row (krc2Sat_30) shows time/memory consumption of each tool for the simple satisfiability checking of model krc2 with the maximum bound $k = 30$. The two subsequent rows (krc2P1_60 and krc2P1_90) are the results for the verification of property P1 with maximum bound $k = 60$ and $k = 90$, respectively. The last row (*Solved Instances*) is the percentage of solved verification problems (models) by each tool on the five benchmarks. To help the reader rank the tools at a first glance, cell background colors indicate the best, second best, and third best tools.

5. We used version 2.6 of NuSMV and version 1.1.1 of nuXmv. The SAT and SMT solvers used with Zot were, respectively, MiniSat version 2.2 and Z3 version 4.8. The code for all the experiments is available, along with all Zot plugins, from the Zot repository [13].

6. Interested readers can refer to the Zot repository [13] for the complete and detailed data about the experiments.

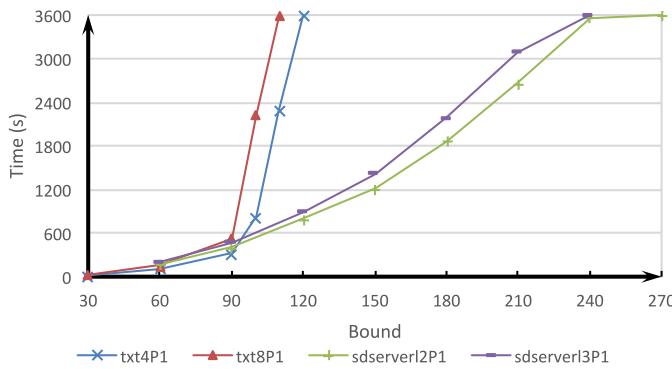


Fig. 5. Excerpts of how *sbvzot* behaves given a one-hour time window.

For each experiment, we set a *maximum bound* k and the tools iteratively (possibly incrementally) tried to find an ultimately periodic model $\alpha\beta^\omega$ where the length of $\alpha\beta$ is $1, 2, \dots, k$. As soon as a model is found, the search stops, and the model is output; if no model is found for any bound up to k , the search stops at k and the formula is declared unsatisfiable.

All the runs reported in Table 4 had a time limit of 1 hour and a memory limit of 10 GB RAM; that is, if the verification has taken longer than 1 hour or occupied more than 10 GB of RAM, it would have been stopped. Hence, the possible outcomes of a run are *satisfiable*, *unsatisfiable*, *out of time* (TO), and *out of memory* (MO). In addition, in some cases the tool stopped because of *heap exhaustion* (HE) while pre-processing the specification to produce the encoding.

Table 4 suggests that the combination of *sbvzot/Z3* is not the fastest for 6 models, but altogether it only needs 53 more seconds to perform the verification of those 6 models. *sbvzot*, however, is the fastest for the remaining 28 models and saves two hours in those experiments.

As Table 4 shows, among the algorithms implemented in NuSMV and nuXmv, *X-sbmciinc* is the one with the highest number of solved instances and mostly the one with the lowest memory consumption, whereas *X-sbmci* is the fastest on average. Indeed, *X-sbmciinc* solved 10 more models than *X-sbmci*; however, if one considers only the models on which both encodings are applicable, *X-sbmci* is usually faster than *X-sbmciinc*. Note that in the case of Fischer’s protocol *X-klive* is the most efficient encoding, but overall it was able to solve only 11 models out of 34 (32 percent). All in all, we can conclude that the experimental results show a promising ability of *sbvzot* to scale as the size of the specification and the time bound increase.

We also carried out some additional experiments with the idea of letting *sbvzot* reach the 3600-second time limit. Fig. 5 shows what happened for *txt4P1*, *txt8P1*, *sdserver12P1*, and *sdserver13P1*. *sbvzot* reached the limit at bounds 241 and 228 for *sdserver12P1* and *sdserver13P1*, respectively, and at bounds 115 and 105 for *txt4P1* and *txt8P1*, respectively. These values witness that the boundaries are very application-specific and give an idea of what the limits of *sbvzot* are.

4.2 Independence of the SMT Solver

One might claim that efficiency of our tool comes mainly from the underlying SMT solver (Z3), rather than from the

encoding itself. To reject this claim, we examined the top five solvers in SMT competitions [21] in recent years, and thus, besides Z3 (version 4.8), we selected four additional SMT solvers. *Boolector* [22] (version 3) supports the quantifier-free theories of fixed-size bit vectors and arrays. This SMT solver won first place in divisions QF_ABV (main and application track), QF_BV (main track) and QF_UFBV (main and application track) in the 2018 SMT competition [36]. *Yices2* [23] (version 2.6) decides the satisfiability of formulae that contain uninterpreted function symbols with equality, real and integer arithmetic, bit vectors, scalar types, and tuples. It also supports nonlinear arithmetic, and has its own specification language (apart from SMT languages). *Mathsat* [15] (version 5.5) supports equality and uninterpreted functions, linear arithmetic, and bit vectors. It also provides additional features like extraction of unsatisfiable cores, generation of models and proofs, and the ability of working incrementally. *CVC4* [24] (version 1.6) is an automatic theorem prover for SMT problems. It supports first-order formulae in a large number of theories and combinations thereof. *CVC4* is intended to be an extensible SMT engine.

Table 5 compares the five implementations of *sbvzot*, that is, based on the five SMT solvers, against the first two best options provided by NuSMV or nuXmv. If no data is reported for NuSMV/nuXmv, these tools were not able to complete the verification process within the given time/memory limit. Again, cell background colors follow the same convention as before to ease the comprehension of the table. When one considers *sbvzot* in general, that is, with any underlying SMT solver, it is, on average, 2 times faster and 8 times more memory-efficient than the best algorithms of NuSMV and nuXmv (column 1st best).

4.3 Lessons Learned

The results above allow us to draw some conclusions on the effectiveness of *sbvzot*, and on the kinds of problems for which it seems particularly well suited.

We noticed a trade-off, at the level of the SMT solver, between the use of bit-blasting, which transforms bit-vector constraints into Boolean constraints, and the simplifications that can be obtained by using bit-vector arithmetic. For example, *bvzot* exploits greater simplifications at the bit-vector level because the encoding heavily depends on arithmetic operators (binary addition in the encoding of U and S). This results in more complex, heavier-to-handle Boolean formulae produced after bit-blasting. *sbvzot* mainly employs bit-wise operators, instead of bit-vector level arithmetic, and the Boolean formulae that are ultimately solved after bit-blasting are easier to handle. Although there are some simplification gains at the bit-vector level, the trade-off seems to favor bit-blasting over arithmetic simplification.

We also want to highlight that when we use MathSAT with our encoding, and NuSMV, that uses MathSAT itself, our use of MathSAT is actually faster. This is another evidence of how the use of bv logic and our encoding help verify (complex) LTL specifications.

5 RELATED WORK

There are essentially two approaches to the problem of satisfiability checking of LTL formulae: bounded and complete

TABLE 5

sbvzot, With the Five SMT Solvers, Against the Two Bests Results Produced by NuSMV/nuXmv on the Five Benchmarks

Tool Model	<i>sbvzot</i>										NuSMV and nuXmv									
	CVC4		Mathsat		Yices2		Boolector		Z3		1st best			2nd best			Tool		Tool	
	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	T	M	X-coismt	X-msatcoi
krc2Sat_60	6	139	4	155	3	131	2	114	2	130	17	777	X-coismt	18	777	X-msatcoi				
krc2P1_60	22	166	21	219	75	159	14	132	2	143	158	4525	X-sbmc	282	4988	S-sbmc				
krc2P1_90	109	238	98	311	506	194	71	150	95	175	482	9533	X-sbmc	1118	264	X-klive				
krc2P2_60	24	163	36	224	119	164	26	135	18	175	227	4500	X-sbmc	362	4979	S-sbmc				
krc2P2_90	579	761	657	349	TO	383	187	704	212	2924	9757	X-sbmc	-----	-----	-----	-----	-----	-----	-----	-----
krc3Sat_60	16	169	13	230	9	172	6	136	7	153	85	2364	X-coismt	85	2364	X-msatcoi				
krc3P1_60	31	187	35	311	77	201	17	150	19	166	244	5660	X-msatcoi	265	5660	X-coismt				
krc3P1_90	140	265	151	425	800	212	95	138	102	151	-----	-----	X-msat	-----	-----	-----	-----	-----	-----	-----
krc3P2_60	34	194	46	312	72	207	27	158	18	169	441	5904	X-msatcoi	466	5904	X-coismt				
krc3P2_90	263	454	460	438	2452	231	268	158	229	187	-----	-----	X-msat	-----	-----	-----	-----	-----	-----	-----
fischerSat_30	5	140	3	136	2	123	3	117	4	128	2	159	X-sbmc	4	73	X-klive				
fischerP1_30	4	139	3	141	1	122	2	114	2	128	1	58	X-klive	2	162	X-sbmc				
fischerP1_60	15	159	15	177	6	145	8	128	8	140	0	58	X-klive	11	389	X-msat				
fischerP1_90	35	184	47	226	21	171	20	140	19	155	1	58	X-klive	23	624	X-msat				
hrcSat_20	25	461	15	534	7	388	11	341	14	462	94	1591	X-bmcinc	132	999	X-sbmc				
hrcP1_30	273	940	159	1095	73	929	123	771	144	1110	231	4047	X-sbmcinc	260	921	S-sbmcinc				
hrcP1_60	1831	1575	716	2031	452	1620	797	1360	596	1931	835	7356	X-sbmcinc	921	7651	S-sbmcinc				
hrcP1_90	TO	2137	3227	2449	2506	2226	1937	1832	2809	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----
txt4Sat_20	18	244	9	237	4	193	6	171	5	186	26	1189	X-sbmc	30	886	X-coismt				
txt4P1_30	63	315	25	324	10	264	19	218	18	236	77	1527	X-msat	101	3612	X-sbmc				
txt4P1_60	333	484	141	588	95	433	127	343	114	338	289	3440	X-msat	411	3880	X-sbmcinc				
txt4P1_90	1176	650	427	943	446	646	432	459	325	455	618	5769	X-msat	885	5494	X-sbmcinc				
txt8Sat_20	29	312	13	306	6	244	11	206	9	226	52	1859	X-sbmc	66	1271	X-msatcoi				
txt8P1_30	115	408	44	428	14	341	31	271	26	294	143	2080	X-msat	189	5555	X-sbmc				
txt8P1_60	789	650	217	917	144	602	206	471	150	460	532	4768	X-msat	630	4862	X-sbmcinc				
txt8P1_90	2830	929	647	1528	630	875	666	632	528	661	1229	8460	X-msat	1352	7068	X-sbmcinc				
sdserver12Sat_50	23	306	18	280	8	218	10	193	11	212	4	310	X-sbmc	8	367	S-sbmc				
sdserver12P1_60	254	912	339	1114	255	817	232	598	169	649	256	6899	X-sbmc	396	3816	X-sbmcinc				
sdserver12P1_90	736	1273	895	1842	782	1277	566	811	407	840	883	5320	X-sbmcinc	945	5598	S-sbmcinc				
sdserver12P1_120	1270	1671	1818	2921	1511	1641	1196	1070	791	1136	1790	7097	X-sbmcinc	1929	7521	S-sbmcinc				
sdserver13Sat_50	37	369	19	346	10	262	17	233	16	255	9	517	X-sbmc	17	639	S-sbmc				
sdserver13P1_60	414	1014	387	1224	310	918	272	649	203	692	324	7631	X-sbmc	551	4414	S-sbmcinc				
sdserver13P1_90	858	1467	1013	2077	1001	1428	644	917	474	924	1128	6070	X-sbmcinc	1216	6442	S-sbmcinc				
sdserver13P1_120	1478	1836	2125	3208	1907	1817	1360	1199	896	1249	2081	8049	X-sbmcinc	2232	8388	S-sbmcinc				
Solved Instances	97%	100%	97%	100%	100%	100%	91%	88%												
Total T (s)	12853	9348	8556	6922	5090	-----	12259	14907												
Total M (MB)	17694	24385	16702	12387	13978	117575	117575	105568												

ones. This paper pursues a bounded approach, and Section 4 compares *sbvzot* against similar ones, and in particular those presented in [18], [19], [20], [33] and [9]. Common complete techniques include automata-based and tableau-based approaches. An exhaustive evaluation of several techniques and tools (including some that are not based on translation to Büchi automata or on bounded approaches) for LTL satisfiability checking can be found in [37]. Although, given their difference in nature, we did not compare our tools against complete ones, in this section we also provide a brief overview of the latter.

As for automata-based approaches (e.g., SPIN [17]), Rozier and Vardi [38] carried out a comparison of satisfiability checkers for LTL formulae based on the translation of LTL formulae into Büchi automata. Rozier and Vardi [39] also propose a novel translation of LTL formulae into Transition-based Generalized Büchi Automata, inspired by the translation presented in [40]. Such automata are used by SPOT [41], which is claimed to be the best explicit LTL-to-Büchi automata translator for satisfiability checking based on the experiments carried out in [38]. Li *et al.* [42] present a novel on-the-fly construction of Büchi automata from LTL formulae that is particularly well suited for finding models of LTL formulae when they exist.

In tableau-based approaches, the LTL formula is analyzed on a tableau—that is, a set of nodes. The root node is labeled by the main LTL formula, and it is repeatedly decomposed based on the tableau rules that create successors labeled by a set of formulae. The LTL formula is satisfiable if, and only if, there exists at least one successful

branch. Goranko *et al.* [43] report on the implementation and experimental evaluation of two well-known tableau-based approaches: Wolper’s multi-pass, LTL tableau presented in [44], and Schwendimann’s one-pass LTL tableau procedure presented in [45], with an evident superior performance to the latter.

Reynolds [46] introduces a novel traditional-style, one-pass, tree-shaped tableau for LTL. The fact that branches can be explored down independently makes this approach particularly suitable for parallel implementation, whereas Schwendimann’s approach [45] requires the full development of branches.

Given the different nature of our approach with respect to automata- and tableau-based ones we did not compare our tools against them, and focused on similar, BSC-based approaches instead.

A simple translation of LTL formulae to Conjunctive Normal Form (CNF) formulae is presented in [19], which deals with the semantic equivalence of LTL and Computation Tree Logic (CTL) when each step has only one successor in the Kripke structure. Another bounded encoding is presented in [20], which virtually unrolls the path up to the maximum depth of past operators (d) in the LTL formula. Unlike other bounded approaches (with bound k), this encoding unfolds the LTL formula up to $d * k$ steps, instead of k .

NuSMV [47] is a symbolic model checker that supports both BDD-based and SAT-based model checking. NuSMV can check LTL and CTL properties against finite state system models, so it can be used as a satisfiability checker for

LTL and CTL formulae. Several algorithms are implemented in NuSMV for the satisfiability checking of LTL formulae. nuXmv [48] is an extension of NuSMV that supports both finite and infinite-state synchronous transition systems. nuXmv extends NuSMV by augmenting basic verification algorithms for finite-state systems and providing new data types and advanced SMT-based model checking techniques for infinite-state systems. Furthermore, nuXmv is the basis for various tools for requirements analysis, contract-based design, model checking of hybrid systems, safety assessment, and software model checking [16]. nuXmv offers more algorithms for checking the satisfiability of LTL formulae than NuSMV.

6 CONCLUSION

This paper presents a new encoding of LTL formulae in bit-vector logic. The encoding is used to solve the satisfiability problem for LTL formulae through a bounded approach. Besides demonstrating the benefits of the proposed encoding by comparing it against the original bv logic-based encoding and some well-known, more “classical” solutions, the paper also investigates the gains provided by the specific SMT checker adopted. While the original proposal exploits Z3, we also carried out experiments with Boolector, Yices2, Mathsatz, and CVC4. Obtained results show that the benefits are mainly independent of the specific solver. All proposed checkers are implemented as dedicated plugins of Zot, our bounded satisfiability checker.

ACKNOWLEDGMENTS

Part of the work was carried out while the first author was at Politecnico di Milano.

REFERENCES

- [1] A. Pnueli, “The temporal logic of programs,” in *Proc. 18th Annu. Symp. Found. Comput. Sci.*, 1977, pp. 46–67.
- [2] K. Y. Rozier, “Linear temporal logic symbolic model checking,” *Comput. Sci. Rev.*, vol. 5, no. 2, pp. 163–203, 2011.
- [3] L. Tan, O. Sokolsky, and I. Lee, “Specification-based testing with linear temporal logic,” in *Proc. IEEE Int. Conf. Inf. Reuse Integr.*, 2004, pp. 493–498.
- [4] P. Tabuada and G. Pappas, “Linear time logic control of discrete-time linear systems,” *IEEE Trans. Autom. Control*, vol. 51, no. 12, pp. 1862–1877, Dec. 2006.
- [5] L. Baresi, M. M. Pourhasem Kallehbasti, and M. Rossi, “Flexible modular formalization of UML sequence diagrams,” in *Proc. 2nd FME Workshop Formal Methods Softw. Eng.*, 2014, pp. 10–16.
- [6] A. Bauer, M. Leucker, and C. Schallhart, “Runtime verification for LTL and TLTL,” *ACM Trans. Softw. Eng. Methodol.*, vol. 20, no. 4, pp. 14:1–14:64, 2011.
- [7] G. Fainekos, H. Kress-Gazit, and G. Pappas, “Temporal logic motion planning for mobile robots,” in *Proc. IEEE Int. Conf. Robot. Autom.*, 2005, pp. 2020–2025.
- [8] K. Y. Rozier and M. Y. Vardi, “LTL satisfiability checking,” in *Proc. Int. SPIN Workshop Model Checking Softw.*, 2007, vol. 4595, pp. 149–167.
- [9] M. Pradella, A. Morzenti, and P. San Pietro, “Bounded satisfiability checking of metric temporal logic specifications,” *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 3, pp. 20:1–20:54, 2013.
- [10] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, “Symbolic model checking without BDDs,” in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 1999, vol. 1579, pp. 193–207.
- [11] L. Baresi, M. M. Pourhasem Kallehbasti, and M. Rossi, “Efficient scalable verification of LTL specifications,” in *Proc. 37th Int. Conf. Soft. Eng.*, 2015, pp. 711–721.
- [12] Microsoft research, “Z3: An efficient SMT solver,” 2020. [Online]. Available: <https://github.com/Z3Prover/z3>
- [13] The Zot bounded model/satisfiability checker, 2020. [Online]. Available: <https://github.com/fm-polimi/zot>
- [14] A. Cimatti et al., “NuSMV 2: An opensource tool for symbolic model checking,” in *Proc. Int. Conf. Comput. Aided Verification*, 2002, vol. 2404, pp. 359–364.
- [15] A. Cimatti, A. Griggio, B. J. Schaafsma, and R. Sebastiani, “The MathSAT5 SMT solver,” in *Proc. 19th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2013, pp. 93–107.
- [16] R. Cavada et al., “The nuXmv symbolic model checker,” in *Proc. Int. Conf. Comput. Aided Verification*, 2014, vol. 8559, pp. 334–342.
- [17] G. J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*, vol. 1003. Boston, MA, USA: Addison-Wesley Reading, 2004.
- [18] A. Biere, K. Heljanko, T. A. Junttila, T. Latvala, and V. Schuppan, “Linear encodings of bounded LTL model checking,” *Log. Methods Comput. Sci.*, vol. 2, no. 5, pp. 1–64, 2006.
- [19] T. Latvala, A. Biere, K. Heljanko, and T. Junttila, “Simple bounded LTL model checking,” in *Proc. Int. Conf. Formal Methods Comput.-Aided Des.*, 2004, vol. 3312, pp. 186–200.
- [20] T. Latvala, A. Biere, K. Heljanko, and T. Junttila, “Simple is better: Efficient bounded model checking for past LTL,” in *Proc. Int. Workshop Verification Model Checking Abstract Interpretation*, 2005, vol. 3385, pp. 380–395.
- [21] The annual SMTCOMP competition website, 2020. [Online]. Available: <http://www.smtcomp.org>
- [22] A. Niemetz, M. Preiner, and A. Biere, “Boolector 2.0,” *J. Satisfiability Boolean Model. Comput.*, vol. 9, pp. 53–58, 2015.
- [23] B. Dutertre and L. De Moura, “The YICES SMT solver,” *Tool paper at http://yices.csl.sri.com/tool-paper.pdf*, vol. 2, no. 2, pp. 1–2, 2006.
- [24] C. Barrett et al., “CVC4,” in *Proc. Int. Conf. Comput. Aided Verification*, 2011, pp. 171–177.
- [25] L. Baresi, M. M. Pourhasem Kallehbasti, and M. Rossi, “How bit-vector logic can help improve the verification of LTL specifications over infinite domains,” in *Proc. 31st Annu. ACM Symp. Appl. Comput.*, 2016, pp. 1666–1673. [Online]. Available: <http://doi.acm.org/10.1145/2851613.2851833>
- [26] O. Lichtenstein, A. Pnueli, and L. Zuck, “The glory of the past,” in *Proc. Workshop Logics Programs*, 1985, vol. 193, pp. 196–218.
- [27] F. Laroussinie, N. Markey, and P. Schnoebelen, “Temporal logic with forgettable past,” in *Proc. Symp. Logic Comput. Sci.*, 2002, pp. 383–392.
- [28] C. A. Furia, D. Mandrioli, A. Morzenti, and M. Rossi, *Modeling Time in Computing*. Berlin, Germany: Springer, 2012.
- [29] L. De Moura and G. O. Passmore, “The strategy challenge in SMT solving,” in *Automated Reasoning and Mathematics*. Berlin, Germany: Springer, 2013, pp. 15–44.
- [30] C. Ghezzi, D. Mandrioli, and A. Morzenti, “TRIO: A logic language for executable specifications of real-time systems,” *J. Syst. Softw.*, vol. 12, no. 2, pp. 107–123, 1990.
- [31] C. Heitmeyer and D. Mandrioli, *Formal Methods for Real-Time Computing*. New York, NY, USA: Wiley, 1996.
- [32] F. Vicentini, M. Askarpour, M. Rossi, and D. Mandrioli, “Safety assessment of collaborative robotics through automated formal verification,” *IEEE Trans. Robot.*, vol. 36, no. 1, pp. 42–61, Feb. 2020. [Online]. Available: <https://doi.org/10.1109/TRO.2019.2937471>
- [33] K. Heljanko, T. Junttila, and T. Latvala, “Incremental and complete bounded model checking for full PLTL,” in *Proc. Int. Conf. Comput. Aided Verification*, 2005, vol. 3576, pp. 98–111.
- [34] R. P. Kurshan, *Computer-Aided Verification of Coordinating Processes: The Automata-Theoretic Approach*. Princeton, NJ, USA: Princeton Univ. Press, 2014.
- [35] A. Cimatti, M. Roveri, and D. Sheridan, “Bounded verification of past LTL,” in *Proc. Int. Conf. Formal Methods Comput.-Aided Des.*, 2004, pp. 245–259.
- [36] SMTCOMP competition 2018. [Online]. Available: <http://smtcomp.sourceforge.net/2018>
- [37] V. Schuppan and L. Darmawan, “Evaluating LTL satisfiability solvers,” in *Proc. Int. Symp. Autom. Technol. Verification Anal.*, 2011, vol. 6996, pp. 397–413.
- [38] K. Y. Rozier and M. Y. Vardi, “LTL satisfiability checking,” *Int. J. Softw. Tools Technol. Transfer*, vol. 12, no. 2, pp. 123–137, 2010.
- [39] K. Y. Rozier and M. Y. Vardi, “A multi-encoding approach for LTL symbolic satisfiability checking,” in *Proc. Int. Symp. Formal Methods*, 2011, vol. 6664, pp. 417–431.

- [40] D. Giannakopoulou and F. Lerda, "From states to transitions: Improving translation of LTL formulae to Büchi automata," in *Proc. Int. Conf. Formal Techn. Netw. Distrib. Syst.*, 2002, vol. 2529, pp. 308–326.
- [41] A. Duret-Lutz and D. Poitrenaud, "SPOT: An extensible model checking library using transition-based generalized Büchi automata," in *Proc. Ann. Int. Symp. Model. Anal. Simul. Comput. Telecommun. Syst.*, 2004, pp. 76–83.
- [42] J. Li, L. Zhang, G. Pu, M. Vardi, and J. He, "LTL satisfiability checking revisited," in *Proc. Int. Symp. Temporal Representation Reasoning*, 2013, pp. 91–98.
- [43] V. Goranko, A. Kyrilov, and D. Shkatov, "Tableau tool for testing satisfiability in LTL: Implementation and experimental analysis," *Electron. Notes Theor. Comput. Sci.*, vol. 262, pp. 113–125, 2010.
- [44] P. Wolper, "The tableau method for temporal logic: An overview," *Logique et Analyse*, vol. 28, no. 110/111, pp. 119–136, 1985.
- [45] S. Schwendimann, "A new one-pass tableau calculus for PLTL," in *Proc. Int. Conf. Autom. Reasoning Analytic Tableaux Related Methods*, 1998, pp. 277–291.
- [46] M. Reynolds, "A traditional tree-style tableau for LTL," *CoRR*, vol. abs/1604.03962, 2016. [Online]. Available: <http://arxiv.org/abs/1604.03962>
- [47] The NuSMV model checker, 2020. [Online]. Available: <http://nusmv.fbk.eu/>
- [48] The nuXmv model checker, 2020. [Online]. Available: <https://nuxmv.fbk.eu/>



Matteo Rossi is an associate professor at Politecnico di Milano, Italy. His research interests are in formal methods for safety-critical and real-time systems, architectures for real-time distributed systems, and transportation systems both from the point of view of their design, and of their application in urban mobility scenarios.



Luciano Baresi is a full professor at the Politecnico di Milano, Italy. He was visiting professor at the University of Oregon, Eugene, Oregon, and visiting researcher at the University of Paderborn, Germany. His research interests are in the broad area of software engineering and include formal approaches for modeling and specification languages, distributed systems, service-based applications and mobile, self-adaptive, and pervasive software systems.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdn.



Mohammad Mehdi Pourhashem Kallehbasti received the PhD degree in software engineering from Politecnico di Milano, Italy, in 2015. He is an assistant professor at the University of Science and Technology of Mazandaran, Behshahr. His research interests are in software engineering and formal methods.

Orderly Generation of Test Data via Sorting Mutant Branches Based on Their Dominance Degrees for Weak Mutation Testing

Xiangjuan Yao^{ID}, Gongjie Zhang^{ID}, Feng Pan^{ID}, Dunwei Gong^{ID}, Member, IEEE, and Changqing Wei

Abstract—Compared with traditional structural test criteria, test data generated based on mutation testing are proved more effective at detecting faults. However, not all test data have the same potency in detecting software faults. If test data are prioritized while generating for mutation testing, the defect detectability of the test suite can be further strengthened. In view of this, we propose a method of test data generation for weak mutation testing via sorting mutant branches based on their dominance degrees. First, the problem of weak mutation testing is transformed into that of covering mutant branches for a transformed program. Then, the dominance relation of mutant branches in the transformed program is analyzed to obtain the non-dominated mutant branches and their dominance degrees. Following that, we prioritize all non-dominated mutant branches in descending order by virtue of their dominance degrees. Finally, the test data are generated in an orderly manner by selecting the mutant branches sequentially. The experimental results on 15 programs show that compared with other methods, the proposed test data generation method can not only improve the error detectability of the test suite, but also has higher efficiency.

Index Terms—Software testing, mutation testing, test data generation, mutant branch, dominance degree

1 INTRODUCTION

THE purpose of software testing is to detect as many faults or defects as possible in a software product. To this end, we first need to generate a test suite according to a given testing criterion. Then, the program under test will be executed with the test data to determine whether the program has defects. The more defects a test datum can find, the higher the quality of the test datum is considered [1]. So, how to find test data with higher defect detectability is a crucial issue of software testing.

Various methods of test data generation are presented based on specific adequacy criteria, e.g., statement coverage criterion or branch coverage criterion. However, test data

generated based on structural coverage criteria generally result in a low capability in detecting faults [2].

Mutation testing, proposed by Demillo *et al.* [3] and Hamlet [4], can simulate real faults of a software product by making one or more grammatical changes to the program under test [5]. Mutation testing was used not only to validate the fault detectability of existing test suites [6], but also to generate test data that can find these seeded faults [7], [8]. Previous studies has proved that test data generated based on mutation testing are capable of finding more faults than those generated under various structural coverage criteria [9], [10]. In contrast, test data designed for traditional coverage testing criteria, such as statement and branch coverage, do not have similar fault detectability with mutation testing [11], [12].

Despite its high defect detectability, mutation testing is deeply suffered from the problem of high cost. Howden first proposed the idea of weak mutation testing [13]. Although weak mutation testing weakens the defect detectabilities of test data to some extent comparing with strong mutation testing [2], the computational cost of weak mutation testing will be greatly reduced. In addition, weak mutation testing is still much stronger than statement and branch coverage testing [14]. So we used weak mutation criterion to generate test data in this paper to achieve the balance between costs and capabilities.

Although test data generated based on mutation testing are more effective at detecting faults as a whole, different mutants have different performance. Some mutants are hard to be killed, but others are not [15]. Therefore, different test data also have different defect detectabilities under the criterion of mutation testing. During the period of software testing, the earlier a software fault is discovered, the less the

- Xiangjuan Yao is with the School of Mathematics, China University of Mining and Technology, Xuzhou, Jiangsu 221116, China.
E-mail: yaoxj@cumt.edu.cn.
- Gongjie Zhang is with the School of Information and Control Engineering, China University of Mining and Technology, Xuzhou, Jiangsu 221116, China, and also with the School of Computer Science and Technology, Jiangsu Normal University, Xuzhou, Jiangsu 221116, China.
E-mail: zhanggongjie@126.com.
- Feng Pan is with the School of Information and Control Engineering, China University of Mining and Technology, Xuzhou, Jiangsu 221116, China. E-mail: fengpan0315@126.com.
- Dunwei Gong is with the School of Information and Control Engineering, China University of Mining and Technology, Xuzhou, Jiangsu 221116, China. E-mail: dwgong@vip.163.com.
- Changqing Wei is with School of Mathematics, China University of Mining and Technology, Xuzhou, Jiangsu 221116, China.
E-mail: 1536113693@qq.com.

Manuscript received 2 Jan. 2020; revised 29 July 2020; accepted 30 July 2020.
Date of publication 7 Aug. 2020; date of current version 18 Apr. 2022.
(Corresponding author: Dunwei Gong.)
Recommended for acceptance by D. Hao.
Digital Object Identifier no. 10.1109/TSE.2020.3014960

cost of correcting the fault will be. Consequently, we hope to execute the test data that have high defect detectabilities as earlier as possible, suggesting that it is significant to prioritize the test data for mutation testing.

In fact, there have been many studies on test data prioritization in regression testing [16]. However, few studies consider the order of test data when generating. If test data are prioritized while generating for mutation testing, the overall defect detectability of the test suite can be further strengthened.

In view of this, we propose a method of test data generation for weak mutation testing based on the dominance degrees of mutant branches. First, we transform all mutants of the original program into mutant branches according to the method proposed by Papadakis and Malevris [17], and form a new program by inserting all these mutant branches into the original program. Then, the dominance relation of mutant branches in the transformed program is analyzed to obtain the non-dominated mutant branches and their dominance degrees. Following that, we prioritize all non-dominated mutant branches based on their dominance degrees. Finally, test data are generated in sequence according to the priorities of the mutant branches. Given the fact that the test data covering a mutant branch with a higher dominance degree can find more program errors, the test data generation method according to the dominance degree can further improve the defect detectability of the test suite. We apply the proposed method to 15 programs. Experimental results show that, compared with other methods, the proposed method can improve the quality of test data, as well as reduce the cost of mutation testing.

The main contributions of this paper are as follows:

- 1) Putting forward the concept of dominance degree and proposing a method to calculate it.
- 2) Presenting a method of test data generation in an orderly manner.

The rest of this paper is organized as follows. Section 2 reviews the related work. The preliminary of the proposed method is presented in Section 3. Section 4 illustrates the prioritization of non-dominated mutant branches. The method of orderly generating test data based on the prioritization of non-dominated mutant branches is proposed in Section 5. Section 6 provides the experimental results and analysis. The threats to validity of the proposed method are listed in Section 7. Finally, Section 8 concludes the whole paper and points out several topics to research in the future.

2 RELATED WORK

Since the paper studies test data generation for mutation testing, this section first summarizes the work on mutation testing, and then reviews the methods of test data generation based on mutation testing.

2.1 Mutation Testing

As a fault-oriented testing technique [18], [19], mutation testing can simulate real faults by injecting artificial fault(s) into a program under test. The program after fault injection is called a mutant, and the rules of fault injection are called mutation operators. A mutant is killed if its output differs from that of the original program. A mutant that is functionally identical

to the original program and cannot be killed by any test datum is called an equivalent one. Generally, the mutation score that reflects the adequacy of mutation testing, i.e., the capability of test data in fault detection [20], is defined as the ratio of the number of killed mutants to the total number of non-equivalent mutants.

Compared with traditional structural test criteria, test data generated based on mutation testing are more effective at detecting faults [21]. Papadakis *et al.* investigated the relation between two independent variables, mutation score and test suite size, with one dependent variable, the detection of (real) faults [22]. Their empirical study shows that mutants provide good guidance for improving the fault detection of test suites, but their correlation with fault detection are weak. In addition, the relation between defect detection and mutation score is non-linear.

Despite its high defect detectability, mutation testing is deeply suffered from the problem of high cost.

One of the main reasons for the high cost of mutation testing is the large number of mutants, which not only increases the cost of running programs, but also skews the measurement of mutation score. Mutant sampling and selective mutation testing are classical methods to solve this problem. Here, mutant sampling randomly samples a small percentage of mutants for mutation testing [23], [24], however, the loss in mutation adequacy increases when decreasing the sampling rate [25]. Selective mutation testing selectively omits some mutation operators to generate a small number of mutants [26], [27], but the test effectiveness will obtain bigger discount when omitting more mutation operators. High order mutants (HOMs, more than one injected fault in a program) can not only reduce the number of mutants, but also simulate complex faults in a software product, and therefore attract researchers in the community of software engineering [28], [29]. Nevertheless, the expensive computation of generating HOMs prevents this technique from practical applications.

To save time spent in executing mutants, Howden first proposed the idea of weak mutation testing [13]. A mutant is weakly killed by a test datum if after the execution of the mutant statement, its state immediately differs from the corresponding state of the original program. Therefore, we can judge whether the mutant is weakly killed or not according to the mutant statement. As a result, running the codes after the mutant statement is unnecessary [30].

Previous studies have shown that test data designed for weak mutation testing have almost the same defect detectability as strong mutation testing while saving 50 percent execution cost [31], [32]. While the research of Chekam *et al.* shows that there is a large difference in defect detectability between weak and strong mutation testing [2].

Papadakis and Malevris structured mutant branches by combining original and mutant statements, and then formed a new program by integrating all these mutant branches into the original program to further reduce the execution cost of weak mutation testing [17]. Based on the work of Papadakis and Malevris, Gong *et al.* reduced the number of mutant branches according to their dominance relations [33]. In addition, Zhang *et al.* further present a statistical method to analysis the dominance relation of mutant branches [34].

Although weak mutation testing is not as powerful as strong mutation testing, it greatly reduces the test cost. In

addition, it is still much stronger than traditional coverage testing, such as statement and branch coverage [14], [31]. So we used weak mutation testing in this paper to achieve the balance between costs and capabilities. Of course, the proposed method can also be applied to strong mutation testing.

Although mutation testing can obtain stronger test suite, not all test data have the same defect detectability. Some mutants are hard to kill, whereas others are not [15], [35]. Test data generated based on mutants that are hard to kill have higher ability to detect defects than mutants that are easy to kill. During the period of software testing, the earlier a fault is detected in the software life cycle, the cheaper the cost of correcting the fault [36]. Consequently, we hope to execute the test data that have high defect detectabilities as earlier as possible. As a result, it is significant to prioritize the test data in mutation testing.

2.2 Test Data Generation Based on Mutation Testing

Generating test data is a critical task in software testing. As a testing criterion, mutation testing can help to generate test data by maximizing the mutation score. Constraint-based test data generation (CBT) is an early method for mutation testing [30]. CBT transforms conditions of killing a mutant into a number of constraints, and generates test data by constraint-solving methods. Experimental results suggest that test data generated by CBT can kill over 90 percent mutants. Dynamic domain reduction (DDR) improves the effectiveness of CBT, and generates test data by the backtracking search [37]. However symbolic execution used in DDR and CBT results in a high cost of test data generation. By optimizing traditional symbolic execution [38] [39], dynamic symbolic execution (DSE) first generates an initial test datum, and then searches for conditions of a path to generate a new one by executing the initial test datum. The above methods are mainly based on the control flow graph (CFG) of a program under test. Taking both control and data dependence relations into consideration, Liu *et al.* improved the success rate and the efficiency of DDR [40]. But the ability of DSE is largely constrained by the performance of a constraint solver.

In recent years, search-based software testing (SBST) has attracted great attention from researchers in the community of generating test data [41], [42], [43], [44], [45], [46], [47], [48]. Zhang *et al.* intended to improve the efficiency of generating test cases for mutation testing by a set-based genetic algorithm [49]. Papadakis and Malevris improved the practicability of the generated test data via path selection [50]. Jia *et al.* focused on searching HOMs which hard to kill by evolutionary optimization [28]. May aimed at discerning useful parameter settings to improve the overall effectiveness of both the genetic algorithm and the immune inspired algorithm [51]. Matnei Filho and Vergilio introduced a multi-objective test data generation approach for mutation testing [52]. According to the study of Carlos *et al.*, SBST is a very promising method for mutation-oriented test data generation [53]. In most cases, SBST is very efficient in generating test data. However, a large number of mutants result in massive execution to evaluate the performance of the mutants, which raises a huge computation cost.

The above methods of generating test data generally guarantee that the test suite has a high defect detectability as a whole. Nevertheless, there have been seldom methods to prioritize test data while generating in mutation testing. As a result, we aim to generate test data in an orderly manner to further enhance their performance in mutation testing.

3 PRELIMINARIES

This section first describes the transformation method of Papadakis and Malevris [17], by which mutant branches are constructed and a new program is formed. Then, the mutant reduction method based on the dominance relation of mutant branches proposed by Gong *et al.* is presented [33].

3.1 Transformation Method for Weak Mutation Testing

The original program is denoted as P , and a mutation point (a statement or an expression) of P is denoted as s . By performing a mutation operator on s , we will obtain a mutated statement of s , denoted as s' . By substituting s' for s , a mutant, denoted as m , is created. A number of mutants can be created by applying a series of mutation operators to different statements of P . The set composed of all mutants is denoted as M .

If a test datum, t , can execute the mutant statement, s' , and a different state appears after executing s' in m , according to the criterion of weak mutation testing, t kills mutant m . Since the state of m differs from that of P just after the execution of s' , the value of predicate " $s' \neq s''$ " is true. Taking " $s' \neq s''$ " as the predicate, a branch statement, denoted as b , is constructed. If a test datum covers the true branch of b , i.e., the value of " $s' \neq s''$ " is true, then the test datum kills m under the criterion of weak mutation testing. Thus the problem of killing mutant m will be transformed into that of covering the true branch of b .

Definition 1. *In view that there is a one-to-one correspondence between branch statement b and mutant m , b is called a mutant branch.*

The set of all mutant branches is denoted as B . Then B and M have the same size, i.e., $|B| = |M|$.

By inserting all mutant branches of B into P , a new program is formed, denoted as P' . If a test suite, T , can cover the true branches of all mutant branches in P' , according to the criterion of weak mutation testing, T will kill all mutants in M . Then the problem of killing mutants of P under the criterion of weak mutation testing is transformed into that of covering the true branches of all mutant branches in P' .

It should be noted that mutant branch b generally does not perform any operation in P' , but indicates whether the corresponding mutant m is killed or not. Therefore, regardless whether b is executed or not, P' has the same function as P . For operators $++v$, $--v$, $v++$ and $v--$, the value of variable v will change after the execution of the mutant branch. In this circumstance, we insert a statement to P' after the mutant branch to recover the value of the variable. For $++v$ and $v++$, we add $v = v - 1$; for $-v$ and $v--$, we add $v = v + 1$.

Given the fact that equivalent mutants do not make any sense for generating test data, we exclude them in the process.

For this purpose, a semi-automatic method is proposed to identify equivalent mutants before transforming a mutant into the corresponding mutant branch. To fulfill this task, we first run all the mutants against a test suite using the tool MuClipse, and obtain alive mutants. MuClipse is a plugin in Eclipse and performs mutation testing for Java programs. The killed mutants are certainly not equivalent. For alive mutants, artificial analysis is used to identify whether they are equivalent or not. It is well known that whether a mutant is equivalent or not is an undecidable problem [54]. In order to improve the accuracy of the analysis results, if a alive mutant is determined as nonequivalent, we will design new test data to kill it; otherwise, we will use a large number of test data to run this mutant to ensure that it will not be killed.

There have been a number of approaches to automatically detecting equivalent mutants. Kintis and Malevris employed data flow patterns to reveal locations that likely generate equivalent mutants [55]. Additionally, they found that mirrored mutants often exhibit analogous behavior of equivalence [56], and if one of mirrored mutants is equivalent, the others are also equivalent. Papadakis *et al.* used a series of classification strategies to isolate equivalent mutants after executing test suites [57], and the experimental study suggested that only a small number of live mutants need to be manually analyzed. Although the above methods can automatically detect equivalent mutants, thus reducing the cost of manual analysis, they have low precision. As a result, manually analyzing equivalent mutants has been the most commonly used method in software testing community up to date [58].

3.2 Mutant Reduction Based on Dominance Relation

There is often close correlation between different branch statements. By analyzing the dominance relation between mutant branches in the new program P' , the mutants corresponding to the dominated mutant branches will be reduced.

Definition 2. Consider two mutant branches, b_i and b_j , in P' .

For any test datum of P , if the true branch of b_i is executed, that of b_j must be executed, b_i is said to dominate b_j , denoted as $b_i \succ b_j$. On this circumstance, b_i is the dominant branch, and b_j is the dominated one.

From Definition 2, the dominance relation has transitivity: for three mutant branches, b_i , b_j , and b_k , in P' , if $b_i \succ b_j$ and $b_j \succ b_k$, then $b_i \succ b_k$.

Specially, if $b_i \succ b_j$ and $b_j \succ b_i$, b_i is said to be equivalent with b_j , denoted as $b_i \cong b_j$. In this circumstance, we select the mutant branch having a front position in P' as the dominant one, and the other dominance relationship is ignored. The reason is that the mutant branch with a front position will be executed early when generating test data so as to potentially save the execution cost.

By Definition 2, test data that cover the true branch of a dominant branch can also cover the true branches of all its dominated ones. Therefore, any dominated mutant branch can be reduced.

As mentioned in Section 2.3, Zhang *et al.* presented a method of statistically determine the dominance relation of mutant branches by running the program against a large number of test data [34], which will be adopted in this paper.

For two mutant branches b_i and b_j , we will present a statistical method to determine the dominance relation them. Let

$$X = \begin{cases} 1, & \text{if the true branch of } b_i \text{ is covered} \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

$$Y = \begin{cases} 1, & \text{if the true branch of } b_j \text{ is covered} \\ 0, & \text{otherwise} \end{cases} \quad (2)$$

Then X and Y can be regarded as two random variables. $b_i \succ b_j$ if and only if $P\{Y = 1|X = 1\} = 1$. Let $P\{Y = 1|X = 1\} = p$. The value of p can be estimated by the statistical method.

First, we use the random method to automatically generate a test suite of size N . Then, each test datum is used to run the program, and the coverage results of mutant branches are counted. For each test datum t , we can obtain the value of X and Y . Because there are N test data, we can obtain a sample of size N , i.e., $(X_1, Y_1), \dots, (X_N, Y_N)$. By the maximum likelihood estimation method, the estimator of p is:

$$\hat{p} = \frac{\sum_{i=1}^N X_i Y_i}{\sum_{i=1}^N X_i}.$$

If $p = P\{Y = 1|X = 1\} = 1$, we obtain $b_i \succ b_j$. The above process can be implemented automatically, including the generation of test data, the calculation of statistics and the determination of dominance relation.

Because all mutant branches are inserted into the same program, we only need to run the program one time with each test datum, therefore, the cost of determining the dominant relation is very small. However, the determination may have a few errors due to the insufficient test suite. In order to improve the accuracy of the determination results, the test suite should be sufficient enough. In this paper, we use decision-branch coverage criterion to generate test data. In addition, the larger the value of N is, the more accurate the result will be. Therefore, we can increase the value of N as much as possible under the premise of computing power.

Definition 3. The set of all the dominance relations between mutant branches in B is called the **dominance relation set** of B , denoted as D^B .

Definition 4. If for any $b_j \in B$, $b_j \succ b_i$ is not held, b_i is called a non-dominated mutant branch. The set of all non-dominated mutant branches is called the non-dominated mutant branch set, denoted as B^{nd} .

By Definitions 2 and 4 we know that test data that cover all branches of B^{nd} must cover all those of B . So we only consider the mutant branches of B^{nd} to generate test data. In this way, the number of mutant branches is greatly reduced.

The specific steps to obtain the non-dominant mutant branches can be provided as follows.

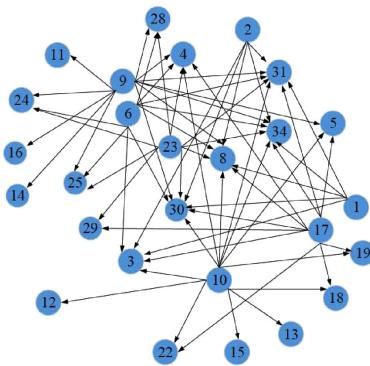
- Step 1: For each $m \in M$, construct a mutant branch b . The set of all mutant branches is denoted as B .
- Step 2: Inserting all mutant branches of B into P , a new program is formed, denoted as P' .
- Step 3: By investigating the dominance relation of all mutant branches in P' , we obtain their dominance relation set, denoted as D^B .

```

1  public static int max(int a, int b){
2      int max;
3      if(a > b){
4          max = a;
5      }else{
6          max = b;
7      }
8      return max;
9  }

```

(a)



(b)

Fig. 1. Dominance graph of an simple program, where (a) is the original program; and (b) is the dominance graph of mutant branches.

- Step 4: Obtain the set of all non-dominated mutant branches according to their dominance relation, denoted as B^{nd} .

4 PRIORITIZATION OF MUTANT BRANCHES BASED ON DOMINANCE DEGREE

This section first gives the definition of dominance degree, and then prioritizes all non-dominated mutant branches according to their dominance degrees.

4.1 Dominance Degree

In order to describe the dominance relation more intuitively, we introduce the concept of dominance relation graph.

Definition 5. The dominance relation graph of P' is a directional graph, denoted as $DG(P') = (V(B), E(D^B))$, where the vertex set $V(B) = B$, and $E(D^B)$ is the edge set such that for $\forall b_i \succ b_j \in D^B$, there is an edge $< b_i, b_j > \in E(D^B)$.

In the dominance relation graph, the in-degree of a vertex, $d_-(b)$, represents the number of mutant branches that dominate b , and the out-degree of a vertex, $d_+(b)$, represents the number of branches that are dominated by b . So the in-degree of the non-dominated branch is 0.

Fig. 1a presents an example program. By applying mutation operators to this program, 34 mutants are generated. The mutant branches transformed from all these mutants are listed in Table 1. For example, by changing “ $a > b$ ” to “ $a < b$ ” in statement 3, we obtain a mutant. Then $if(a > b)! = (a < b)$ will be a mutant branch. Among all these mutants, 7 are equivalent under the criterion of strong mutation testing, corresponding to mutant branches $b_7, b_{20}, b_{21}, b_{26}, b_{27}, b_{32}$, and b_{33} (underlined). After analysis, there are 64 dominance relations among these 27 non-equivalent mutant branches, listed in Table 2. Based on the dominance relations of mutant branches, a dominance relation graph is built, shown as Fig. 1b, in which vertices of in-degree zero correspond to non-dominated mutant branches.

Definition 6. The number of mutant branches dominated by $b_i \in B$ is the dominance degree, denoted as $df(b_i)$.

From Definition 6, the dominance degree of $b_i \in B$ equals to the out-degree of b_i in $DG(P')$. The value of $df(b_i)$

indicates the capability of a test datum that covers b_i in covering all the dominated mutant branches of b_i .

It is enough to consider only the non-dominated mutant branches when generating test data. Table 3 lists all the non-dominated mutant branches and their dominance degrees of the example program in Fig. 1a. For example, b_1 dominates b_3, b_8, b_{30}, b_{31} and b_{34} , therefore $df(b_1) = 5$.

TABLE 1
The Mutant Branches

No.	Mutant branch
1	$if((a > b)! = (\sim a > b))$
2	$if((a > b)! = (a > \sim b))$
3	$if((a > b)! = (a < b))$
4	$if((a > b)! = (a <= b))$
5	$if((a > b)! = (a == b))$
6	$if((a > b)! = (a! = b))$
7	$if((a > b)! = (a >= b))$
8	$if((a > b)! = !(a > b))$
9	$if((a > b)! = ((++a) > b))$
10	$if((a > b)! = ((--a) > b))$
11	$if((a > b)! = ((a++) > b))$
12	$if((a > b)! = ((a--) > b))$
13	$if((a > b)! = (a > (++b)))$
14	$if((a > b)! = (a > (--b)))$
15	$if((a > b)! = (a > (b + +)))$
16	$if((a > b)! = (a > (b - -)))$
17	$if(a! = -a)$
18	$if(a! = a + +)$
19	$if(a! = a - -)$
20	$if(a! = a + +)$
21	$if(a! = a - -)$
22	$if(a! = \sim a)$
23	$if(b! = -b)$
24	$if(b! = + + b)$
25	$if(b! = --b)$
26	$if(b! = b + +)$
27	$if(b! = b - -)$
28	$if(b! = \sim b)$
29	$if(max! = -max)$
30	$if(max! = + + max)$
31	$if(max! = --max)$
32	$if(max! = max + +)$
33	$if(max! = max - -)$
34	$if(max! = \sim max)$

TABLE 2
The Dominance Relations

Non-dominated mutant branch	Dominated mutant branches
b_1	$b_3, b_8, b_{30}, b_{31}, b_{34}$
b_2	$b_3, b_8, b_{30}, b_{31}, b_{34}$
b_6	$b_3, b_4, b_8, b_{24}, b_{25}, b_{28}, b_{30}, b_{31}, b_{34}$
b_9	$b_4, b_5, b_8, b_{11}, b_{14}, b_{16}, b_{24}, b_{25}, b_{28}, b_{30}, b_{31}, b_{34}$
b_{10}	$b_3, b_4, b_5, b_8, b_{12}, b_{13}, b_{15}, b_{18}, b_{19}, b_{22}, b_{30}, b_{31}, b_{34}$
b_{17}	$b_3, b_4, b_5, b_8, b_{18}, b_{19}, b_{22}, b_{29}, b_{30}, b_{31}, b_{34}$
b_{23}	$b_4, b_8, b_{24}, b_{25}, b_{28}, b_{29}, b_{30}, b_{31}, b_{34}$

4.2 Prioritization of Mutant Branches Based on the Dominance Degree

As mentioned in the previous subsection, $df(b_i)$ reflects the capability of a test datum covering b_i in covering other mutant branches. The larger the dominance degree, $df(b_i)$, the higher the defect detectability of the test datum that can cover b_i . So all the non-dominated mutant branches can be prioritized according to their dominance degrees. To fulfill this task, non-dominated mutant branches are first obtained based on the in-degree of each vertex in $DG(P')$. Then the priority of each non-dominated mutant branch is determined based on the dominance degree.

Suppose that the non-dominated mutant branch with the maximal dominance degree in B^{nd} is b_{i_1} . Then b_{i_1} is chosen and added to the prioritized branch set, denoted as $Priori_set$, i.e., $Priori_set = \langle b_{i_1} \rangle$.

A non-dominated mutant branch generally dominates at least one mutant branch. As a result, once b_{i_1} is chosen, we will also delete the mutants branches that b_{i_1} dominates in $DG(P')$. So the dominance degree of each remainder non-dominated mutant branch should change.

Generally, suppose that the k th non-dominated mutant branches, b_{i_k} is determined. Before determining the $(k+1)$ -th non-dominated mutant branch, vertex b_i and the vertices corresponding to the dominated mutant branches of b_i will be deleted from $DG(P')$.

The above process repeats until all non-dominated mutant branches are ordered. The specific steps of prioritizing mutant branches can be provided as follows.

- Step 1: Construct the dominance relation graph $DG(P')$ according to D^B .
- Step 2: Calculate the value of $df(b_i), b_i \in B^{nd}$, based on the out-degree of vertex b_i in $DG(P')$.

TABLE 3
The Dominance Degrees of Non-Dominated Mutant Branches

Non-dominated mutant branch	Dominance degree
b_1	5
b_2	5
b_6	9
b_9	12
b_{10}	13
b_{17}	11
b_{23}	9

- Step 3: Add the mutant branch, b_i , which has the maximal dominance degree in B^{nd} , to $Priori_set$.
- Step 4: Delete b_i and its dominated mutant branches from $DG(P')$.
- Step 5: Check whether $DG(P')$ is empty or not. If yes, output $Priori_set$; otherwise, go to Step 2.

Note. When more than one non-dominated mutant branch has the same dominance degree, as a common rule, we select the non-dominated mutant branch according to its position in P' .

For the example program in Fig. 1a, the non-dominated mutant branch with the maximal dominance degree is b_{10} ($df(b_{10}) = 13$), which is chosen and added to $Priori_set$. As a result, $Priori_set = \langle b_{10} \rangle$. Then $DG(P')$ is updated by deleting b_{10} and the mutant branches dominated by b_{10} from $DG(P')$. Fig. 2 shows the process of updating the dominance relation graph, where (a) is the graph after deleting b_{10} from $DG(P')$. In Fig. 2a, the maximal dominance degree is $df(b_9) = 6$, thus $Priori_set = \langle b_{10}, b_9 \rangle$. Fig. 2b is the dominance relation graph after deleting vertex b_9 and the vertices corresponding to its dominated mutant branches. Now $df(b_{17}) = df(b_{23}) = 1$. We select the non-dominated mutant branch according to its position in P' . As a result, b_{17} is chosen, and the dominance relation graph after deleting vertex b_{17} is shown as Fig. 2c. Since the remaining vertices are all with dominance degree 0 now, we successively select them according to their positions, i.e., b_1, b_2, b_6, b_{23} . According to the above steps, the prioritization of B^{nd} is $Priori_set = \langle b_{10}, b_9, b_{17}, b_1, b_2, b_6, b_{23} \rangle$.

5 ORDERLY GENERATION OF TEST DATA BASED ON PRIORITIES OF MUTANT BRANCHES

The above section proposes a method of prioritizing all non-dominated mutant branches according to their dominance

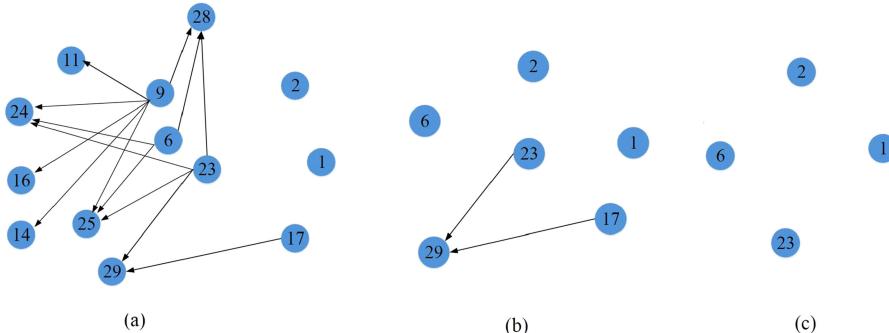


Fig. 2. The update of dominance relation graphs, where (a) is the graph after deleting b_{10} ; (b) is the graph after deleting b_9 ; and (c) is the graph after deleting b_{17} .

degrees. In this section, we focus on generating test data based on the priorities of mutants.

Generally, the earlier a defect is discovered in a program, the less harm it will bring. So we hope that the test data with higher defect detectabilities can be executed earlier. Therefore it is meaningful to prioritize test data according to their defect detectabilities. According to Section 4, a test datum that covers a non-dominated mutant branch must also cover all its dominated mutant branches. Thus a test datum that can cover a mutant branch with higher priority could cover more other mutant branches, and therefore have higher defect detectabilities. So the main idea of our test data generation method is that a mutant branch with the highest priority is first considered in generating test data.

First, the mutant branch with the highest priority in $Priori_set$ is selected. Then a test datum that can cover this branch is generated, denoted as t . Finally, t is added to the test suite, TCS , and delete all mutant branches that covered by t in $Priori_set$. The above process is repeated until $Priori_set$ is empty.

Because the focus of this paper is how to generate high-quality test data, rather than improving the efficiency of test data generation, we just use the random method to generate test data to cover the selected mutant branch. Of course, we can also use some more efficient algorithms here.

In addition, a test datum that kills one non-dominated mutant branch can also kill some other non-dominated mutant branches. Therefore, it is possible that the number of generated test data is less than the number of mutant branches.

The main steps of test data generation are provided as follows.

Step 1: Select the mutant branch with the highest priority in $Priori_set$, and generate a test datum that can cover this branch using the random method, denoted as t . Add t to the test suite, TCS .

Step 2: Delete all the mutant branches covered by t in $Priori_set$.

Step 3: Check whether $Priori_set$ is empty or not. If yes, output $Priori_set$ and TCS ; otherwise, go to Step 1.

According to the above method, in the example program, the mutant branch with the highest priority, i.e., b_{10} ($df(b_{10}) = 11$), is selected from $Priori_set = \langle b_{10}, b_9, b_{17}, b_1, b_2, b_6, b_{23} \rangle$. Then the test datum of b_{10} , " $t_1 = (0, -1)$ ", is generated by the random method. Besides b_{10} , t_1 also covers b_1 , b_2 and b_{23} in $Priori_set$. So we obtain $CSA = \{t_1\}$. By deleting b_{10}, b_1, b_2, b_{23} , we obtain the updated $Priori_set = \langle b_9, b_{17}, b_6 \rangle$. Following that, the mutant branch with the maximal dominance degree is b_9 . The test datum of b_9 , " $t_2 = (-10, -10)$ ", is generated by the same method. So we obtain $CSA = \{t_1, t_2\}$ and $Priori_set = \langle b_{17}, b_6 \rangle$. We generate test data for b_{17} , and obtain $t_3 = (-6, -7)$. Then $CSA = \{t_1, t_2, t_3\}$ and $Priori_set = \langle b_6 \rangle$. At last, we generate $t_4 = (0, 3)$ to cover b_6 . So we obtain $CSA = \{t_1, t_2, t_3, t_4\}$ and $Priori_set = \emptyset$.

The main steps of the proposed technique of our test data generation method for mutation testing can be shown in Fig. 3. First, the set of mutant branches is constructed by the transformation method for weak mutation testing. Then, the non-dominated mutant branch set is obtained according to the dominance relations of all mutant branches. Next, the

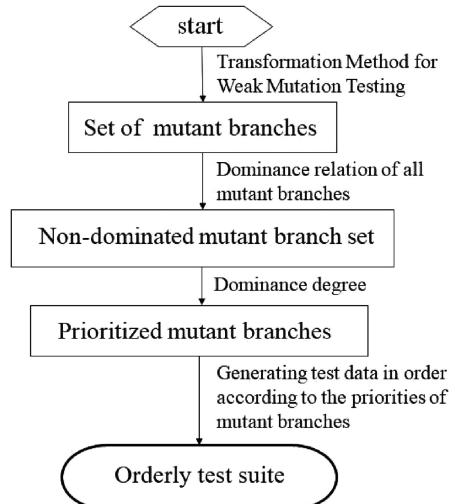


Fig. 3. Main steps of the proposed technique of the test data generation method for mutation testing.

non-dominated branches are prioritized based on their dominance degrees. Finally, the test data are generated in order according to the priorities of the non-dominated branches.

6 EXPERIMENTS

This section validates the effectiveness of the proposed method through a series of experiments. First, the research questions are raised. Then, the programs, mutation operators and the experimental design are given. Finally, the experimental results are provided and analyzed.

6.1 Research Questions

This paper proposes a method of test data generation via sorting mutant branches based on their dominance degrees for weak mutation testing, so as to improve the defect detectability of the generated test suite. The order of mutant branches determines the order of test data. Therefore, one natural research question that we concern is:

- RQ1: Whether the method of sorting mutant branches according to their dominance degrees is reasonable?

In order to answer RQ1, we select two other strategies to sort the mutant branches: one is the position-based strategy (PS), and the other is the random strategy (RS). PS prioritizes the mutant branches in B^{nd} according to their position-orders in P' , while RS randomly sorts all mutant branches in B^{nd} . Because our method sorts mutant branches according to their dominance degrees, we label it DS.

It should be noted that the difference between these three methods is mainly due to the different strategies of prioritizing mutant branches. For all strategies, we apply the same technique, i.e., the random method to generate test data to cover mutant branches.

We set up three sub-questions to compare the performance of these three methods from different perspectives.

- RQ1.1: How is the overall defect detectability of the generated test suite?

To answer this question, we use both weak mutation testing and strong mutation testing to measure the capability of

test suite in fault detection. The mutation scores of the generated test suites for weak mutation testing and strong mutation testing are calculated respectively. Suppose that the number of killed mutants is $\#Killed$, and that of non-equivalent mutants is $\#Non\text{-equivalent}$, then the mutation score:

$$MS = \frac{\#Killed}{\#Non\text{-equivalent}}$$

The higher the mutation score, the higher the defect detectability of the test suite. In order to distinguish, the mutation scores of weak mutation testing and strong mutation testing are denoted as MS_w and MS_s , respectively.

- *RQ1.2: How is the fault detection rate of the test suite?*

Mutation score can reflect the overall defect detectability of a test suite, but can not measure the effect of sorting test data. To illustrate the fault detection rate of the test suite, we adopt a commonly used index in regression testing, i.e., APFD, which is a measure of how rapidly a prioritized test suite detects faults [35], [59], [60]. The value of APFD is in the range of [0, 100](%), and a high APFD value means a fast (good) fault detection rate.

In addition to the fault detectability of the test suite, another concern of mutation testing is the cost. If the speed of the mutants being killed is improved by prioritization, the efficiency of the mutation testing will also be improved. So, our next concern is:

- *RQ1.3: How is the efficiency of generating test data?*

The cost of generating test data for these three methods is mainly caused by the execution of mutants which determines whether the mutants are killed or not. So we use the total execution times of mutants to reflect the cost of generating test data. Here the test data are generated under the criterion of strong mutation testing.

The main aim of this paper is to generate test data for mutation testing based on the priorities of mutation branches, so the second research problem is:

- *RQ2: How does our method compare with traditional mutation testing methods?*

In order to verify the performance of our method, we choose two evolutionary mutation testing methods for comparison, which are from references [51] and [52], respectively. The objective function of reference [51] is to maximize the mutation score. It investigated two evolutionary optimization methods: the genetic algorithm and the immune inspired algorithm. In the experiment, we choose the genetic algorithm to generate test data because of its popularity. All parameters of the genetic algorithm are set according to the instructions of Chapter 6 in reference [51]. The optimization model in Reference [52] includes three objective functions, i.e., the number of test cases and dead mutants, and the pairwise coverage. Evolutionary algorithms NSGA-II is selected to solve the optimization problem.

For mutation testing, one important problem is the fault detectability of the generated test data. So the first sub-problem is:

- *RQ2.1: Whether the test suite obtained by our method has higher defect detectability than two comparison methods?*

In addition, the efficiency of generating test data is another important issue for mutation testing. Therefore, we will compare the time consumption with the other two methods. Thus the second sub-problem is:

- *RQ2.2: Can our method improve the efficiency of test data generation comparing with the other two methods?*

Finally, because the test suite we generate are orderly, we also want to examine its rate of error detection. So we set up the third research question as follows:

- *RQ3: Can our method improve the rate of fault detection?*

There are many test data prioritization methods to improve the rate of fault detection of a given test suite in regress testing [54], [59], [61], [62], [63]. Our method is quite different from these existing ones. Test data prioritization method aims to sort an existing test suite, nevertheless our method prioritizes the test data while generating. So for comparison, we need first generate a test suite, and then prioritize it by some test data prioritization method.

To generate the test suite, first, we randomly select a mutant branch b in the mutant branch set B , and generate a test datum t that can cover b . Then all mutant branches that are covered by t are deleted from B . The above process continues until B is empty. In our method the test data covering a given mutant branch are generated by randomly searching in the input domain. For fairness, we still adopt the random method to generate test data here. By this way a test suite is generated.

Next, the test suite is sorted by traditional test data prioritization method. We adopt the Additional Strategy (AS) proposed in reference [59] to prioritize the test suite. In addition, because the random prioritization (RP) is often the lowest bound method, we also use it to prioritize the test suite. It is noted that in the first group of experiments, we use the random strategy (RS) to prioritize the mutant branches, but here we use the random prioritization method (RP) to directly prioritize the test suite.

6.2 Programs

Fifteen programs are selected for the experiments, listed in Table 4. Among these programs, J1, J2 and J4 are experimental programs from [17], J3 is from [64], and J5-J14 are from <http://commons.apache.org>. In addition, J1-J4 are benchmark programs which are well known in mutation testing and usually as example and experimental programs in many studies. J5-J15 are from open-source projects. All these programs are programmed in Java.

Table 4 lists the basic information for each program under test. We can see from Table 4 that the total numbers of LOCs and methods of these programs under test are 43262 and 3970, respectively. The biggest program is Colt with LOCs 31407.

6.3 Mutation Operators

The rule of grammatically changing a program is called a mutation operator. King and Offutt presented 22 kinds of mutation operators, each of which is represented by a three letter acronym [65]. The research of Offutt and Lee found that five classes of operators (ABS, AOR, LCR, ROR, and

TABLE 4
Basic Information of Experimental Programs

	Program	LOCs	Methods	Description
J1	Mid	26	1	Return the middle value of three integer
J2	TrashAndTakeOut	30	2	Not reported
J3	Triangle	36	1	Return the type of a triangle with three integer inputs
J4	Cal	50	2	Calculate the days between two dates in the same year
J5	Md5Crypt	107	7	Class from org.apache.commons.codec.digest
J6	WordUtils	173	12	Class from org.apache.commons.lang3.text
J7	UnixCrypt	311	12	Class from org.apache.commons.codec.digest
J8	DurationFormatUtils	365	9	Class from org.apache.commons.lang3.time
J9	HelpFormatter	416	39	Class from org.apache.commons.cli
J10	NumberUtils	636	47	Class from org.apache.commons.lang3.math
J11	Dfp	1702	113	Class from org.apache.commons.math3.dfp
J12	FastMath	2311	100	Class from org.apache.commons.math3.util
J13	StringUtils	2434	200	Class from org.apache.commons.lang3
J14	ArrayUtils	3258	319	Class from org.apache.commons.lang3
J15	Colt	31407	3106	Project for high performance scientific and technical computing
	Sum.	43262	3970	

UOI) are deemed to be sufficient to achieve almost full mutation coverage [66].

We use MuClipse to perform mutation testing for Java programs in this paper. There are total 15 kinds of Method-Level mutation operators in MuClipse. We use Eclipse to generate mutants by applying all these 15 kinds of mutation operators. Different mutation operators can be used for different types of statements. Detailed information of these 15 kinds of mutation operators is listed in Table 5, including their representation and description, etc.

6.4 Experimental Design

The experimental environment is as follows: Intel(R) Core (TM) i5-6200U @ 2.30GHz 2.30GHz, 12.00GB RAM, Windows 7 Operating System, and Eclipse SDK 4.2.2 with MuClipse 1.3. MuClipse is an Eclipse plug-in of MuJava, which can automatically generate and execute mutants against test data. The most steps of the experiments in this paper are realized by MuClipse.

In the experiments, mutants are generated after performing all these 15 Method-Level mutation operators of MuClipse on the programs in Table 4. To obtain the

dominance relations, mutant branches are first automatically constructed by parsing “mutation_log” created by MuClipse when generating mutants. Then, the new program, P' , is formed by inserting all the mutant branches into the original program, P . Finally, the dominance relations are identified by statically analyzing P' to obtain D^B and B^{nd} .

6.5 Results and Analysis

6.5.1 The Results and Analysis of the First Group of Experiments

- RQ1: Whether the method of sorting mutant branches according to their dominance degree is reasonable?

To illustrate the effectiveness of our method, the number of generated test data, the mutation scores of weak and strong mutation testing, and the APFD value are recorded. For each method, each program in Table 4 is independently run 20 times. The average results of numbers of generated test data and mutation scores are given in Tables 6, 7 and 8, respectively. Fig. 4 shows the APFD boxplot for each program.

- RQ1.1: How is the overall defect detectability of the generated test suite?

It can be observed from Table 6 that, (1) there are total 119748 mutants for these 15 programs, including 16076 equivalent ones. The proportion of equivalent mutants is 13.42 percent, which provides a valuable reference for mutation testing. It should be noted that if the types of mutation operators and programs are different, the results will be changed accordingly. (2) The number of non-equivalent mutants is 103672 ($= 119748 - 16076$), which can simulate 103672 injected faults. After dominance analysis, 5202 non-dominated mutant branches are obtained, which means the reduction ratio is $(103672 - 5202)/103673 = 94.98\%$. The results fully verify that the number of mutants can be greatly reduced according to their dominance relations, and thus the cost of mutation testing will decrease. (3) To cover all non-dominated mutant branches, PS generates 2248.1 test data, RS generates 2266.4 test data, whereas DS generates 2178.5 test data.

Although the number of test data using our method is smaller than those generated by PS and RS, the difference is

TABLE 5
15 Kinds of Method-Level Mutation Operators in MuClipse

Method level	Operator	description
Arithmetic	AORB	Arithmetic Operator Replacement(binary)
	AORS	Arithmetic Operator Replacement(binary)
	AOIU	Arithmetic Operator Insertion(unary)
	AOIS	Arithmetic Operator Insertion(short-cut)
	AODU	Arithmetic Operator Deletion(unary)
	AODS	Arithmetic Operator Deletion(short-cut)
Relational	ROR	Relational Operator Replacement
Conditional	COR	Conditional Operator Replacement
	COI	Conditional Operator Insertion
	COD	Conditional Operator Deletion
Shift	SOR	Shift Operator Replacement
Logical	LOR	Logical Operator Replacement
	LOI	Logical Operator Insertion
	LOD	Logical Operator Deletion
Assignment	ASRS	Assignment Operator Replacement(short-cut)

TABLE 6
The Mutants and Generated Test Data

Program	# Mutants	# Equivalent mutants	# Non-dominated mutant branches	# Test data		
				PS	RS	DS
J1	115	18	30	12.8	13.8	11.6
J2	111	29	5	5.0	5.0	5.0
J3	325	40	46	25.7	24.2	23.9
J4	316	43	43	19.6	19.0	18.0
J5	158	12	14	4.7	4.3	4.5
J6	243	34	36	5.9	6.0	5.8
J7	1097	209	116	6.7	6.5	6.6
J8	576	65	83	22.5	21.7	20.8
J9	291	42	91	21.3	20.1	19.9
J10	1406	212	293	85.6	82.9	79.3
J11	2133	258	154	73.4	79.4	69.7
J12	6486	967	284	84.4	82.1	81.8
J13	4955	603	982	216.1	203.7	195.8
J14	7551	795	660	357.8	358.2	352.4
J15	93985	12749	2365	1306.6	1339.5	1283.4
Sum.	119748	16076	5202	2248.1	2266.4	2178.5

not very obvious. This is because the mutant branches have been reduced according to their dominance relations. The remaining mutant branches are non-dominated ones. A test datum that can cover a non-dominated mutant branch must cover all its dominated mutant branches, but is not necessarily covering other non-dominated branches. That is to say, a test datum killing a non-dominated mutant branch with higher dominance degree is not necessarily covering more non-dominated branches. Therefore, our method does not necessarily reduce the number of test data. Nevertheless, our method still has slight advantage, indicating that the prioritization method according to dominance degree is reasonable.

In addition, our main purpose is to prioritize test data. So we will validate the error detectabilities of these test data by weak mutation testing and strong mutation testing, respectively. Table 7 provides the experimental results of test data generated by different methods for weak mutation testing.

We can learn from Table 7 that, (1) test data generated by PS cover 98287.1 out of 103672 mutant branches, and the average mutation score is 94.81 percent, (2) The average mutation score of RS is 95.00 percent that is slightly higher than that of PS, and (3) The average mutation score of DS is 96.32 percent that is slightly higher than that of RS. From the above observations, the overall performance of the test suite generated by our method is slightly better than those of the other two methods.

When generating test data, we aim to cover all non-dominated mutant branches. Theoretically, the test data that cover all non-dominated mutant branches can also cover all mutant branches, including the dominated ones. However, there are a small number of mutant branches that cannot be covered, for example, 5384.9 (103672 – 98287.1) mutant branches in Table 6 cannot be covered by test data generated using PS. The reason why not all mutant branches can be covered is provided as follows. We analyze

TABLE 7
Defect Detectability of the Generated Test Data by Weak Mutation Testing

Program	# Mutant branches	PS		RS		DS	
		Covered	MS_w (%)	Covered	MS_w (%)	Covered	MS_w (%)
J1	97	97.0	100.00	97.0	100.00	97.0	100.00
J2	82	82.0	100.00	82.0	100.00	82.0	100.00
J3	285	285.0	100.00	285.0	100.00	285.0	100.00
J4	273	273.0	100.00	273.0	100.00	273.0	100.00
J5	146	146.0	100.00	146.0	100.00	146.0	100.00
J6	209	209.0	100.00	209.0	100.00	209.0	100.00
J7	888	872.4	98.24	874.2	98.45	873.9	98.41
J8	511	507.7	99.36	508.1	99.44	508.1	99.44
J9	249	247.4	99.37	247.6	99.43	247.6	99.45
J10	1194	1175.8	98.31	1174.2	98.34	1177.5	98.62
J11	1875	1852.0	98.77	1846.0	98.45	1869.0	99.68
J12	5519	5328.0	96.54	5331.2	96.60	5402.8	97.89
J13	4352	4264.2	97.98	4283.1	98.42	4306.3	98.95
J14	6756	6569.5	97.24	6569.5	97.24	6569.5	97.24
J15	81236	76378.1	94.02	76564.9	94.25	77807.8	95.78
Sum.	103672	98287.1	94.81	98490.8	95.00	99854.5	96.32

TABLE 8
Defect Detectability of the Generated Test Data by Strong Mutation Testing

Program	PS		RS		DS	
	Killed	MS _s (%)	Killed	MS _s (%)	Killed	MS _s (%)
J1	97.0	100.00	97.0	100.00	97.0	100.00
J2	82.0	100.00	82.0	100.00	82.0	100.00
J3	274.5	96.32	274.3	96.25	275.1	96.53
J4	254.4	93.19	256.6	93.99	259.1	94.91
J5	146.0	100.00	146.0	100.00	146.0	100.00
J6	196.6	98.81	195.7	98.33	196.9	98.93
J7	840.3	94.63	843.2	94.95	843.7	95.01
J8	496.3	97.13	501.1	98.07	506.2	99.06
J9	239.1	96.05	240.2	96.45	241.4	96.93
J10	1135.8	95.13	1143.5	95.77	1145.5	95.94
J11	1712.2	91.32	1773.1	94.56	1829.7	97.58
J12	5221.4	94.61	5224.6	94.67	5348.8	96.92
J13	4050.9	93.08	4111.8	94.48	4277.1	98.28
J14	6438.1	95.29	6457.2	95.58	6504.4	96.28
J15	75005.2	92.33	75931.3	93.47	77369.2	95.24
Sum.	96189.8	92.78	97277.6	93.83	99122.1	95.61

the dominance relations among mutant branches by statical method, which may lead to some mistaken dominance relations. As a result, a few non-dominated mutant branches are classified as dominated ones, and thus reduced. In addition, although we deleted the equivalent mutants that are detected, there may still be some unidentified ones. These two kinds of mutants have no opportunity to be covered. In addition, it is also possible that some mutant branches are difficult to cover, so no corresponding test data is generated.

Table 8 is the experimental results of fault detection of test data generated by different methods for strong mutation criterion. From Table 8, we have two observations, (1) test data generated by PS and RS detect 96189.8 and 97277.6 injected faults with the mutation scores of 92.78 and 93.83 percent, respectively, and (2) the mutation score of test data generated by DS is 95.61 percent, which is higher than those generated by PS and RS. Moreover, our method has more advantages than the other two methods compared with the weak mutation testing.

The number of killed mutants in Table 8 is smaller than that of covered mutant branches in Table 7 by the same test data. For instance, test data generated by DS cover 99854.5 mutant branches, whereas they only kill 99122.1 mutants for strong mutation criterion. The reason can be provided as follows: the condition of killing a mutant under the criterion of weak mutation testing is weaker than that of strong mutation testing. As a result, some mutants killed for weak mutation testing may not be killed for strong mutation testing. But we also see that, on the whole, the test data generated by our method still has a good effect on strong mutation testing.

The mutation score can only reflect the overall performance of the test suite, but cannot reflect the advantage of test data prioritization. The purpose of our method is to make test data with higher defect detection capability have higher priority, so that it can be executed earlier. Thus more errors can be detected as early as possible. In view of this, we will evaluate the performance of the test suite by the fault detection rate.

- RQ1.2: How is the fault detection rate of the test suite?*

Fig. 4 shows the boxplots of the APFD values of these three methods for each program. We can see from the figure

TABLE 9
Numbers of Executing Mutants for Strong Mutation Testing

Program	Times			Comparison	
	PS	RS	DS	DS / PS(%)	DS / RS(%)
J1	416.5	392.8	340.2	81.68	86.61
J2	179.0	159.1	118.0	65.92	74.17
J3	3758.9	2294.7	1491.5	39.68	65.00
J4	1496.5	1226.3	800.1	53.46	65.25
J5	336.6	202.8	155.3	46.14	76.58
J6	511.6	375.0	262.3	51.27	69.95
J7	2361.8	1910.3	1098.3	46.50	57.49
J8	1316.9	1231.6	872.3	66.24	70.83
J9	1629.9	1083.0	719.3	44.13	66.42
J10	6483.7	5558.6	3987.1	61.49	71.73
J11	5823.9	4996.1	3701.5	63.56	74.09
J12	11612.4	11842.5	10340.9	89.05	87.32
J13	17585.7	14002.2	10349.0	58.85	73.91
J14	32163.7	18015.5	14155.6	44.01	78.57
J15	312188.3	229891.1	179226.5	57.41	77.96
Sum.	397865.4	293181.6	227617.9	57.21	77.64

that our method performs better than the other two methods. In fact, the average APFD values of PS and RS are 66.88 and 75.10 percent, respectively. While the average APFD value of DS is 82.94 percent. To be specific, our method improved the average APFD value of PS by 24.01 percent, and that of RS by 10.44 percent.

In order to make a more scientific analysis of the experimental results, we also used statistical tool *Spass* to conduct hypothesis test. The statistical results show that the APFD value of our method is significantly higher than those of the other two methods.

The experimental results fully show that the method of sorting mutant branches according to their dominance degrees is reasonable. Meanwhile, it is necessary to prioritize the test suite. If we run the test data randomly, we can kill the same mutants eventually, but the speed of killing mutants is different. In addition, we also see that the results of RS is better than those of PS. PS sorts the mutant branches according to their positions in the program. The experimental results verify that this sorting method has not any advantage. Instead, RS synthesizes the advantages and disadvantages of different sorting methods, so gets better results than PS.

- RQ1.3: How is the efficiency of generating test data?*

To compare the efficiency of different methods, the number of executing mutants for strong mutation testing by the generated test data is calculated. For a selected test datum, we employ it to execute all the remaining mutants to determine whether they can be killed or not. If yes, the mutants will be deleted from the mutant set. This process continues until all mutants are killed. For the same test suite, different orders of test data generally result in different numbers of executing mutants.

The experimental results are provided in Table 9, in which “Times” represents the numbers of executing mutants by different methods when killing all mutants, and “Comparison” means the comparison of Times between our method and PS or RS.

As shown in Table 9, DS has less times than PS and RS. When employing the 2248.1 test data (listed in Table 6) generated by PS to kill the 98287.1 mutants (listed in Table 8), the times of executing mutants is 397865.4. For test data

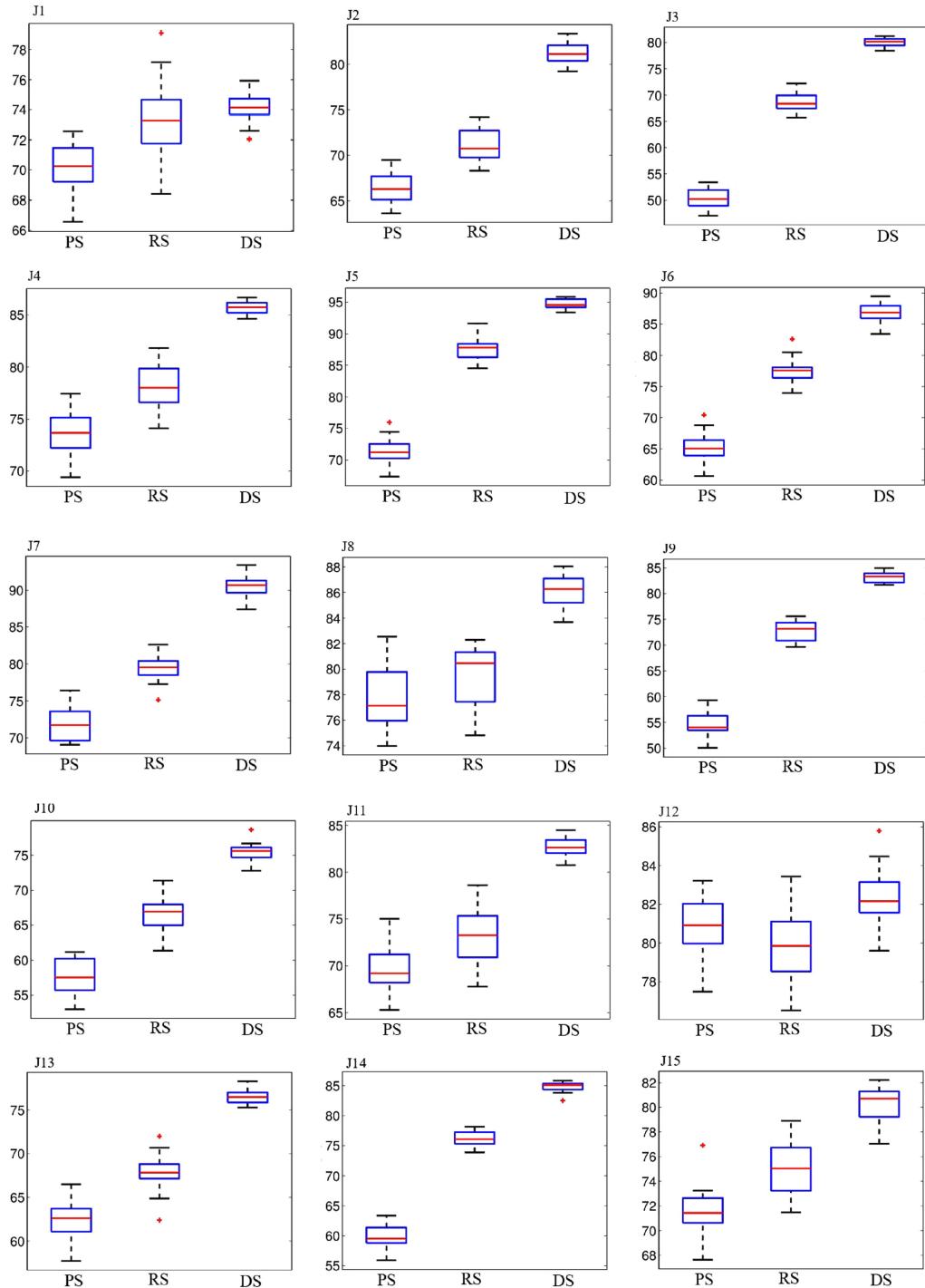


Fig. 4. APFD boxplots: by program, by technique (vertical axis is APFD score).

generated by RS, mutants are executed 227617.9 times when killing all the 98490.8 mutants. However, in order to kill all the 99854.5 mutants, mutants are executed 227617.9 times by test data generated by DS. The comparison shows that the times of executing mutants of DS is only 57.21 and 77.64 percent of that achieved by PS and RS, respectively, which means our method reduced time cost by 42.79 percent comparing with PS and 22.36 percent comparing with RS.

The significant reduction in the times of executing mutants for strong mutation testing by our method is benefit from the prioritization of test data based on the

dominance degree, such that the generated test data that can detect more faults are executed earlier. The above results show that sorting the mutant branches according to their dominance degrees can effectively improve the effect of generating test data. This is because test data that can kill mutant branches with higher dominance degrees have better defect detectabilities, so are possible to kill more other mutant branches.

In fact, the data presented in Table 9 should be highly negatively correlated with the value of APFD, as the higher APFD means the fewer times to reach the maximum value.

TABLE 10
Number of Test Data and Mutation Score of the Second Group of Experiments

Program	Number of test data			Mutation score $MS_s(\%)$		
	Our method	Method in [51]	Method in [52]	Our method	Method in [51]	Method in [52]
J1	11.6	20.0	15.3	100.00	100.00	100.00
J2	5.0	10.0	7.3	100.00	100.00	100.00
J3	23.9	30.0	29.1	96.53	95.31	95.81
J4	18.0	20.0	19.2	94.91	93.03	93.85
J5	4.5	10.0	6.5	100.00	99.79	99.99
J6	5.8	10.0	6.6	98.93	97.95	98.48
J7	6.6	10.0	8.1	95.01	94.17	94.91
J8	20.8	30.0	25.2	99.06	96.78	98.04
J9	19.9	20.0	20.3	96.93	95.62	95.95
J10	79.3	120.0	90.8	95.94	94.63	95.27
J11	69.7	100.0	82.9	97.58	91.29	94.08
J12	81.8	120.0	98.1	96.92	93.41	94.49
J13	195.8	230.0	217.9	98.28	92.93	94.22
J14	352.4	410.0	394.6	96.28	94.96	95.36
J15	1283.4	1630.0	1523.8	95.24	92.19	93.10
Sum.	2178.5	2770.0	2545.7 Ave.	97.44	95.47	96.24

We calculate the correlation coefficient between Times and APFD for these three methods. The result is -1 , which means that there is a strong negative correlation between Times and APFD.

Combining Tables 6, 7, 8, and 9 and Fig. 4, we can draw the following conclusion. (1) Compared with PS and RS, the proposed method generates a smaller number of test data, and achieves a higher mutation score for strong mutation testing, and (2) when detecting the injected faults, DS needs smaller times of executions of mutants, and obtains a higher APFD value. Therefore, the test suite generated by the proposed method are cost-efficient in detecting faults.

The above experimental results fully demonstrate that it is reasonable to sort the mutant branches according to their dominance degrees.

6.5.2 The Results and Analysis of the Second Group of Experiments

- RQ2: How does our method compare with traditional test data generation method?

In this group of experiments, the number of generated test data, the mutation scores of strong mutation testing, and the runtime are recorded. For each method, each program in Table 4 is independently run 20 times, and the average results are given in Table 10 and Fig. 5, respectively. It is noted that the runtime of our method includes the time of mutant branch reduction and sorting, and the time of generating test data. All the above processes can be realized automatically.

- RQ2.1: Whether the test suite obtained by our method has higher defect detectability than two comparison methods?

We can learn from Table 10 that, (1) for each program, the number of test data obtained by our method is always the least. The total number of test data generated by our method is 2178.5, while that of method in [51] and method [52] are 2770.0 and 2545.7, respectively, (2) the average mutation score of test data generated by our method is 97.44 percent, which is higher than those generated by method in [51] and method in

[52], and (3) Because method in [52] presents multiple optimization objectives, its result is better than that of method in [51].

The above results fully demonstrate that our method can obtain higher mutation score with less test data. This can not only reduce the cost of mutation testing, but also improve the quality of the test suite.

- RQ2.2: Can our method improve the efficiency of test data generation comparing with the other two methods?

Fig. 5 shows the time consumption for each method and program. We can see that the time consumption of our method is always less than that of the other two methods. The main reason is that by reducing mutant branches, the number of test objects is greatly reduced. In addition, the test data covering mutant branches with higher dominance degrees are generated preferentially in our method. These test data can kill more mutants. Once a test datum is generated, we will delete all mutants that it can kill. Therefore, prioritization of mutant branches can speed up the generation of test data.

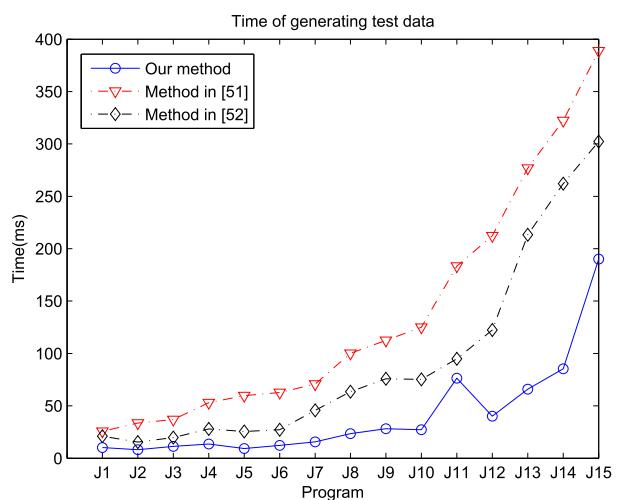


Fig. 5. Runtime of generating test data of the second group of experiments.

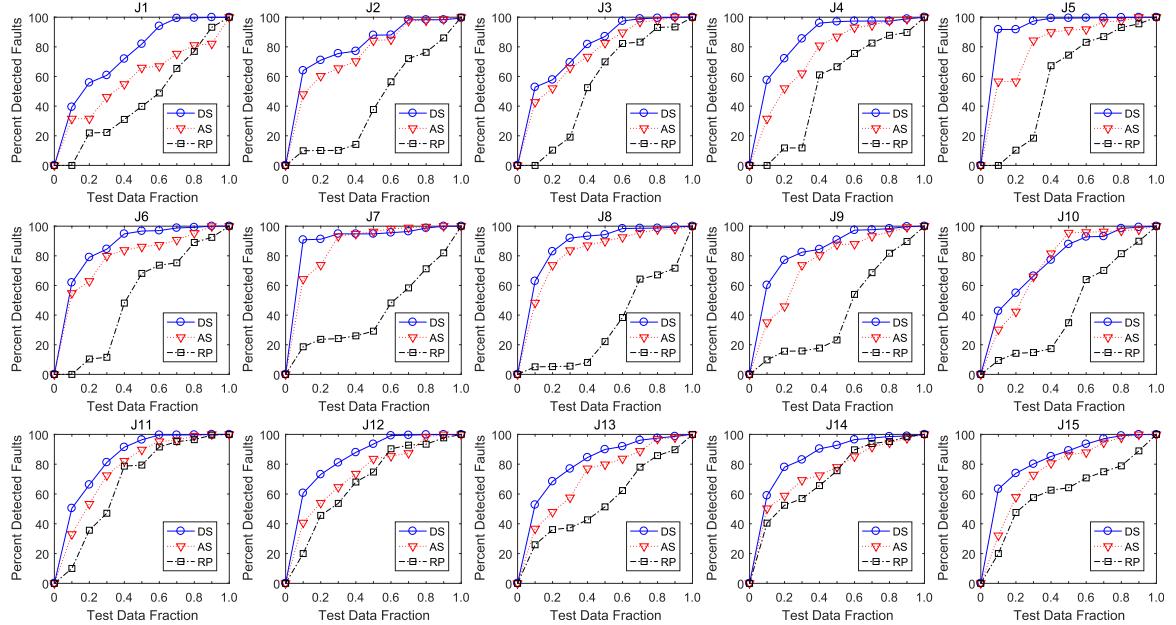


Fig. 6. APFD graph of test data generated by DS, AS and RP.

6.5.3 The Results and Analysis of the Third Group of Experiments

- RQ3: Can our method improve the rate of fault detection?

In order to observe the error detection rate of different test suite more intuitively, we use APFD graph to show the growth process of mutation score. Here, the test data increases by 20 percent each time. The results are shown as Fig. 6.

From Fig. 6 we see that the mutation score of our method always grows fastest, which indicates that the defect detection rate of the test suite generated by our method is the highest. In addition, the results of AS are better than that of RP. The experimental results further illustrate that prioritizing mutant branches according to their dominance degrees is reasonable.

The above three groups of experimental results show that the test data generation method proposed in this paper can not only improve the overall error detectability of the test suite, but also improve the rate of error detection.

7 THREATS TO THE VALIDITY

This section proposes main threats to the validity of the experiments and the methods of addressing them.

The first threat is the determination of equivalent mutants. In this paper, we use a semi-automatic method to judge whether a mutant is equivalent. For each mutant that is determined as non-equivalent, we will ensure that at least one test datum can kill it. So it is possible for a tester to identify a non-equivalent mutant as an equivalent one. To reduce this kind of threats, we will be as familiar with the structure and the function of each program as possible. In addition, we will use more test data to run the mutant, guaranteeing the equivalence of its function with the source program.

In addition, this paper sorts mutant branches based on their dominance degrees. So, the correctness of the dominant relation is the second major threat. We use statistical methods to determine dominance relations, which may

result in some errors. If test data are insufficient, a pseudo dominant relation may be identified as the dominant one. To reduce this kind of threats, we use decision-branch coverage criterion to generate test data, so as to ensure the accuracy of the statistical results.

In the experiments, we utilize the random method to generate test data to cover non-dominated mutant branches. Although the random method is highly intuitive and easy to implement, it is not an approach with the highest efficiency in the field of test data generation. However, the target of this paper is to improve the defect detectability of test data by prioritization, instead of the efficiency of test data generation. As a result, we will not put much effort at the means of generating test data. Nevertheless, we will explore more effective methods to fulfil this task in the future.

Additionally, although the preliminary experimental studies suggest that our approach is effective for the programs under test, the above conclusion cannot be directly generalized to more complicated real programs. So the proposed method needs further examination in practice.

8 CONCLUSION

Mutation testing can help to generate test data with a high capability in fault detection. However, not all test data have the same potency in detecting software faults. If test data are prioritized while generating for mutation testing, the defect detectability of the test suite can be further strengthened.

Considering this, we propose a method of test data generation for weak mutation testing via sorting mutant branches based on their dominance degrees. First, we obtain the non-dominated mutant branches by analyzing the dominance relation of mutant branches in the transformed program, and calculate the dominance degree of each non-dominated branch. Then, we prioritize all the non-dominated mutant branches in descending order by virtue of their dominance degrees. Finally, when generating test data, we select the mutant branches sequentially and the test data are also

prioritized according to the order of generation. The experimental results on 15 programs show that compared with other methods, the proposed test data generation method can not only improve the error detectability of the test suite, but also has higher efficiency.

We adopt the criterion of weak mutation testing to achieve the balance between costs and capabilities. In fact, all of these methods can be extended to the strong mutation testing, including the dominance relation between mutants, dominance degree, and prioritization of mutants. However, each mutant (a program) needs to be executed using the test data. As a result, the workload will increase. However, the defect detectability of the test suite will be strengthened accordingly. The experimental results show that the test data generated by our method also has a high mutation score for strong mutation testing.

In the experiments, weak mutation testing and strong mutation testing are employed to investigate the error detectabilities of test data via the mutation score. Given the fact that only traditional (method-level) mutation operators are utilized to the programs under test, the generated mutants may not represent all actual defects. Therefore, in the future work, we will apply more mutation operators to more practical programs, so as to measure the capability of test data in detecting actual defects.

ACKNOWLEDGMENTS

This work was jointly supported by National Natural Science Foundation of China (No. 61573362 and 61773841), National Key Research and Development Program of China (No. 2018YFB1003802), and Fundamental Research Funds for the Central Universities (No. 2020ZDPYMS40).

REFERENCES

- [1] G. J. Myers, C. Sandler, and T. Badgett, *The Art of Software Testing*, Hoboken, NJ, USA: Wiley, 2011.
- [2] T. T. Chekam, M. Papadakis, Y. L. Traon, and M. Harman, "An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 597–608.
- [3] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, 1978.
- [4] G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Softw. Eng.*, vol. SE-3, no. 4, pp. 279–290, Jul. 1977.
- [5] M. Kintis, M. Papadakis, A. Papadopoulos, E. Valvis, N. Malevris, and Y. L. Traon, "How effective are mutation testing tools? An empirical analysis of Java mutation testing tools with manual analysis and real faults," *Empir. Softw. Eng.*, no. 8, pp. 1–38, 2017.
- [6] P. G. Frankl, S. N. Weiss, and C. Hu, "All-uses versus mutation testing: An experimental comparison of effectiveness," *J. Syst. Softw.*, vol. 38, no. 3, pp. 235–253, 1997.
- [7] G. Fraser and A. Zeller, "Mutation-driven generation of unit tests and oracles," in *Proc. 19th Int. Symp. Softw. Testing Anal.*, 2010, pp. 147–158.
- [8] M. Harman, Y. Jia, and B. Langdon, "Strong higher order mutation-based test data generation," in *Proc. 8th Eur. Softw. Eng. Con. ACM SIGSOFT Symp. Foundations Softw. Eng.*, 2011, pp. 212–222.
- [9] D. Shin, S. Yoo, and D. H. Bae, "A theoretical and empirical study of diversity-aware mutation adequacy criterion," *IEEE Trans. Softw. Eng.*, vol. 44, no. 10, pp. 914–931, Oct. 2018.
- [10] H. Do and G. Rothermel, "On the use of mutation faults in empirical assessments of test case prioritization techniques," *IEEE Trans. Softw. Eng.*, vol. 32, no. 9, pp. 733–752, Sep. 2006.
- [11] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 435–445.
- [12] X. Cai and M. R. Lyu, "The effect of code coverage on fault detection under different testing profiles," *ACM SIGSOFT Softw. Eng. Notes*, vol. 30, no. 4, pp. 1–7, 2005.
- [13] W. E. Howden, "Weak mutation testing and completeness of test sets," *IEEE Trans. Softw. Eng.*, vol. SE-8, no. 4, pp. 371–379, Jul. 1982.
- [14] A. J. Offutt and S. D. Lee, "How strong is weak mutation," in *Proc. Symp. Testing Anal. Verification*, 1991, pp. 200–213.
- [15] X. Yao, M. Harman, and Y. Jia, "A study of equivalent and stubborn mutation operators using human analysis of equivalence," in *Proc. Int. Conf. Softw. Eng.*, 2014, pp. 919–930.
- [16] J. A. Jones and M. J. Harrold, "Test-suite reduction and prioritization for modified condition/decision coverage," *IEEE Trans. Softw. Eng.*, vol. 29, no. 3, pp. 195–209, Mar. 2003.
- [17] M. Papadakis and N. Malevris, "Automatically performing weak mutation with the aid of symbolic execution, concolic testing and search-based testing," *Softw. Quality J.*, vol. 19, no. 4, pp. 691–723, 2011.
- [18] Y. Jia and M. Harman, "An analysis and survey of the development of mutation testing," *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, Sep./Oct. 2011.
- [19] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. L. Traon, and M. Harman, "Mutation testing advances: An analysis and survey," *Advances Comput.*, vol. 112, pp. 275–378, 2018.
- [20] M. Papadakis and Y. Le Traon, "Mutation testing strategies using mutant classification," in *Proc. 28th Annu. ACM Symp. Appl. Comput.*, 2013, pp. 1223–1229.
- [21] A. J. Offutt, J. Pan, K. Tewary, and T. Zhang, "An experimental evaluation of data flow and mutation testing," *Softw.: Practice Experience*, vol. 26, no. 2, pp. 165–176, 1996.
- [22] M. Papadakis, D. Shin, S. Yoo, and D. Bae, "Are mutation scores correlated with real fault detection?: A large scale empirical study on the relationship between mutants and real faults," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 537–548.
- [23] A. Derezińska and A. Rudnik, "Evaluation of mutant sampling criteria in object-oriented mutation testing," in *Proc. Federated Conf. Comput. Sci. Inf. Syst.*, 2017, pp. 1315–1324.
- [24] T. A. Budd, *Mutation Analysis of Program Test Data*, New Haven, CT, USA: Yale University, 1980.
- [25] A. P. Mathur and W. E. Wong, "An empirical comparison of data flow and mutation-based test adequacy criteria," *Softw. Testing Verification Rel.*, vol. 4, no. 1, pp. 9–31, 1994.
- [26] J. Strug and B. Strug, "Using classification for cost reduction of applying mutation testing," in *Proc. Federated Conf. Comput. Sci. Inf. Syst.*, 2017, pp. 99–108.
- [27] A. J. Offutt, G. Rothermel, and C. Zapf, "An experimental evaluation of selective mutation," in *Proc. 15th Int. Conf. Softw. Eng.*, 1993, pp. 100–107.
- [28] Y. Jia and M. Harman, "Constructing subtle faults using higher order mutation testing," in *Proc. 8th IEEE Int. Working Conf. Source Code Anal. Manipulation*, 2008, pp. 249–258.
- [29] M. Polo, M. Piattini, and I. Garcia-Rodriguez, "Decreasing the cost of mutation testing with second-order mutants," *Softw. Testing Verification Rel.*, vol. 19, no. 2, pp. 111–131, 2009.
- [30] R. A. Demill and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, Sep. 1991.
- [31] J. R. Horgan and A. P. Mathur, *Weak Mutation is Probably Strong Mutation*, West Lafayette, IN, USA: Purdue University, 1993.
- [32] A. J. Offutt and S. D. Lee, "An empirical evaluation of weak mutation," *IEEE Trans. Softw. Eng.*, vol. 20, no. 5, pp. 337–344, May 1994.
- [33] D. Gong, G. Zhang, X. Yao, and F. Meng, "Mutant reduction based on dominance relation for weak mutation testing," *Inf. Softw. Technol.*, vol. 81, pp. 82–96, 2017.
- [34] G. Zhang, D. Gong, and X. Yao, "Mutation testing based on statistical dominance analysis," *Chinese J. Softw.*, vol. 26, no. 10, pp. 2504–2520, 2015.
- [35] P. McMinn, C. J. Wright, C. J. McCurdy, and G. M. Kapfhammer, "Automatic detection and removal of ineffective mutants for the mutation analysis of relational database schemas," *IEEE Trans. Softw. Eng.*, vol. 45, no. 5, pp. 427–463, May 2019.
- [36] L. Layman, L. A. Williams, and R. S. Amant, "MimEc: Intelligent user notification of faults in the eclipse IDE," in *Proc. Int. Workshop Cooperative Human Aspects Softw. Eng.*, 2008, pp. 73–76.
- [37] A. J. Offutt, Z. Jin, and J. Pan, "The dynamic domain reduction procedure for test data generation," *Softw. Practice. Experience*, vol. 29, no. 2, pp. 167–193, 1999.

- [38] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," *Commun. ACM*, vol. 59, no. 6, pp. 93–100, 2016.
- [39] N. Tillmann and J. De Halleux, "Pex-white box test generation for .NET," in *Proc. 2nd Int. Conf. Tests Proofs*, 2008, pp. 134–153.
- [40] X. Liu, G. Xu, L. Hu, X. Fu, and Y. Dong, "An approach for constraint-based test data generation in mutation testing," *J. Comput. Res. Develop.*, vol. 48, no. 4, pp. 617–626, 2011.
- [41] B. Korel, "Automated software test data generation," *IEEE Trans. Softw. Eng.*, vol. 16, no. 8, pp. 870–879, Aug. 1990.
- [42] D. Gong and X. Yao, "Testability transformation based on equivalence of target statements," *Neural Comput. Appl.*, vol. 21, no. 8, pp. 1871–1882, 2012.
- [43] A. Aleti, I. Moser, and L. Grunske, "Analysing the fitness landscape of search-based software testing problems," *Automated Softw. Eng.*, vol. 24, pp. 603–621, 2016.
- [44] D. S. Rodrigues, M. E. Delamaro, C. G. Correa, and F. L. S. Nunes, "Using genetic algorithms in test data generation: A critical systematic mapping," *ACM Comput. Surv.*, vol. 51, no. 2, 2018, Art. no. 41.
- [45] D. Gong, W. Zhang, and X. Yao, "Evolutionary generation of test data for many paths coverage based on grouping," *J. Syst. Softw.*, vol. 84, no. 12, pp. 2222–2233, 2011.
- [46] M. Harman, S. G. Kim, K. Lakhotia, P. McMinn, and S. Yoo, "Optimizing for the number of tests generated in search based test data generation with an application to the oracle cost problem," in *Proc. 3rd Int. Conf. Softw. Testing Verification Validation Workshops*, 2010, pp. 182–91.
- [47] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Trans. Softw. Eng.*, vol. 39, no. 2, pp. 276–291, Feb. 2013.
- [48] X. Yao, D. Gong, and G. Zhang, "Constrained multi-objective test data generation based on set evolution," *IET Softw.*, vol. 9, no. 4, pp. 103–108, 2015.
- [49] G. Zhang, D. Gong, and X. Yao, "Test case generation based on mutation analysis and set evolution," *Chinese J. Comput.*, vol. 38, no. 11, pp. 2318–2331, 2015.
- [50] M. Papadakis and N. Malevris, "Mutation based test case generation via a path selection strategy," *Inf. Softw. Technol.*, vol. 54, no. 9, pp. 915–932, 2012.
- [51] P. S. May, "Test data generation: Two evolutionary approaches to mutation testing," PhD Thesis, Dept. Comput. Sci., The Univ. Kent, London, U.K., 2007.
- [52] R. A. M. Filho and S. R. Vergilio, "A multi-objective test data generation approach for mutation testing of feature models," *J. Softw. Eng. Res. Develop.*, vol. 4, no. 1, pp. 1–29, 2016.
- [53] F. Carlos, M. Papadakis, V. Durelli, and E. M. Delamaro, "Test data generation techniques for mutation testing: A systematic mapping," in *Proc. 11th Workshop Exp. Softw. Eng.*, 2014, pp. 1–14.
- [54] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Softw. Testing Verification Rel.*, vol. 7, no. 3, pp. 165–192, 1997.
- [55] M. Kintis and N. Malevris, "Using data flow patterns for equivalent mutant detection," in *Proc. IEEE 7th Int. Conf. Softw. Testing Verification Validation Workshops*, 2014, pp. 196–205.
- [56] M. Kintis and N. Malevris, "Identifying more equivalent mutants via code similarity," in *Proc. Asia-Pacific Softw. Eng. Conf.*, 2013, pp. 180–188.
- [57] M. Papadakis, Y. L. Traon, and M. Delamaro, "Mitigating the effects of equivalent mutants with mutant classification strategies," *Sci. Comput. Program.*, vol. 95, no. P3, pp. 298–319, 2014.
- [58] M. Papadakis, Y. Jia, M. Harman, and Y. L. Traon, "Trivial compiler equivalence: A large scale empirical study of a simple, fast and effective equivalent mutant detection technique," in *Proc. Int. Conf. Softw. Eng.*, 2015, pp. 936–946.
- [59] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, Oct. 2001.
- [60] S. Elbaum, A. Malishevsky, and G. Rothermel, "Test case prioritization: A family of empirical studies," *IEEE Trans. Softw. Eng.*, vol. 28, no. 2, pp. 159–182, Feb. 2002.
- [61] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal, "A study of effective regression testing in practice," in *Proc. 8th Int. Symp. Softw. Rel. Eng.*, 1997, pp. 264–274.
- [62] Y. Lou, D. Hao, and L. Zhang, "Mutation-based test-case prioritization in software evolution," in *Proc. Int. Symp. Softw. Rel. Eng.*, 2015, pp. 46–57.
- [63] C. Hettiarachchi, H. Do, and B. Choi, "Risk-based test case prioritization using a fuzzy expert system," *Inf. Softw. Technol.*, vol. 69, no. C, pp. 1–15, 2016.
- [64] P. Ammann and J. Offutt, *Introduction to Software Testing*, Cambridge, U.K.: Cambridge Univ. Press, 2008.
- [65] K. N. King and A. J. Offutt, "A fortran language system for mutation-based software testing," *Softw.: Practice Experience*, vol. 21, no. 7, pp. 685–718, 1991.
- [66] A. J. Offutt and J. Pan, "Detecting equivalent mutants and the feasible path problem," in *Proc. Annu. Conf. Comput. Assurance*, 1996, pp. 224–236.



Xiangjuan Yao received the PhD degree in control theory and control engineering from the China University of Mining and Technology, Beijing, China, in 2011. She is a professor with the School of Mathematics, China University of Mining and Technology. Her main research interests include intelligence optimization and search based software testing.



Gongjie Zhang received the PhD degree in computer application technology from the China University of Mining and Technology, Beijing, China, in 2017. He is a lecturer with the School of Computer Science and Technology, Jiangsu Normal University. His main research interests include software testing and mutation testing.



Feng Pan received the bachelor's degree in electrical engineering and automation from the Xuzhou University of Technology, Xuzhou, Jiangsu, in 2017. Currently he is working toward the master's degree in the School of Information and Control Engineering, China University of Mining and Technology. His main research interests include search-based software testing.



Dunwei Gong (Member, IEEE) received the PhD degree in control theory and control engineering from the China University of Mining and Technology, Beijing, China, in 1999. He is a professor with the School of Information and Control Engineering, China University of Mining and Technology. His main research interests include intelligence optimization and control.



Changqing Wei received the bachelor's degree in applied mathematics from Taishan University, in 2017. Currently he is working toward the master's degree in the School of Mathematics, China University of Mining and Technology. His main research interest includes search-based software testing.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

Chatbot4QR: Interactive Query Refinement for Technical Question Retrieval

Neng Zhang^{ID}, Qiao Huang, Xin Xia^{ID}, Ying Zou, David Lo^{ID}, and Zhenchang Xing^{ID}

Abstract—Technical Q&A sites (e.g., Stack Overflow (SO)) are important resources for developers to search for knowledge about technical problems. Search engines provided in Q&A sites and information retrieval approaches (e.g., word embedding-based) have limited capabilities to retrieve relevant questions when queries are imprecisely specified, such as missing important technical details (e.g., the user's preferred programming languages). Although many automatic query expansion approaches have been proposed to improve the quality of queries by expanding queries with relevant terms, the information missed in a query is not identified. Moreover, without user involvement, the existing query expansion approaches may introduce unexpected terms and lead to undesired results. In this paper, we propose an interactive query refinement approach for question retrieval, named *Chatbot4QR*, which can assist users in recognizing and clarifying technical details missed in queries and thus retrieve more relevant questions for users. Chatbot4QR automatically detects missing technical details in a query and generates several clarification questions (CQs) to interact with the user to capture their overlooked technical details. To ensure the accuracy of CQs, we design a heuristic-based approach for CQ generation after building two kinds of technical knowledge bases: a manually categorized result of 1,841 technical tags in SO and the multiple version-frequency information of the tags. We develop a Chatbot4QR prototype that uses 1.88 million SO questions as the repository for question retrieval. To evaluate Chatbot4QR, we conduct six user studies with 25 participants on 50 experimental queries. The results are as follows. (1) On average 60.8 percent of the CQs generated for a query are useful for helping the participants recognize missing technical details. (2) Chatbot4QR can rapidly respond to the participants after receiving a query within approximately 1.3 seconds. (3) The refined queries contribute to retrieving more relevant SO questions than nine baseline approaches. For more than 70 percent of the participants who have preferred techniques on the query tasks, Chatbot4QR significantly outperforms the state-of-the-art word embedding-based retrieval approach with an improvement of at least 54.6 percent in terms of two measurements: Pre@k and NDCG@k. (4) For 48–88 percent of the assigned query tasks, the participants obtain more desired results after interacting with Chatbot4QR than directly searching from Web search engines (e.g., the SO search engine and Google) using the original queries.

Index Terms—Interactive query refinement, chatbot, question retrieval, stack overflow

1 INTRODUCTION

ONLINE technical Q&A sites, e.g., Stack Overflow¹ (SO) have emerged to serve as an open platform for knowledge sharing and acquisition [1], [2], [3]. The Q&A sites

allow users to ask technical questions or provide answers to questions asked by others. For example, SO, which has been gaining increasing popularity in the software programming domain, has accumulated more than 19 million questions and 28 million answers as of December 20, 2019.² The questions and answers in the Q&A sites form a huge resource pool for developers to search for and solve programming problems [4], [5].

Question retrieval is a key step for users to seek for knowledge from Q&A sites, as well as a requisite step for many automatic tasks, such as answer summarization [6], API recommendation [5], and code search [7]. Most of the Q&A sites provide a search engine for users to retrieve questions using a query. Typically, a query is simply a free form text that describes a technical problem [8]. The search engines mainly rely on traditional information retrieval (IR) techniques (e.g., keyword matching and term frequency-inverse document frequency (TF-IDF) [9]), which cannot retrieve semantically similar questions for queries due to the lexical gaps between questions and queries [5]. Recently, word embedding techniques (e.g., word2vec [10]) are widely used by the state-of-the-art question retrieval approaches to bridge the lexical gaps [3], [5], [6], [11]. Such word embedding-based approaches have shown to be able to achieve better performance than traditional IR techniques.

1. <https://stackoverflow.com/>

- Neng Zhang is with the College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang 310027, China, and the Ningbo Research Institute, Zhejiang University, Ningbo, Zhejiang 315100, China, and also with the PengCheng Laboratory, Shenzhen, Guangdong 518066, China. E-mail: nengzhang@zju.edu.cn.
- Qiao Huang is with the College of Computer Science and Technology, Zhejiang University, Hangzhou, Zhejiang 310027, China. E-mail: tkdsheep@zju.edu.cn.
- Xin Xia is with the Faculty of Information Technology, Monash University, Clayton, VIC 3800, Australia. E-mail: xin.xia@monash.edu.
- Ying Zou is with the Department of Electrical and Computer Engineering, Queen's University, Kingston, ON K7L 3N6, Canada. E-mail: ying.zou@queensu.ca.
- David Lo is with the School of Information Systems, Singapore Management University, Singapore 188065. E-mail: davidlo@smu.edu.sg.
- Zhenchang Xing is with the Research School of Computer Science, Australian National University, Canberra, ACT 0200, Australia. E-mail: zhenchang.xing@anu.edu.au.

Manuscript received 14 Feb. 2020; revised 25 July 2020; accepted 7 Aug. 2020. Date of publication 12 Aug. 2020; date of current version 18 Apr. 2022.

(Corresponding author: Xin Xia.)

Recommended for acceptance by G. Canfora.

Digital Object Identifier no. 10.1109/TSE.2020.3016006

2. <https://data.stackexchange.com/>

A practical issue overlooked by the existing search engines and question retrieval approaches is that *it is not an easy task for users to formulate a good query* [8], [12]. A survey conducted by Xu *et al.* [6] with 72 developers in two IT companies shows that a query could be imprecisely specified as users may not know the important keywords that the search engines expect. Rahman *et al.* [13] conducted an empirical study on code search using Google, which reveals that it is common for developers to miss some important technical details (e.g., programming languages and operating systems) in the initial queries. Consequently, *inaccurate queries will lead to unsatisfactory results of question retrieval*, as illustrated in the motivating example (see Section 2). To enhance the quality of queries, many automatic query expansion approaches have been proposed to expand queries with relevant terms extracted from a thesaurus (e.g., WordNet [14]) or similar resources [8], [12], [15]. Although the approaches can help retrieve relevant results, they are insufficient to obtain accurate results due to two reasons: (1) lack of techniques to identify the missing information in a query; and (2) queries expanded with unexpected terms without user involvement (as demonstrated in Section 6.1).

In this paper, we propose to interactively refine queries with users using a chatbot, named Chatbot4QR, in order to retrieve accurate technical questions from SO (or other Q&A sites) for users. Chatbot4QR focuses on accurately detecting the missing technical details in a query. It first retrieves an initial set of top- n SO questions similar to the query. To build a responsive chatbot, we adopt a two-phase approach to explore a large-scale repository of SO questions by combining Lucene [16] (an ultra-fast text search engine that implements BM25) and a word embedding-based approach. Next, several *clarification questions* (CQs)³ [17] are generated using a heuristic-based approach based on the technical SO tags appearing in the query and the top- n similar questions. To identify the types of technical details missed in a query for CQ generation, we build two technical knowledge bases: a manually categorized result of 1,841 SO tags and the multiple version-frequency information of the tags. Then, Chatbot4QR interacts with the user by prompting the CQs to the user and gathers the user's feedback (i.e., a set of technical tags and versions answered by the user to CQs). Finally, the user's feedback is used to adjust the initial similarities of SO questions (by assigning a weight coefficient η to the feedback), which results in a refined list of top- k similar questions for recommendation.

To evaluate Chatbot4QR, we collected 1,880,269 SO questions as a large-scale repository for implementing a Chatbot4QR prototype and testing the performance of question retrieval for queries. Since our evaluation process contains six user studies that require a great amount of manual efforts, we built 50 experimental queries from the titles of another 50 SO questions. We conducted the user studies with 25 recruited participants to investigate the following research questions:

RQ1. What are the proper settings of the parameters n and η in Chatbot4QR?

In Chatbot4QR, there are two key parameters: (1) n is the number of the initial top similar SO questions used for CQ

3. We define “clarification question” as a question that asks for some unclear information that is not given in the context of a query.

generation; and (2) η is the weight coefficient used for similarity adjustment of SO questions. We conducted a user study to evaluate the quality of CQs generated for queries by setting n from 5 to 50 and the quality of the top ten SO questions recommended by setting η from 0 to 1. Based on the results, we determine the proper settings of n and η as 15 and 0.2, respectively.

RQ2. How effective can Chatbot4QR generate CQs?

We conducted a user study to examine the usefulness of the CQs, i.e., whether the CQs can help the participants recognize the missing technical details in queries. The results show that on average, 60.8 percent of the generated CQs are useful for a query.

RQ3. Can Chatbot4QR retrieve more relevant SO questions than the state-of-the-art question retrieval and query expansion approaches?

We conducted a user study to evaluate the relevance of the top ten SO questions retrieved by Chatbot4QR and nine baselines, which apply two popular retrieval approaches (i.e., the Lucene search engine and a word embedding-based approach) and four query expansion approaches (see Section 5.3). The results show that Chatbot4QR outperforms the baselines by at least 54.6 percent in terms of two popular metrics: Pre@k and NDCG@k. For more than 70 percent of the participants, the improvement of Chatbot4QR over the baselines is statistically significant.

RQ4. How efficient is Chatbot4QR?

We recorded the execution time of Chatbot4QR during the experiments. Chatbot4QR takes approximately 1.3 seconds to start interaction with the user after receiving a query, which is efficient for practical uses.

RQ5. Can Chatbot4QR help obtain better results than using Web search engines alone?

We conducted four user studies (including the user study conducted in RQ3) for answering this research question. We asked the participants to search for their satisfied results for queries using Web search engines (e.g., the SO search engine and Google [18]) by applying the original queries and the refined queries after interacting with Chatbot4QR. Then, the participants evaluated the relevance of the search results. Finally, the participants chose their preferred results from three kinds of results: the two kinds of Web search results and the SO questions retrieved by Chatbot4QR. The results show that for 48-88 percent of the assigned query tasks, the participants obtain more desired results either from Chatbot4QR or by applying the queries reformulated after the interaction with Chatbot4QR to Web search engines.

Paper Contributions:

1. We propose a novel chatbot to assist users in refining queries. To the best of our knowledge, this is the first work that uses an interactive approach to improving the quality of queries for technical question retrieval.
2. We conduct six user studies to evaluate Chatbot4QR. The evaluation results show that Chatbot4QR can generate useful CQs to help users recognize and clarify the missing technical details in queries efficiently. The refined queries contribute to retrieving better results than using the existing question retrieval approaches and Web search engines alone.

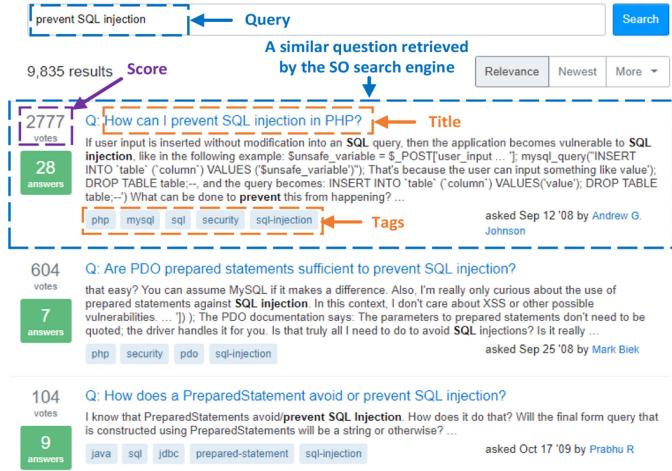


Fig. 1. The top three questions retrieved for a query by the SO search engine.

3. We release the source code of Chatbot4QR and the experimental dataset [19] to help other researchers replicate our experiments and extend our study.

Paper Organization. Section 2 describes a motivating example for our interactive query refinement approach. Section 3 presents the details of Chatbot4QR. Sections 4 and 5 report the experimental setup and results, respectively. Section 6 discusses several key aspects of Chatbot4QR and the threats to validity of our work. Section 7 reviews the related work. Section 8 concludes the paper and discusses future work.

2 MOTIVATING EXAMPLE

To motivate the use of an interactive approach to assisting users in refining queries, we illustrate the impact of a vague query on the quality of the questions retrieved by the SO search engine, and explain the key idea of Chatbot4QR.

Fig. 1 shows an annotated screenshot of the top three questions retrieved by the SO search engine for a query “*prevent SQL injection*”. Each retrieved question is tagged with a set of relevant technical terms, i.e., tags. For example, the first question is tagged with techniques, such as ‘*php*’ and ‘*mysql*’. Obviously, the query is vague due to missing some important technical details, e.g., the preferred programming languages and databases. Looking at the tags associated with each question, the first and the second questions are related to the programming language ‘*php*’, while the third question is related to ‘*java*’. Only the first question is explicitly tagged with the database ‘*mysql*’. Although the titles of the three questions are similar to the query, they are not satisfactory to every potential user as the users may have different technical background or programming context. For example, if a user prefers ‘*java*’, the top two questions are undesirable, while the third question may be suitable depending on the user’s preferred database. If a user is only familiar with the programming language ‘*c#*’, none of the three questions is relevant to the user. However, we find that there are similar questions tagged with ‘*c#*’ outside the top three returned results. To retrieve more desired questions for a user, it is necessary to assist the user in clarifying technical details that are not initially specified.

We propose Chatbot4QR to work interactively with users to improve the quality of queries. Chatbot4QR can heuristically generate several CQs to ask for two kinds of technical details: (1) the types of techniques widely adopted in software development, such as programming languages, databases, and libraries; and (2) the version of a technique as different versions of the technique may have substantial changes (e.g., ‘*python-2.x*’ and ‘*python-3.7*’), which may cause version-sensitive problems. In Fig. 1, there are two programming languages in the top three retrieved questions, but no programming language is specified in the query. Therefore, a CQ can be generated, e.g., “*What programming language, e.g., php or java, does your problem refer to?*”. Suppose that the user answers the CQ with ‘*java*’, since ‘*java*’ can have tags with multiple versions (e.g., ‘*java-7*’ and ‘*java-8*’), a new CQ is generated to ask for a specific version, e.g., “*Can you specify the version of ‘java’, e.g., 7 or 8?*”.

We strive to make our generated CQs easy for users to understand and answer, for the purpose of adoption in practice. Although a user needs to interact with our chatbot to answer the CQs, the amount of time spent is acceptable by the participants in our user studies (see Section 5.4). The feedback to CQs can help retrieve more relevant results and reduce the time required for the manual examination of undesirable results.

3 THE APPROACH

Fig. 2 gives an overview of our approach, which consists of two components: (1) *offline processing* which builds the Lucene index of SO questions, two language models (i.e., word2vec and word Inverse Document Frequency (IDF) vocabulary), and the categorization and version-frequency information of SO tags; and (2) *Chatbot4QR* which contains four main steps, namely ① retrieving the initial top-*n* similar SO questions for a query, ② generating CQs by detecting the missing technical details in the query, ③ interacting with the user by asking the CQs to assist them in refining the query, and ④ producing the final top-*k* recommended questions by adjusting the similarities of questions based on the user’s feedback to CQs.

3.1 Offline Processing

As shown in Fig. 2, Chatbot4QR needs to retrieve the initial top-*n* similar SO questions for a query before generating CQs. We build two language models, i.e., word2vec and word IDF vocabulary, to measure similarities between SO questions and queries, similar to the previous work [5], [6], [20]. The word2vec model is used to measure the semantic similarities among words; and the word IDF vocabulary measures the importance of words in the corpus. However, it is time-consuming to compute the semantic similarity between a query and each question in a large-scale repository, e.g., SO. To reduce the search space, we utilize Lucene to build the index for SO questions and retrieve a set of possibly similar questions before applying the word embedding-based approach. Moreover, we build two technical knowledge bases from SO tags for SO generation, i.e., the categorization and multiple version-frequency information of tags.

We use the text corpus of SO questions (including the titles, tags, and bodies) and the SO tags (including the descriptions

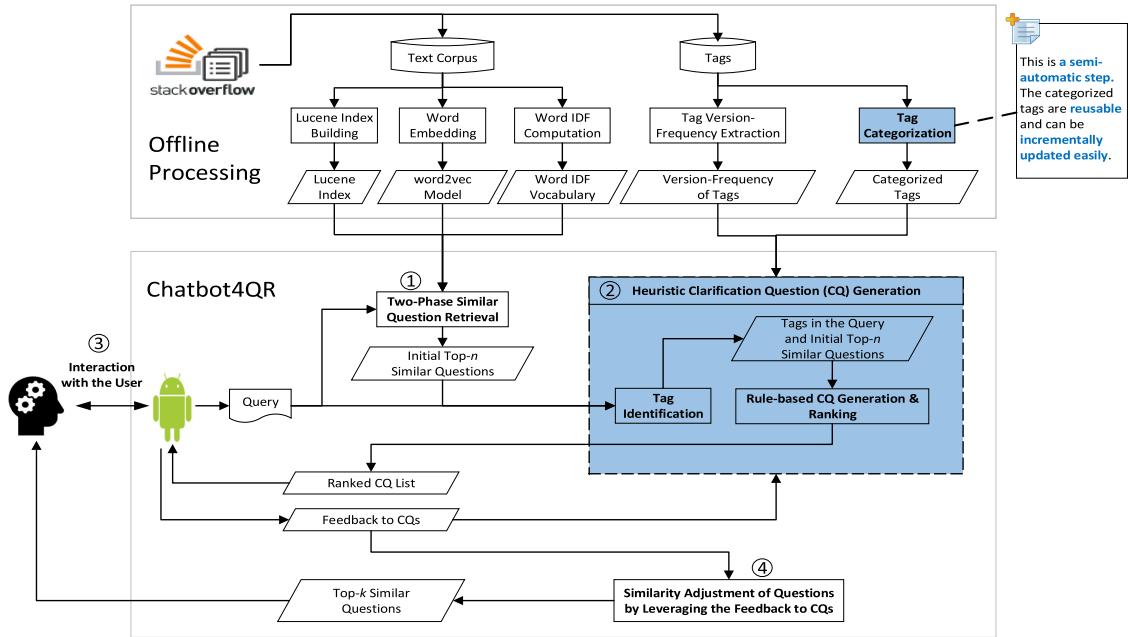


Fig. 2. An overview of our approach.

and synonyms crawled from the SO TagWiki [21] as the input of the offline processing component. Fig. 3 shows the description and ten synonyms of SO tag ‘java’. For questions, we remove the long code snippets enclosed in HTML tag `<pre>` from the bodies. We also reduce each word to its root form (aka. stemming) using the Porter stemmer in NLTK [22], a Python toolkit for natural language processing. As typical users would decide the relevance of a SO question to a query using the title and tags before checking the long body, we only consider the titles and tags of SO questions for question retrieval.

3.1.1 Lucene Index Building

We create a document for each SO question by gathering the title and tags, and build the index for all questions using Lucene.

3.1.2 Word Embedding

We apply the sentence tokenizer in NLTK to the titles and bodies of SO questions. Using the collected sentences, we

train a word2vec model using Gensim [23] with the default parameter setting.

3.1.3 Word IDF Computation

We remove the stopwords provided in NLTK from SO questions and build the word IDF vocabulary by computing the IDF metric of each word.

3.1.4 Tag Version-Frequency Extraction

Many SO tags have multiple versions due to the update of techniques; and each version has its own frequency. The frequency of a SO tag reflects the number of SO questions that have been tagged with it. For example, the tag ‘java’ has several versions, e.g., ‘7’ and ‘8’; and the frequencies of ‘java-7’ and ‘java-8’ are 2,861 and 18,302, respectively. We extract the multiple version-frequency information of SO tags for generating a particular kind of CQs that ask users to specify the version of a technique that they are interested in.

By examining the SO tags with versions, there are two common templates of a technique and the corresponding versions: (1) concatenate a technique and a version number by ‘-’, e.g., ‘java-8’ and ‘python-3.x’; and (2) append a version number to a technique, e.g., ‘sqlite3’ and ‘sslv2’. We extract the version numbers in SO tags using regular expressions. The extracted versions and the corresponding frequencies of each tag t are stored in a dictionary, denoted as $\text{ver_freqs}(t)$. For example, two elements in $\text{ver_freqs}(\text{'java'})$ are {‘7’: 2,861, ‘8’: 18,302}.

3.1.5 Tag Categorization

Categorizing SO tags to a set of meaningful types is critical for generating CQs for queries. Existing work that categorizes SO tags (e.g., [24], [25], [26]) is either incomplete or too fine-grained for our CQ generation. For example, only six types of SO tags were considered by Ye *et al.* [24] while neglecting some important types, e.g., *operating system* and *plugin*. Chen *et al.* [25] automatically generated 167 types

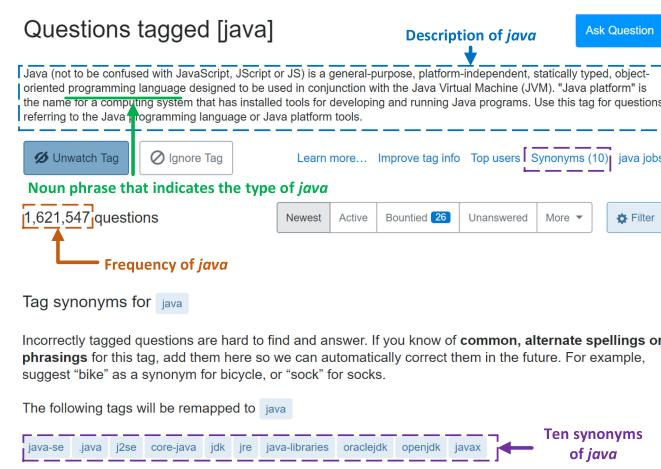


Fig. 3. An example SO tag “java”.

TABLE 1
Twenty Types of SO Tags

Type	#Tags Categorized to the Type	Example Tags
Library	418(36)	jquery, winforms, pandas, opencv, numpy
Framework	285(83)	.net, node.js, hibernate, spring, twitter-bootstrap
Tool	211(27)	maven, curl, gcc, ant, openssl
Class	171(3)	uitableview, listview, , httprequest, imageview, applet
Programming Language	96(31)	javascript, java, c#, python, c++
non-OS System	77(12)	wpf, git, svn, gps, hdf5
Platform	74(14)	azure, github, amazon-ec, google-cloud-platform, ibm-cloud
Service	65(2)	outlook, firebase-authentication, gmail, google-cloud-messaging, google-play-services
Technique	64(2)	jsp, reflection, proxy, nlp, deep-learning
Database	63(16)	mysql, sql-server, mongodb, oracle, neo4j
non-PL Language	60(4)	css, sql, wsdl, plsql, sparql, xml
Operating System	58(28)	android, ios, linux, windows, macos
Server	55(17)	tomcat, nginx, websphere, weblogic, jboss
Format	46(6)	json, xml, csv, pdf, jar
Plugin	44(6)	silverlight, jquery-validate, android-gradle, pydev, jstree
Environment	33(9)	eclipse, netbeans, visual-studio, webstorm, spyder, jdeveloper
Engine	32(2)	apache-spark, google-app-engine, elasticsearch, andengine, innodb
Design Pattern	15(0)	model-view-controller, singleton, adapter, inversion-of-control, decorator
Model/Algorithm	15(1)	dom, classification, rsa, svm, logistic-regression
Browser	15(6)	google-chrome, internet-explorer, firefox, safari, opera
Total	1,841(305)	

For each value in the “#Tags Categorized to the Type” column, the first number is the total number of tags categorized to the corresponding type; and the second number in the parenthesis is the number of tag synonyms categorized to the type.

where the analogous types (e.g., *library* and *module*) should be better merged. Incomplete types result in missing useful CQs, while fine-grained types lead to redundant CQs.

We strive to manually build a high-quality categorization of SO tags. However, the manual categorization of more than 50 thousands tags in SO is a cumbersome task. As the chances for querying the low frequency tags are low, we focused on the tags with the frequency of more than 1,000. As a result, we selected 4,158 tags. Despite the synonyms defined in SO, we also considered the tags marked with version numbers to be synonyms. For example, an extended synonyms set is {'java', 'java-se', 'jdk', 'java-7', 'java-8', ...}. We kept only the most frequent tag in each set of synonyms. Consequently, we obtained 3,772 tags. Then, we categorized the tags by using two iterations of a card sorting approach [27] as follows.

- *Build a set of types.* We observed that many SO tags have a noun phrase in the first description sentence to indicate the types of them, which are typically expressed in the form of “X is a/an noun phrase ...” [26]. As shown in Fig. 3, the first description sentence of ‘java’ shows that it is a programming language. We randomly sampled 349 tags from the 3,772 tags, which is a statistically significant sample size considering a confidence level of 95 percent and a confidence interval of 5. We used the Stanford POS (Part-of-Speech) tagger in NLTK to parse the first description sentence of each tag and extracted the first noun phrase behind the articles ‘a’ or ‘an’. The first two co-authors independently examined the noun phrases and built their own sets of types. Then, the two co-authors and a postdoc (who is not a co-author of the paper) together discussed the disagreements, eventually resulting in 20 types, as presented in Table 1. The two types ‘non-PL Language’ and ‘non-OS System’ respectively represent the non-programming languages (e.g., the query

language ‘sql’) and the non-operating systems, e.g., the version-control system ‘svn’.

- *Categorize tags based on types.* Based on the built types, the two co-authors further independently categorized each of the 3,772 tags. In total, 1,641 tags were initially categorized by at least one co-author. The uncategorized tags belong to the ignored types which are too general and are likely to be useless for CQ generation, e.g., *concept* and *keyword*. There were 215 tags with disagreement. The Fleiss Kappa [28] value of the two categorization results is 0.86, meaning an almost perfect agreement. The two co-authors and the postdoc worked together again to discuss the disagreements. Finally, they reached consensus on the categorization of 1,548 tags. The synonyms of a tag were then categorized to be the same type(s) as the tag. Table 1 presents the numbers of tags categorized to each of the 20 types, along with example tags. The numbers in parentheses are the numbers of synonyms categorized to the corresponding types. For example, “1,841 (305)” in the bottom row means that 1,841 tags including 305 synonyms are categorized to the 20 types. Note that the sum of the number of SO tags categorized to the 20 types is 1,897, which is larger than 1,841, since some tags are categorized to multiple types. For example, the tag ‘xml’ is categorized to the two types ‘non-PL Language’ and ‘Format’.

In our approach, tag categorization is a semi-automatic step. We took approximately 65 hours and nine hours to complete the two iterations, respectively. It is worth to mention that the categorized tags are reusable and can be incrementally updated easily. More specifically, when the frequencies of a number of (e.g., 50) uncategorized SO tags exceed 1,000, we can automatically extract the noun phrases from the first description sentences of the tags and then categorize them.

3.2 Chatbot4QR

Once the offline processing component is completed, the Chatbot4QR component shown in Fig. 2 is launched when a user submits a query. The query is processed first by two steps: stemming and stopword removal. Then, the four steps ①-④ in Fig. 2 are conducted to help the user refine the query if it has unclear technical details and recommend the top- k similar SO questions to the user.

3.2.1 Two-Phase Similar Question Retrieval

To detect if there are technical details left out in the query, denoted as q , we obtain the initial top- n SO questions similar to q using a two-phase approach. More specifically, we first use the Lucene search engine to retrieve a reduced set of N possible similar questions based on the Lucene index built for SO questions. Then, we use the word embedding-based approach adopted in the previous work [5], [6], [20] to retrieve the top- n semantically similar questions, denoted as $iSimQ_n(q)$, from the reduced set. To ensure that the majority of semantically similar questions can be covered by the reduced set, we set $N = 10,000$.

3.2.2 Heuristic Clarification Question Generation

Based on the initial top- n similar SO questions obtained for query q , we design a heuristic-based approach to automatically detecting the missing technical details in q and generate a set of CQs to help the user refine q interactively. The approach contains two sub-steps: tag identification and rule-based CQ generation & ranking.

Tag Identification. To generate CQs, we identify the SO tags appearing in q and the top- n similar questions $iSimQ_n(q)$. This is not an easy task due to the diverse appearances of SO tags in natural language texts. More specifically, every SO tag is lowercase and multiple tokens are concatenated by ‘-’, e.g., ‘sql-injection’. Moreover, SO tags can have versions, e.g., ‘java-8’. In contrast, the tags and versions can appear in a variety of forms in queries and the titles of SO questions, e.g., ‘java 8’, ‘Java8’, and ‘Java 8’s’. Before identifying tags in q and the similar questions, we transform each categorized SO tag by removing the possible version and replacing ‘-’ with a blank character. We also transform the original query as well as the original title and tags of each question in $iSimQ_n(q)$ by (1) converting them to lowercase, (2) replacing punctuations (except ‘#’ and ‘+’ as such symbols can be used as a part of a tag, e.g., ‘c#’ and ‘c++’) with a blank character, and (3) separating the possible version at the end of each token.

Using the transformed results described above, we identify the tags in q and each question $Q \in iSimQ_n(q)$. We also extract the version number, if it exists, of each tag identified from q . We filter out the version numbers of tags in the top- n similar questions as we directly use the version-frequency information of tags stored in ver_freqs (see Section 3.1.4) to generate CQs, which may help cover more similar questions outside the top- n . We group the two sets of tags identified from q and similar questions by the types of tags. The two grouped sets of tags are denoted as $typed_tags(q)$ and $typed_tags(iSimQ_n(q))$, respectively. Table 2 presents the grouped tags identified from the query and the top three SO

TABLE 2
Tags Identified From the Query “*Prevent SQL injection*” (q)
and the Top Three SO Questions Shown in Fig. 1

Type	$typed_tags(q)$	$typed_tags(iSimQ_3(q))$
Programming Language		{ php: ['7', '5.3'], java: ['8', '7'] }
non-PL Language	{ sql: [] }	{ sql: [] }
Database		{ mysql: ['2', '5.7'] }
Framework		{ .net: ['4.0', '3.5'] }
Library		{ jdbc: [] }
Class		{ pdo: [] }
Technique	{ sql-injection: [] }	{ sql-injection: [] }

questions shown in Fig. 1. In the table, we display the two most frequent versions of each tag in $typed_tags(iSimQ_3(q))$.

Rule-Based CQ Generation & Ranking. By comparing the two sets of identified tags, we generate three kinds of CQs for query q using the following three heuristic rules:

- *Rule 1 (version related CQ generation).* For each tag t in $typed_tags(q)$, if it has no specified version in q and it is a multi-version tag (i.e., $len(ver_freqs(t)) \geq 2$), a version related CQ is generated, such as “*Can you specify the version of t, e.g., v₁ or v₂?*”. v_1 and v_2 are the two most frequent versions of t in $ver_freqs(t)$, which are displayed to help the user better understand the CQ and provide feedback correctly.
- *Rule 2 (selection related CQ generation).* For each type $type$ in $typed_tags(iSimQ_n(q))$ but not in $typed_tags(q)$, if there are two or more tags included in the type, a selection related CQ is generated, such as “*What type, e.g., t₁ or t₂, are you using?*”. t_1 and t_2 are the two most frequent tags belonging to $type$ in $typed_tags(iSimQ_n(q))$. To make the selection related CQs sounded more natural, we customized the CQ expressions for the 20 types of SO tags, as shown in Table 3.
- *Rule 3 (confirmation related CQ generation).* For each type $type$ in $typed_tags(iSimQ_n(q))$ but not in $typed_tags(q)$, if only one tag t is included in the type, a confirmation related CQ is generated, such as “*Are you using t? (y/n), or some other types.*”.

Rule 3 is a special case of Rule 2. We distinguish them because a confirmation related CQ is more informative, implying that only one tag belonging to that type is identified from the initial top- n similar questions. If a user indeed uses the asked technique, they can easily answer the CQ with ‘y’.

In the subsequent interaction with the user, CQs that are more relevant to the query should be asked first. We rank the generated CQs by assigning a score to each CQ as follows:

- If cq is a version related CQ, its score is set to 1.0 because the tag asked in cq is explicitly specified by the user.
- If cq is a selection or confirmation related CQ, its score is calculated according to the similarities of the questions that contain any tags belonging to the $type$ asked in cq , i.e., $\frac{\sum_{Q \in iSimQ(type)} sim(q, Q)}{\sum_{Q \in iSimQ_n(q)} sim(q, Q)}$, where $iSimQ(type)$ denotes the subset of questions in $iSimQ_n(q)$ that contains a tag categorized to $type$; and $sim(q, Q)$ is the semantic similarity between q and Q .

TABLE 3
Customized CQ Expressions for the 20 Types of SO Tags Shown in Table 1

Type	Customized Selection Related CQ Expression for the Type
Library	Which library, e.g., X or Y, are you using?
Framework	If you are using a framework, e.g., X or Y, please specify:
Tool	Maybe you are using a tool, e.g., X or Y, for the problem. If so, what is it?
Class	Are you using a specific class, e.g., X or Y? Please input it:
Programming Language	What programming language, e.g., X or Y, does your problem refer to?
non-OS System	Apart from the operating system (OS), is there a non-OS, e.g., X or Y, used for your problem?
Platform	Tell me a possible platform, e.g., X or Y, you are using:
Service	For the problem, if you are using a service, e.g., X or Y, please provide:
Technique	Please give a possible technique, e.g., X or Y, you might use for the problem:
Database	I want to know whether you are using a database, e.g., X or Y. Can you provide it?
non-PL Language	Despite the programming language (PL), are you using any non-PL languages, e.g., X or Y?
Operating System	Could you provide an operating system, e.g., X or Y?
Server	Which server, e.g., X or Y, does your program intend to run on?
Format	What is the format, e.g., X or Y, of the data/file you are handling?
Plugin	I am wondering if you are using a plugin, e.g., X or Y. Specify it if there is one:
Environment	Would you like to provide an environment, e.g., X or Y, you are using?
Engine	Give me a possible engine, e.g., X or Y, that you need to execute your program:
Design Pattern	Any design patterns, e.g., X or Y, used for your problem?
Model/Algorithm	Do you use a model or an algorithm, e.g., X or Y? Please specify:
Browser	Your problem may be related to a browser, e.g., X or Y. Can you specify it?

In each CQ expression, "X" and "Y" are two example SO tags of the corresponding type that appear in the initial top similar SO questions retrieved for a query.

3.2.3 Interaction With the User

Based on the ranked CQ list, Chatbot4QR interacts with the user by asking each CQ one-by-one and gathers the user's feedback. Fig. 4 illustrates the chat process with a user by submitting the query shown in Fig. 1 to Chatbot4QR. The five CQs without a dotted frame are initially generated based on the top 15 (the proper value of the parameter n in Chatbot4QR, as evaluated in Section 5.1) similar SO questions retrieved using the two-phase approach. As shown in Fig. 4, Chatbot4QR has the following features:

1. It can interact with the user in multiple rounds.
2. It can generate new version related CQs to ask for the versions of the multi-version tags (e.g., 'java' and 'mysql') that are answered by the user to confirmation or selection related CQs.
3. To be user-friendly, it allows the user to skip any CQs that might be not useful or difficult to answer by pressing $\langle Enter \rangle$, or to terminate the interaction anytime.

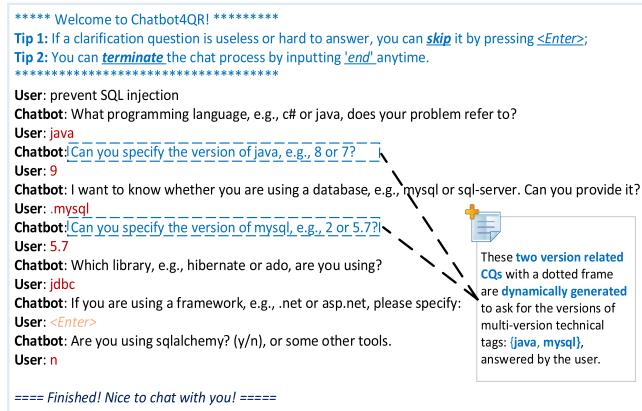


Fig. 4. The interaction with a user for the query shown in Fig. 1.

3.2.4 Similarity Adjustment of Questions

We distinguish two kinds of a user's feedback to the CQs of query q : (1) *positive feedback*, denoted as $pfb(q)$, which includes the tags and versions answered by the user; and (2) *negative feedback*, denoted as $nfb(q)$, which includes the tags involved in the confirmation related CQs whose answers are explicitly 'n' (means that the user does not use the asked technique). We do not consider the possible negative feedback to CQs since the user's rationale is unknown. For example, if a confirmation related CQ has no answer (i.e., the CQ was skipped), it is not certain that the user does not use the asked technique. It might be the reason that users are not familiar with the programming context and thus have difficulties in answering. In Fig. 4, the positive and negative feedback given by the user are

- $pfb(q) = \{\text{'java 9'}, \text{'mysql 5.7'}, \text{'jdbc'}\}$,
- $nfb(q) = \{\text{'sqlalchemy'}\}$.

Using the two kinds of feedback, we adjust the semantic similarity between q and each question Q in the reduced set retrieved using Lucene as

$$\begin{aligned} sim(q, Q) = sim(q, Q) \times \left(1 + \eta \times \left(\sum_{e \in pfb(q)} md(e, Q) \right. \right. \\ \left. \left. - \sum_{e \in nfb(q)} md(e, Q) \right) \right), \end{aligned} \quad (1)$$

where $md(e, Q)$ measures the degree that Q matches the tag and its possible version in the feedback element $e = (t, v)$ (where t is the tag and v is the version), e.g., 'java 9'. The coefficient $\eta \in [0, 1]$ is used to weight the importance of the technical feedback. A larger η means to put more weight on the feedback. More specifically, $\eta = 0$ ignores the feedback, while $\eta = 1$ means that the feedback has the same importance as the original query. In this work, we define $md(e, Q)$ as

$$md(e, Q) = \begin{cases} 1.5, & \text{if } e.v \text{ exists and both } e.t \text{ and } e.v \text{ are matched by } Q \\ 1.0, & \text{if only } e.t \text{ is matched by } Q \\ 0, & \text{otherwise.} \end{cases} \quad (2)$$

The idea of Eq. (1) is to increase (resp. decrease) the semantic similarity of Q according to the amount of positive (resp. negative) feedback matched by Q . A refined list of the top- k similar questions is produced based on the adjusted similarities and recommended to the user.

4 EXPERIMENTAL SETUP

Chatbot4QR is an interactive approach that considers users' personalized technical background and programming context to retrieve desired questions. We design a series of user studies to evaluate Chatbot4QR. In this section, we describe the experimental setup of our user studies. Our experimental environment is a laptop with Intel Core i5-8300H CPU, 16G RAM, and Windows 10 Operating System.

4.1 Data Collection and Prototype Implementation

We downloaded the official SO data dump released in September, 2018 and built a repository of 1,880,269 SO questions that are tagged with six popular programming language tags: {'javascript', 'java', 'c#', 'python', 'c++', 'c'}. To ensure the quality of our repository, every question needs to have an accepted answer and a positive score (i.e., the votes of a question shown in Fig. 1). Using the collected questions, we built a text corpus by removing the long code snippets in the bodies of questions and processing all words in the title, tags, and body of each question using the Porter stemmer in NLTK. We then trained a word2vec model using Gensim (with the default parameter setting), computed the word IDF vocabulary, and built the Lucene index for all questions. Moreover, we crawled the descriptions and synonyms of 55,661 SO tags from the TagWiki, and built two technical knowledge bases: the categorization and version-frequency information of tags. The details of these offline steps are described in Section 3.1.

As described in Section 3.2, Chatbot4QR has three parameters: (1) n is the number of the initial top similar SO questions used for CQ generation; (2) η is the weight coefficient of users' technical feedback in Eq. (1) used to adjust the similarities of questions; and (3) k is the number of the top similar questions recommended to the user. We determined the proper settings of n and η as 15 and 0.2, respectively, by conducting a user study (see Section 5.1). Considering the fact that users are likely to be only interested in the top ranked results [29], we set $k = 10$ in our prototype implementation, similar to the previous work [30], [31], [32].

4.2 Experimental Query Selection

In the existing research work on information retrieval from SO [5], [6], [8], [20], [32], [33], the experimental queries used for evaluation are built from the titles of SO questions selected according to some criteria, of which two commonly used criteria are: (1) the questions should have accepted answers; and (2) the scores of questions should be higher than a threshold (e.g., 5). This is suitable because the title of a SO question is a simple text that briefly describes a technical problem that a developer wants help for. We built 50

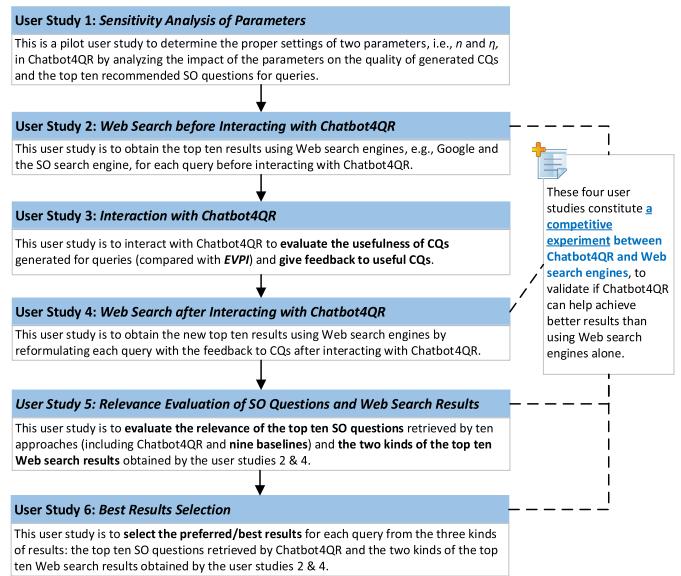


Fig. 5. The flow of our six user studies.

experimental queries from the titles of SO questions outside our repository. We chose 50 queries due to two reasons: (1) it is a relatively common number of experimental queries used in the previous work [30], [32], [33]; and (2) our user studies contain six consecutive stages (see Fig. 5) which require a great amount of manual efforts.

Our experimental queries were selected as follows. We first collected the popular SO questions which are tagged with the aforementioned six programming languages but not in our repository using two criteria: (1) the view count should be no less than 1,000; and (2) the score should be at least five. Then, we randomly selected 50 queries from the titles of the collected questions. For each query, we further ensured that there is no duplicate question contained in the repository, similar to the previous work [6]. As listed in Table 4, the 50 queries cover a variety of technical problems, which involve different techniques, e.g., programming languages, databases, and deep learning libraries. Some of the queries are simple, e.g., "Reading a line using scanf()" while others are complex, e.g., "How to sort dictionaries of objects by attribute value in python?". Moreover, there are queries expressed with technical terms, e.g., "Killing thread after some specified time limit in Java", while some queries have no specified technique, e.g., "Recognize numbers in images". The diversity of the queries can improve the generality of our experiment results.

We processed the queries by performing stemming and stop word removal. Based on the Lucene index built for SO questions, we retrieved the top $N=10,000$ similar questions for each query using the Lucene search engine. We then re-ranked the 10,000 questions by measuring semantic similarities between the questions and the query using the word embedding-based approach adopted in the previous work [5], [6].

4.3 Participant Recruitment

To conduct our user studies shown in Fig. 5 for evaluating Chatbot4QR, we recruited participants through the mailing lists of the first and the third co-authors' colleges. In the e-mail, we briefly introduced Chatbot4QR and our evaluation plan, and asked a few questions about the programming

TABLE 4
Fifty Experimental Queries

No.	SO Question ID	Experimental Query (the Title of the SO Question)
1	17294809	Reading a line using scanf()
2	423006	How do I generate points that match a histogram?
3	15389110	How to convert Json String with dynamic fields to Object?
4	2733356	Killing thread after some specified time limit in Java
5	20458401	How to insert multiple rows into database using hibernate?
6	15626686	Better way to parse xml
7	2592985	ArrayList shallow copy iterate or clone()
8	6262084	how to slide image with finger touch in android?
9	5108926	how to encrypt data using AES in Java
10	7918593	How can I determine the week number of a certain date?
11	90838	How can I detect the encoding/codepage of a text file
12	12981190	How to make a static variable thread-safe
13	22173762	Check if two Lists are equal
14	2411893	Recognize numbers in images
15	3561202	Check If Instance Of A Type
16	8702165	How to clone (and restore) a DOM subtree
17	11182924	How to check if JavaScript object is JSON
18	30950032	How can I run multiple NPM scripts in parallel?
19	531998	Set path programmatically
20	28052395	Find whether a 2d matrix is subset of another 2d matrix
21	14268053	Most efficient way to calculate pairwise similarity of 250k lists
22	5450055	How can I improve my INSERT statement performance?
23	3548495	Download, extract and read a gzip file in Python
24	44274701	Make predictions using a tensorflow graph from a keras model
25	4869189	How to transpose data in a csv file?
26	215557	Most elegant way to implement a circular list (FIFO)
27	1558402	Memory usage of current process in C
28	1805518	Replacing all non-alphanumeric characters with empty strings
29	6390339	How to query XML that has XSL in Java with XPath?
30	2676719	Calculating the angle between two points
31	10975913	How to make a new list with a property of an object which is in another list
32	8892073	how to compare webpages structure (dom) similarity in java?
33	9963331	java : How to know how many Threads have been Created and running?
34	891345	Get a screenshot of a specific application
35	8910840	Using LINQ to extract ints from a list of strings
36	21461102	Converting Html Table to JSON
37	6773550	Get id of div from its class name
38	2617515	Recommendation for a HTTP parsing library in C/C++
39	1323824	how to read numbers from an ascii file (C++)
40	3823921	Convert big endian to little endian when reading from a binary file
41	13340955	Convert linear Array to 2D Matrix
42	1623849	Fastest way to zero out low values in array?
43	32109319	How to implement the ReLU function in Numpy
44	14472795	How do I sort a list of datetime or date objects?
45	5741518	Reading each column from csv file
46	22722079	Choosing elements from python list based on probability
47	7891697	Numpy Adding two vectors with different sizes
48	8022530	Python check for valid email address?
49	459981	BeautifulSoup - modifying all links in a piece of HTML?
50	10052912	How to sort dictionaries of objects by attribute value in python?

background. We received 25 responses that agreed to join our user studies. The number of our participants is close to the numbers of participants used to conduct user studies in the previous work [34], [35], which is considered to be sufficient for our user studies. Table 5 presents the profiles of the 25 participants. Since some of the participants have working experience in companies like Hengtian,⁴ the "#Years of Programming Experience" column shows the years

4. Hengtian is an outsourcing company in China that has more than 2,000 employees and mainly does outsourcing projects for American and European corporations.

TABLE 5
Profiles of 25 Participants

Participant	Familiar Programming Languages	#Years of Programming Experience
P1	python	3.5
P2	python	4.0
P3	java, python	8.0
P4	java, python	6.0
P5	java	4.5
P6	python	7.5
P7	java, python	4.0
P8	java, python, c	10.0
P9	java, python	5.5
P10	java	3.5
P11	java	3.0
P12	java, c#	2.0
P13	java, python, matlab	8.5
P14	java, python, c#	6.5
P15	java	3.5
P16	java, python	4.0
P17	java, python, c++	8.0
P18	java, python	8.5
P19	java, javascript	2.5
P20	java	3.5
P21	java, python	8.0
P22	java, javascript	7.0
P23	java, python	11.0
P24	java, python	6.5
P25	python, c, c++	9.0

of both working experience and student experience in programming for each participant. We observe that the participants have diverse familiar programming languages. Some of them are only familiar with Java or Python, while others have multiple familiar languages. Moreover, there are notable differences in the participants' years of programming experience (from 2 to 11 years) with an average of 5.92 years.

We asked the participants to review the experimental queries and no participant reported being unable to understand the queries after we allowed them to search for the definitions of unfamiliar technical terms (e.g., 'LINQ' and 'NPM') online. As listed in Table 4, our queries cover multiple programming languages, e.g., Java, Python, and C++. We did not guarantee that the participants are familiar with all the programming languages because *Chatbot4QR* intends to help both experienced developers and novices. The diversity of the participants' technical background can help improve the generality of our experiment results.

4.4 Research Questions and the Allocation of Queries to Participants for User Studies

As shown in Fig. 5, we designed six user studies to investigate the following research questions:

RQ1. What are the proper settings of the parameters n and η in Chatbot4QR?

RQ2. How effective can Chatbot4QR generate CQs?

RQ3. Can Chatbot4QR retrieve more relevant SO questions than the state-of-the-art question retrieval and query expansion approaches?

RQ4. How efficient is Chatbot4QR?

RQ5. Can Chatbot4QR help obtain better results than using Web search engines alone?

It is a cumbersome task for a participant to conduct the five user studies 2-6 (the user study 1 is a pilot user study)

TABLE 6
The Allocation of Queries to Participants for the Six User Studies Shown in Fig. 5;
and the Research Questions Investigated by Each User Study

User Study No.	Investigated Research Questions	Allocation of Queries to Participants
1	RQ1	Ten queries randomly selected from Q1-Q50 are allocated to PG0
2	RQ5	
3	RQ2, RQ4	
4	RQ5	
5	RQ3, RQ4, RQ5	QG1 are allocated to PG1 , QG2 are allocated to PG2
6	RQ5	

.“RQ1-RQ5” are the five research questions. “PG0-PG2” are three participant groups. “Q1-Q50” are the 50 experimental queries. “QG1” and “QG2” are two query groups.

for all the 50 experimental queries. Therefore, we allocated 25 queries to each participant as follows.

We randomly divided the 50 queries into two equally sized groups: *QG1* and *QG2*. The queries in *QG1* are indexed by Q1-Q25 and those in *QG2* are indexed by Q26-Q50, as shown in Table 4. From the 25 participants, we first randomly selected five participants, denoted as *PG0* = P1-P5, who are responsible for conducting a pilot user study to determine the proper settings of the two key parameters n and η in Chatbot4QR. Then, we divided the remaining 20 participants evenly into two groups: *PG1* = P6-P15 and *PG2* = P16-P25, while ensuring that members of the two groups have comparative years of programming experience.

Table 6 lists the allocation of queries to participants for our six user studies and the research questions investigated by each user study. More specifically, the *user study 1* is a pilot user study to investigate *RQ1* by randomly selecting ten queries and allocating the queries to the participants in *PG0*. For the other five user studies, we allocated *QG1* and *QG2* to *PG1* and *PG2*, respectively. The *user study 3* investigates *RQ3* by examining the usefulness of the CQs generated by Chatbot4QR for the 50 queries. *RQ4* is answered by recording the amount of time spent on the steps of Chatbot4QR during the *user studies 3* and *5*. The *user studies 2, 4, 5*, and *6* constitute a competitive experiment to investigate *RQ5* by comparing the quality of the top ten SO questions retrieved by Chatbot4QR and the two kinds of the top ten results retrieved using Web search engines (e.g., the SO search engine and Google) before and after interacting with Chatbot4QR. As the participants interact more with Chatbot4QR, they may gradually learn to recognize some technical details missed in their initial queries. Therefore, we required the participants to perform the *user study 2* (i.e., *Web Search before Interacting with Chatbot4QR*) before the *user study 3* (i.e., *Interaction with Chatbot4QR*), in order to minimize the learning effect that the participants may transfer the knowledge learned from Chatbot4QR to enhance the queries for Web search.

Before performing the user studies, the participants are expected to find a solution for each allocated query task. Given a query, when searching results using Web search engines, interacting with Chatbot4QR for evaluating the CQs, and judging the relevance of SO questions and Web search results, the participants should be based on the existing technical context specified in the query and/or their technical background. For example, for the query Q6 “*Better way to parse xml*”, it has no specified programming language, the participants can perform the user studies with their preferred

programming languages. For the query Q46 “*Choosing elements from python list based on probability*”, it has a programming language Python. The participants should perform the user studies based on Python, but they can determine the other technical context, e.g., a Python library, based on their technical background.

5 EXPERIMENT RESULTS

In this section, we answer the five research questions by conducting the corresponding user studies shown in Table 6.

5.1 RQ1: What are the Proper Settings of the Parameters n and η in Chatbot4QR?

Motivation. In Chatbot4QR, n and η are two key parameters for generating CQs and recommending SO questions for queries. The settings of n and η will affect the quality of generated CQs and recommended questions. It is necessary to figure out the proper settings of the parameters.

Approach. We randomly selected ten queries from the 50 experimental queries and allocated the queries to the five participants in *PG0*. Then, we conducted the pilot *user study 1* shown in Fig. 5 as follows.

1. *CQ generation using different settings of n .* We generated different CQs for each query by setting n from 5 to 50 with a step size 5.
2. *Usefulness evaluation of CQs.* We gathered the CQs generated using different values of n for each query. The participants used the interactive interface of our Chatbot4QR prototype to evaluate the CQs. Before evaluation, we gave a tutorial using a video conference call with the participants to introduce the prototype with an example query outside the experimental query set. Then, the participants evaluated the CQs of each query by performing two tasks: (1) rate the usefulness of each CQ by five grades ranging from 0 to 4, where 0, 1, 2, 3, and 4 mean ‘*strongly useless*’, ‘*useless*’, ‘*neutral*’, ‘*useful*’, and ‘*strongly useful*’, respectively; and (2) give feedback to the useful CQs. The *usefulness* of a CQ is judged by whether the CQ can help recognize any important information missed in a query for question retrieval.
3. *Sensitivity analysis of n .* For each setting of n , we counted the numbers of CQs with different usefulness and measured the ratio of useful CQs that are rated as 3 or 4 for each query. The usefulness of

skipped CQs was deemed to 0; and we considered the usefulness of the CQs that were not prompted to the participants (due to the early termination of interaction) as unknown, because such CQs were not evaluated by the participants. Then, we determined a proper value of n according to the results.

4. *Similarity adjustment using different settings of η .* Using the participants' feedback to the CQs generated with the proper n , we adjusted the initial semantic similarities of the 10,000 SO questions retrieved for each query (see Section 4.2) by setting η from 0 to 1 with a step size 0.1.
5. *Relevance evaluation of SO questions.* We gathered the top ten SO questions obtained using different values of η for each query. The participants evaluated the relevance of each question by five grades 0-4, where 0, 1, 2, 3, and 4 mean 'strongly irrelevant', 'irrelevant', 'neutral', 'relevant', and 'strongly relevant', respectively. In the aforementioned video conference, we explained to the participants that the relevance of a SO question to a query should be judged by evaluating the degree of matching between the SO question and the query task with the specified technical context (i.e., the technical terms appearing in the original query or given by the participants to the CQs).
6. *Sensitivity analysis of η .* For each setting of η , we measured the average performance of the top ten SO questions obtained for the ten queries using two metrics: Pre@k (Precision at k) [32] and NDCG@k (Normalized Discounted Cumulative Gain at k) [31], which are widely adopted in the IR community. Pre@k measures the percentage of relevant questions that are rated as 3 or 4 in the top- k ranking list. NDCG@k considers the ranking and rating scores of relevant questions.

$$\text{Pre}@k = \frac{\# \text{ relevant questions in the top-}k}{k} \quad (3)$$

$$\text{NDCG}@k = \frac{1}{IDCG_k} \sum_{i=1}^k \frac{2^{rel_i} - 1}{\log_2(1 + i)}, \quad (4)$$

where rel_i is the relevance score of the question at the ranking position i ; and $IDCG_k$ represents the maximum possible DCG score through position k that can achieve for a query. Then, we determined a proper setting of η according to the performance results.

Results. Fig. 6 shows the numbers of three kinds of CQs generated for ten queries using different $n \in [5, 50]$, with respect to each of the five participants P1-P5 in PG0. "Useful CQs" are the CQs rated as 3 or 4. "Useless & Neutral CQs" are the CQs rated as 0, 1, or 2. "Unknown CQs" are the CQs with unknown usefulness. From the figure, we have the following findings:

- Under each setting of n , the total numbers of CQs generated for the five participants are different. For example, 15 (= 7 + 8) and 18 (= 2 + 16) CQs were generated for the participants P1 and P2, respectively, when $n = 5$. This result is because that during the interaction, Chatbot4QR can dynamically generate subsequent CQs based on the participants' feedback to the initially

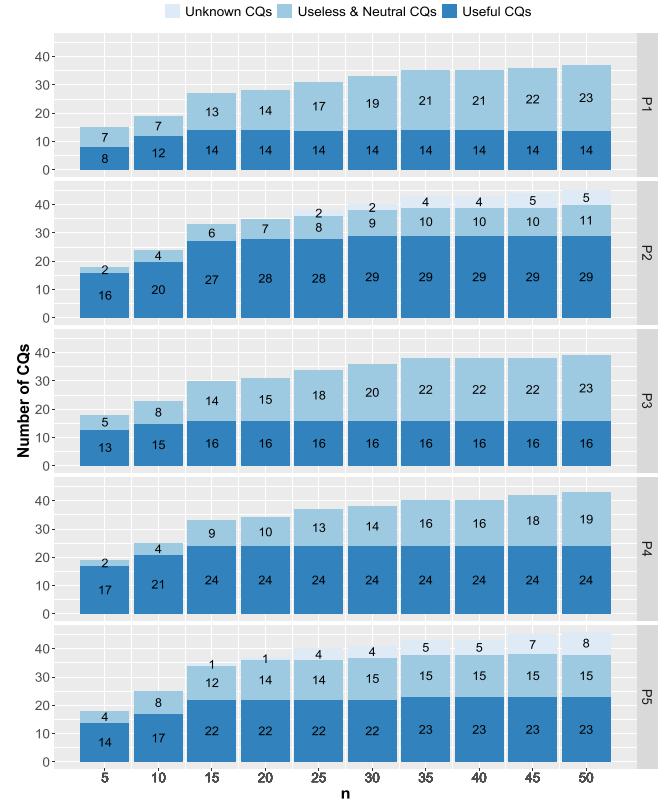


Fig. 6. The numbers of three kinds of CQs generated for ten queries using different settings of the parameter n (i.e., ranging from 5 to 50). n is the number of the initial top similar SO questions used for CQ generation. "P1-P5" are five participants.

generated CQs, as illustrated in Fig. 4. In particular, the participants have their own personalized technical background; and their feedback to CQs can be varied. Therefore, it leads to different numbers of CQs.

- There are notable differences among the five participants with respect to the numbers of the three kinds of CQs. For example, when $n = 5$, only eight of the 15 CQs generated for P1 were evaluated as useful, while P2 evaluated 16 of the 18 generated CQs as useful. This result indicates that the participants had personalized judgement on the usefulness of CQs. Moreover, there are unknown CQs in the evaluation results of P2 and P5 when n is a little large (e.g., $n = 25$ for P2), meaning that some participants may only pay attention to a limited number of CQs during the interaction.
- For the five participants, at least 93.1 percent (= 27/29) of the useful CQs are generated by setting $n = 15$. When n is larger than 15, only one or two CQs are evaluated as useful by P2 and P5, while the number of useless, neutral, and unknown CQs increases. Therefore, we determine that $n = 15$ is a good setting for Chatbot4QR.

In Chatbot4QR, the parameter n , i.e., the number of the initial top similar SO questions used for CQ generation, is suggested to be set as 15.

Table 7 presents the detailed evaluation results of each participant on the CQs generated for each query using $n =$

TABLE 7
Evaluation of the CQs Generated Using the Initial Top 15 Similar SO Questions Retrieved for Ten Queries (i.e., Setting the Parameter $n = 15$)

Query No.	#Initial CQs	P1		P2		P3		P4		P5	
		#CQs	Ratio of Useful CQs								
5	6	8	0.500	8	0.750	7	0.429	8	0.750	8	0.875
14	4	5	0.400	8	0.875	5	0.600	6	1.000	7	0.857
15	2	3	0.667	3	1.000	3	1.000	3	1.000	3	1.000
21	4	5	0.800	5	1.000	5	0.600	6	0.833	5	0.800
26	6	7	0.429	8	0.875	7	0.429	7	0.571	7	0.429
31	2	3	1.000	3	1.000	3	1.000	3	0.667	3	1.000
35	1	2	1.000	2	1.000	2	1.000	2	1.000	2	0.500
42	3	4	0.750	4	0.750	4	0.500	4	0.750	4	0.500
45	2	3	1.000	3	1.000	3	0.667	2	0.500	3	0.667
48	5	5	0.400	5	0.800	5	0.400	6	0.667	5	0.600

"P1-P5" are five participants. "#Initial CQs" is the number of CQs that are initially by Chatbot4QR before interacting with the participants. "#CQs" is the number of CQs eventually generated by Chatbot4QR based on the participants' personalized feedback to CQs.

TABLE 8
The Average Performance of the Top Ten SO Questions Retrieved by Chatbot4QR for Ten Queries Using Different Settings of the Parameter η (i.e., Ranging From 0.0 to 1.0)

η	Pre@1	Pre@5	Pre@10	NDCG@1	NDCG@5	NDCG@10
0.0	0.480	0.456	0.358	0.453	0.506	0.558
0.1	0.720	0.652	0.518	0.653	0.728	0.788
0.2	0.840	0.680	0.550	0.741	0.764	0.821
0.3	0.900	0.648	0.502	0.783	0.743	0.790
0.4	0.900	0.616	0.482	0.783	0.727	0.764
0.5	0.880	0.576	0.462	0.765	0.697	0.736
0.6	0.820	0.556	0.442	0.719	0.679	0.708
0.7	0.800	0.536	0.430	0.710	0.665	0.698
0.8	0.760	0.536	0.428	0.675	0.650	0.681
0.9	0.760	0.516	0.414	0.675	0.625	0.664
1.0	0.760	0.516	0.398	0.675	0.624	0.653

η is the Weight Coefficient of the Participants' Feedback to CQs in Eq. (1).

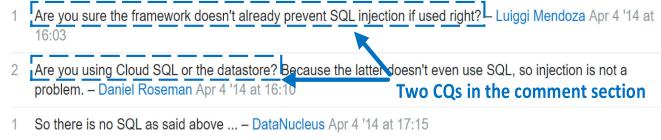
15. "#Initial CQs" is the number of CQs that are initially generated by Chatbot4QR before interacting with the participants. "#CQs" is the total number of CQs generated after interacting with each participant. "Ratio of Useful CQs" is the ratio of useful CQs to the total CQs. We observe that the values of "#CQs" and "Ratio of Useful CQs" vary from the participants. For example, for the query Q5, the participant P3 got seven CQs (i.e., one CQ was dynamically generated), while the other participants got eight CQs (i.e., two CQs were dynamically generated). The ratios of useful CQs for P2, P4, and P5 are more than 0.75 and much higher than those for P1 and P3. These results show that our chatbot can generate personalized CQs based on the individual interaction with a participant; and the participants have personalized judgement on the usefulness of CQs.

Table 8 presents the average performance of the top ten SO questions retrieved using different $\eta \in [0, 1]$ by leveraging the participants' feedback to the CQs generated with $n = 15$. From the table, we have the following findings:

- The performance achieved with a positive η is much better than that achieved with $\eta = 0.0$, indicating that the participants' feedback to CQs can indeed help retrieve more relevant SO questions.

- As η increases from 0.0 to 1.0, the Pre@k and NDCG@k values increase first until reach a peak; thereafter they decreases. This result can be explained by the fact that a query typically contains only a few keywords, a relatively large η can overweight the user's technical feedback. Consequently, the recommended questions can match the user's technical requirements perfectly but are irrelevant to the programming problem.
- The optimal Pre@1 and NDCG@1 are achieved when $\eta = 0.3$ or 0.4. When $k = 5$ and 10, the optimal Pre@k and NDCG@k are achieved with $\eta = 0.2$. Based on these results, there are two proper settings of η depending on the user's desired number of recommended questions. If a user focuses on the top one question, it is suggested to set $\eta = 0.3$ or 0.4, otherwise $\eta = 0.2$ is suggested. Moreover, in terms of Pre@1 and NDCG@1, the performance achieved with $\eta = 0.2$ is close to the optimal performance. Therefore, it is also a simple and good suggestion to set $\eta = 0.2$, regardless of the value of k.

In Chatbot4QR, for simplicity, the weight coefficient η in Eq. (1) used for generating the recommended SO questions is suggested to be set as 0.2.



add a comment

Fig. 7. Two CQs in the comment section of a SO question.

5.2 RQ2: How Effective Can Chatbot4QR Generate CQs?

Motivation. Our work is the first attempt to automatically generate CQs to interactively refine queries with the user involvement, in order to retrieve more relevant technical questions from Q&A sites. We want to evaluate the effectiveness of Chatbot4QR for CQ generation and verify whether the CQs can help users recognize missing technical details in queries.

Approach. We conducted a user study (i.e., the *user study 3* shown in Fig. 5) to evaluate the CQs generated by Chatbot4QR for the 50 experimental queries, under the setting of $n = 15$. To the best of our knowledge, there is a similar work proposed by Rao *et al.* [17], named *EVPI*, which aims to generate CQs for asking good technical questions in Q&A sites. Unlike Chatbot4QR that can automatically generate CQs, *EVPI* extracts the existing CQs in the comment sections of the top ten similar questions retrieved using Lucene. Fig. 7 shows two example CQs in the comment section of a SO question.⁵ We implemented *EVPI* using the source code released at Github,⁶ and used *EVPI* to generate CQs for each experimental query.

The 20 participants in PG1 and PG2 were required to evaluate the two kinds of CQs (one kind is generated by Chatbot4QR and the other kind is generated by *EVPI*) for their 25 allocated queries in QG1 and QG2, respectively. We modified the interactive interface of our Chatbot4QR prototype to run for the CQs generated by *EVPI*. More specifically, the prototype automatically prompted each query and the two kinds of CQs generated for the query in random order. The participants did not know which kind of CQs were generated by Chatbot4QR or *EVPI*. After completing the evaluation of CQs for a query, the participants needed to choose a preferred kind of CQs (i.e., the first or the second prompted kind). Before starting the evaluation, we launched a video conference to introduce the modified prototype to the participants with an example query. Then, the participants used the prototype to evaluate the two kinds of CQs for each allocated query by performing three tasks:

1. Rate the usefulness of each CQ by five grades 0-4, which are defined in Section 5.1.
2. Give feedback to the useful CQs.
3. Specify the preferred kind of CQs (when both Chatbot4QR and *EVPI* generated a set of CQs).

Note that the three tasks are not mandatory. The participants had the freedom to choose to perform any of the tasks. More specifically, the participants can skip a CQ if they think it is useless or feel difficult to answer. The participants

can terminate the interaction with the chatbot early when they think that they have answered enough CQs for a query. If the participants have no preference for any of the two kinds of CQs, they can skip the Task 3. Since the participants may not know some technical terms asked in the CQs, they can search for unfamiliar technical terms (e.g., OpenCV and Keras) online during the interaction. Moreover, we asked the participants to manually record the amount of time spent on the interaction with Chatbot4QR for 25 allocated queries, as the participants can take a short break during the user study in case of personal work or fatigue. After the user study, we interviewed the participants to obtain their comments about the CQs produced by both approaches.

For each query, we counted the numbers of CQs generated by *EVPI* and Chatbot4QR, and measured the average ratio of useful CQs evaluated by the ten participants who were responsible for the query. We considered the usefulness of skipped CQs as 0 and excluded the CQs that were not displayed to the participants as the usefulness of such CQs was unknown. We also analyzed the participants' preferred kinds of CQs for the queries that have CQs generated by both approaches. We first identified two sets of participants for a query who preferred Chatbot4QR or *EVPI*, which are denoted as $P_{Chatbot4QR}$ and P_{EVPI} , respectively. Then, we defined the “preference ratio” of the ten participants for the query as $|P_{Chatbot4QR}| : |P_{EVPI}|$.

Furthermore, according to the 20 types of SO tags shown in Table 1 and the three heuristic rules for CQ generation described in Section 3.2.2, Chatbot4QR can generate CQs that ask for 40 types of technical details, i.e., 20 types of SO tags and the versions. To examine whether the CQs that ask for some specific types of technical details would be more likely to be perceived as useful by users, we counted the numbers of CQs that ask for different types of technical details. We also measured the ratio of useful CQs that ask for each type.

Results. Table 9 presents the numbers of CQs generated by Chatbot4QR and *EVPI*, as well as the average ratio of useful CQs for each query. For Chatbot4QR, we present the number of initially generated CQs and the average number of CQs (i.e., “Avg. #CQs”) obtained by the ten participants in PG1 or PG2 after interaction, for each query. The bottom row shows the overall average results of both approaches on the 50 queries. From the table, we have the following findings:

- As for *EVPI*, it generated 1.3 CQs for a query on average and generated zero CQs for ten queries. The overall ratio of useful CQs for 50 queries is 16.7 percent, meaning that only a few CQs generated for a query were useful. Obviously, *EVPI* failed to generate any useful CQs for some vague queries. For example, the query Q6, i.e., “Better way to parse xml”, is vague due to the missing of a specific programming language. However, no CQ was generated by *EVPI* for Q6.
- As for Chatbot4QR, on average for a query, it initially generated 4.1 CQs and finally generated 5.1 CQs after interacting with the participants. This result means that on average one CQ was dynamically generated for a query based on the participants’ feedback. More specifically, 0-2.2 new CQs were generated for the 50 queries during the interaction. We observe that the

5. <https://stackoverflow.com/questions/22867636>

6. https://github.com/raosudha89/ranking_clarification_questions

TABLE 9
Evaluation of the CQs Generated by Chatbot4QR and *EVPI*

Query No.	CQs Generated by Chatbot4QR			CQs Generated by <i>EVPI</i>	
	#Initial CQs	Avg. #CQs	Avg. Ratio of Useful CQs	#CQs	Avg. Ratio of Useful CQs
1	2	3	0.833	1	0.400
2	3	4	0.750	2	0.250
3	3	4.4	0.565	2	0.000
4	4	4.6	0.590	0	—
5	7	9	0.522	1	0.000
6	5	6	0.500	0	—
7	3	4	0.425	1	0.000
8	9	9.9	0.314	1	0.400
9	2	2.4	0.750	1	0.000
10	4	5.9	0.607	1	0.400
11	3	4.9	0.590	2	0.200
12	5	6.3	0.412	0	—
13	2	3	0.733	1	0.000
14	5	7.1	0.541	0	—
15	3	4	0.775	2	0.200
16	7	9	0.496	1	0.000
17	2	2.8	0.783	4	0.000
18	6	7.7	0.488	1	0.000
19	6	8.2	0.449	0	—
20	3	4	0.900	1	0.000
21	5	6.1	0.624	0	—
22	4	5	0.620	2	0.150
23	3	3	0.500	3	0.000
24	2	2.8	0.750	4	0.325
25	6	7.1	0.577	2	0.250
26	6	7	0.471	0	—
27	4	4.5	0.512	0	—
28	3	4	0.775	2	0.250
29	4	4	0.700	0	—
30	3	4.5	0.710	2	0.250
31	2	3	0.800	2	0.200
32	5	6	0.642	2	0.300
33	3	3.2	0.767	1	0.600
34	7	8.9	0.479	1	0.200
35	2	3.6	0.725	1	0.700
36	5	6.4	0.626	2	0.200
37	5	7.3	0.664	1	0.300
38	8	8.4	0.419	1	0.000
39	5	5	0.553	1	0.100
40	4	5	0.460	2	0.300
41	3	4	0.775	2	0.300
42	4	4.9	0.565	2	0.100
43	4	4.8	0.595	3	0.267
44	4	5	0.480	1	0.600
45	2	3	0.933	1	0.100
46	4	4	0.600	2	0.300
47	2	2.7	0.483	0	—
48	7	7.1	0.377	1	0.600
49	4	4.8	0.570	1	0.100
50	2	2	0.600	1	0.000
Avg.	4.1	5.1	0.608	1.3	0.167

For a query, “#Initial CQs” is the number of CQs that are initially generated by Chatbot4QR; “Avg. #CQs” is the average number of CQs generated by Chatbot4QR after the interaction with ten participants; and “#CQs” is the number of CQs generated by *EVPI*.

number of CQs generated by Chatbot4QR is approximately four times the number of CQs generated by *EVPI*. Compared with *EVPI*, the effectiveness of Chatbot4QR depends on the number of increased useful CQs. If more than 16.7 percent of the increased CQs were useful, the effectiveness of Chatbot4QR would be better than that of *EVPI*. For each approach, we counted the number of times the generated CQs

TABLE 10
Statistics on the Evaluation of the CQs Generated by Chatbot4QR and *EVPI*

Approach	#CQs Evaluated by the Participants	#Useful CQs Evaluated by the Participants
EVPI	650	131
Chatbot4QR	2,565	1,479

are evaluated by the participants and the number of times the CQs are evaluated as useful, as shown in Table 10. Among the 1,915 (=2,565-650) additional evaluations of the CQs generated by Chatbot4QR, 1,348 (i.e., 70.4 percent) are useful. Moreover, as listed in Table 9, the overall ratio of useful CQs that are generated by Chatbot4QR is 60.8 percent for the 50 queries. For 37 queries, the average ratios of useful CQs generated by Chatbot4QR are no less than 50 percent. In contrast, the average ratios of useful CQs generated by *EVPI* are no less than 50 percent for only four queries, i.e., Q33, Q35, Q44, and Q48. As demonstrated in the results, Chatbot4QR in CQ generation for a query is more effective than *EVPI*.

Fig. 8 shows the numbers of queries with different preference ratios. There are 18 queries with the preference ratio ‘10:0’, meaning that for these queries, all the ten participants preferred the CQs generated by Chatbot4QR. For the nine queries with ‘5:3’, ‘8:1’, and ‘9:0’, one or two participants had no preference on the two kinds of CQs. We observe that most of the participants preferred the CQs generated by Chatbot4QR for the 40 queries (that have CQs generated by both approaches).

Two major comments about *EVPI* given by the participants are: (1) most of the generated CQs are too specific to a particular problem and often not useful for retrieving relevant questions, e.g., the CQ “What exactly is a week number in this context, and what does ‘date.weekday’ have to do with it?” generated for the query Q10; and (2) even some CQs might be useful but they are difficult to answer with a few words, e.g., the CQ “What output did you get?” generated for Q7. These issues can be explained by the objective of *EVPI* that it aims at generating CQs to help users refine technical questions, so that the questions can be easier to answer. Therefore, most of the CQs generated by *EVPI* may not be useful for question retrieval.

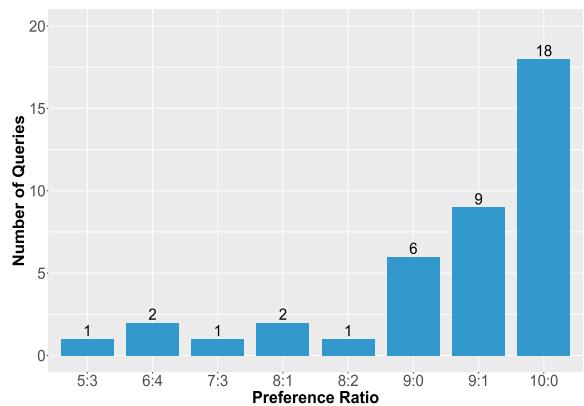


Fig. 8. Preference ratios of the CQs that are generated by Chatbot4QR and *EVPI* for 40 queries.

TABLE 11
The CQs Generated by Chatbot4QR for the Two Queries Q18 and Q42 in Table 4

Query No.	CQs Generated by Chatbot4QR for the Query	Score for CQ Ranking	Prompting Order	Usefulness	Feedback	Positive and Negative Feedback Used to Refine the Query
18	What programming language, e.g., javascript or python, does your problem refer to?	1.000	1	0		pfb(Q18) = {node.js, windows 8, babel}
	If you are using a framework, e.g., node.js or mocha, please specify:	0.604	2	3	node.js	
	Could you provide an operating system, e.g., windows or linux?	0.198	3	4	windows	
	Can you specify the version of windows, e.g., 7 or 8?	—	4	4	8	
	Which library, e.g., package.json or babel, are you using?	0.167	5	3	babel	
	Are you using json? (y/n), or some other formats.	0.135	6	0		
42	Are you using terminal? (y/n), or some other classes.	0.066	7	0		pfb(Q42) = {python 2.7, numpy} nfb(Q42) = {.net, indexing}
	What programming language, e.g., java or #, does your problem refer to?	1.000	1	3	python	
	Can you specify the version of python, e.g., 3.x or 2.7?	—	2	3	2.7	
	Are you using .net? (y/n), or some other frameworks.	0.033	3	1	n	
	Are you using numpy? (y/n), or some other libraries.	0.032	4	4	numpy	
	Are you using indexing? (y/n), or some other techniques.	0.032	5	2	n	

For the CQs related to each query, we present the scores for ranking the CQs, the orders of the CQs prompted to a participant (i.e., P16 for Q18 and P7 for Q42), as well as the usefulness and feedback given by the participant. $pfb(q)$ and $nfb(q)$ are the positive feedback and negative feedback to all the CQs of a query q , respectively.

Table 11 presents the CQs generated by Chatbot4QR for the two queries Q18 and Q42. For the CQs related to each query, we present (1) the scores for ranking the CQs (see Section 3.2.2), (2) the orders of prompting the CQs to a participant (i.e., P16 for Q18 and P7 for Q42), and (3) the usefulness and feedback given by the participant. The two CQs without scores are dynamically generated based on the participants' feedback, e.g., P16's feedback 'windows' to the third CQ of Q18. We observe that the ratios of useful CQs for Q18 and Q42 are 57.1 and 60.0 percent, respectively. We can use the final two kinds of feedback collected from all CQs of a query q (i.e., the positive feedback $pfb(q)$ and negative feedback $nfb(q)$) to refine q . More specifically, $pfb(q)$ and $nfb(q)$ are used to adjust the initial semantic similarities of SO questions retrieved for q , as demonstrated in Eq. (1).

Table 12 presents the numbers of CQs and useful CQs generated by Chatbot4QR that ask for 30 types of technical details (including 18 technique types of SO tags and the versions of 12 technique types). The types with '(v)' are the versions of the corresponding technique types. For example, 'programming language (v)' means the version of a

programming language. The first row shows that there are 370 CQs that ask for programming languages; and 350 (i.e., 94.6 percent) of the CQs are evaluated as useful by the participants. We observe that the top five types of technical details asked by the maximum numbers of CQs are 'programming language (v)', 'programming language', 'library', 'framework', and 'operating system'. The top five types with the highest ratios of useful CQs are 'non-OS system (v)', 'model/algorithm (v)', 'programming language', 'programming language (v)', and 'database'. Moreover, excluding the version types, the top five technique types with the highest ratios of useful CQs are 'programming language', 'database', 'operating system', 'library', and 'technique'.

On average, Chatbot4QR generates approximately five CQs for a query and 60.8 percent of the CQs are helpful for users to recognize missing technical details in the query. The CQs generated by Chatbot4QR are much better than the ones generated by the EVPI approach (as only 16.7 percent of the CQs generated by EVPI are helpful). Moreover, the CQs generated by Chatbot4QR that ask for some specific types of technical details are more likely to be perceived as useful by users. For the 20 types of SO tags shown in Table 1, the top five types with the highest ratios of useful CQs are 'programming language', 'database', 'operating system', 'library', and 'technique'.

TABLE 12
The Numbers of CQs and Useful CQs Generated for the 50 Queries by Chatbot4QR That Ask for Different Types of Technical Details, as Well as the Ratios of Useful CQs

Type	#CQs that ask for the Type	#Useful CQs that ask for the Type	Ratio of Useful CQs that ask for the Type
programming language	370	350	0.946
programming language (v)	494	422	0.854
database	20	15	0.750
database (v)	11	7	0.636
operating system	130	84	0.646
operating system (v)	61	37	0.607
library	329	179	0.544
library (v)	49	33	0.673
technique	129	48	0.372
class	98	35	0.357
class (v)	19	7	0.368
non-PL language	89	31	0.348
non-PL language (v)	47	29	0.617
format	130	37	0.285
format (v)	72	52	0.722
model/algorithm	19	5	0.263
model/algorithm (v)	22	21	0.955
tool	50	12	0.240
tool (v)	3	0	0.000
framework	239	55	0.230
framework (v)	22	14	0.636
design pattern	10	2	0.200
environment	59	10	0.169
environment (v)	2	0	0.000
non-OS system	70	11	0.157
non-OS system (v)	1	1	1.000
platform	10	1	0.100
engine	10	1	0.100
server	10	1	0.100
browser	20	1	0.050

Each type with "(v)" means the version of the corresponding technique type.

5.3 RQ3: Can Chatbot4QR Retrieve More Relevant SO Questions Than the State-of-the-Art Question Retrieval and Query Expansion Approaches?

Motivation. The ultimate goal of Chatbot4QR is to retrieve accurate SO questions for users based on their feedback to the CQs. Although it has been validated in RQ2 that most of the CQs generated by Chatbot4QR for a query are useful, it is necessary to check whether the refined queries (i.e., the participants' feedback to CQs) can improve the relevance of recommended questions.

Approach. We retrieved the top ten SO questions for the 50 experimental queries based on the participants' feedback to the CQs of each query, under the setting of $\eta = 0.2$. Then, we conducted a user study (i.e., the *user study 5* shown in Fig. 5) to evaluate the SO questions. We compared Chatbot4QR with several existing question retrieval and query expansion approaches. More specifically, we summarized two state-of-the-art approaches used for question retrieval:

- *Lucene*: This is the Lucene search engine which retrieves SO questions similar to a query based on the Lucene index built for a question repository [8]. We implemented *Lucene* using its source code⁷ released at Github.
- *Word Embedding (WE)*: This is the word embedding-based question retrieval approach widely used in recent work [5], [6]. We implemented *WE* using the source code⁸ released by Huang *et al.* [5].

A rich body of research work improves the performance of IR systems by reformulating queries using relevant terms extracted from thesauruses or similar resources. We summarized three major query expansion approaches:

- *WordNet (WN)*: This approach expands a query with the synonyms of keywords in WordNet. We implemented the WordNet-based query expansion approach proposed by Lu *et al.* [12].
- *QECK*: This approach expands a query using the important keywords contained in the top similar SO question-and-answer pairs [8]. The importance of a keyword is measured by considering both the TF-IDF score and the scores of SO questions and answers. We implemented *QECK* according to the details presented in the paper.
- *Tag Recommendation (TR)*: There are a number of papers on recommending SO tags for a technical question [36], [37], [38]. These papers are similar to Chatbot4QR to a certain extent as all of them focus on finding relevant SO tags for a target (a question or a query). We viewed the *TR* approaches as a specific kind of query expansion approaches, to check whether they can be used to recommend SO tags for queries. We implemented the neural network-based *TR* approach proposed by Liu *et al.* [38] using the open-source code⁹ and expanded a query with the top ten recommended SO tags.

We built nine baselines by combining the two retrieval approaches: *Lucene* and *WE*, and four query expansion approaches: *WN*, *QECK*, *TR*, and *IQR* (which refers to our interactive query refinement approach used in Chatbot4QR). The baselines are described as follows.

1. *Lucene*: This is the *Lucene* approach described above.
2. *WE*: This is the *WE* approach described above.
3. *Lucene+WN*: This approach uses *Lucene* to retrieve questions after expanding a query using *WN*.
4. *Lucene+QECK*: This approach uses *Lucene* to retrieve questions after expanding a query using *QECK*.
5. *Lucene+TR*: This approach uses *Lucene* to retrieve questions after expanding a query using *TR*.
6. *Lucene+IQR*: This approach uses *Lucene* to retrieve questions based on the query refined using *IQR*, i.e., the user's positive and negative feedback to CQs. More specifically, We first retrieved similar questions by applying the query and positive feedback to *Lucene*. Then, we removed the similar questions that contain any negative feedback.

7. <https://github.com/apache/lucene-solr>
 8. <https://github.com/tkdsheep/BIKER-ASE2018>
 9. <https://pan.baidu.com/s/1slujtU1>

7. *WE+WN*: This approach uses *WE* to retrieve questions after expanding a query using *WN*.
8. *WE+QECK*: This approach uses *WE* to retrieve questions after expanding a query using *QECK*.
9. *WE+TR*: This approach uses *WE* to retrieve questions after expanding a query using *TR*.

Note that Chatbot4QR can be simply viewed as a combination of *WE+IQR*. We applied the eight baselines except *Lucene+IQR* to the 50 queries and obtained eight lists of the top ten SO questions for each query. Since *IQR* is a personalized query refinement approach, we applied *Lucene+IQR* to retrieve the top ten questions based on each participant's feedback to the CQs of each query. Then, for each participant, we collected the different top ten questions retrieved for each query using Chatbot4QR and nine baselines. The participants evaluated the relevance of the questions by five grades 0-4, as defined in Section 5.1.

As the participants may probably have different preferences of techniques (e.g., the familiar programming languages shown in Table 5), they may get different SO questions retrieved by Chatbot4QR and *Lucene+IQR* for a query. Moreover, the participants may have their own judgement on the relevance of the questions. Therefore, we measured the overall Pre@k or NDCG@k performance of an approach *A* as its average performance evaluated by the 20 participants. More specifically, given a specific Pre@k or NDCG@k metric *m*, for each participant *P*, we computed *m* of each query according to *P*'s evaluation results of the SO questions retrieved by *A*. Then, we computed the average *m* of the 25 queries allocated to *P*. Finally, we computed the overall *m* of *A*, denoted as *m_A*, with respect to the average of the *m* values of 20 participants. Based on the overall performance results, we measured the “improvement degree” of Chatbot4QR over each baseline *B* in terms of a specific metric *m* as $\frac{m_{Chatbot4QR} - m_B}{m_B}$.

Furthermore, we examined whether the improvement of Chatbot4QR (denoted as *C*) over a baseline *B* is statistically significant. Considering that the participants may obtain different SO questions and have personalized benchmarks of relevant questions for a query, we defined a metric “significant ratio” to measure the statistical significance of the performance improvement of *C* over *B* as follows. For each participant, given a specific Pre@k or NDCG@k metric *m*, we built two samples for *C* and *B*, respectively, by gathering the *m* values of *C* and *B* on the 25 assigned queries. We used the Wilcoxon signed-rank test [39] to test the significance of *C* over *B* based on the two samples with three p-values {0.05, 0.01, 0.001}. For each p-value *p*, we identified the set of participants whose samples of *C* are significantly better than those of *B*, which are denoted as *SigP_{m,p}(C, B)*. Then, the significant ratio of *C* over *B*, given *m* and *p*, is measured as

$$SigR_{m,p}(C, B) = \frac{|SigP_{m,p}(C, B)|}{\# \text{ participants}}. \quad (5)$$

Finally, we chose the maximum significant ratio and the corresponding p-value.

As described in Section 5.2, the CQs generated by Chatbot4QR can ask for different types of technical details. In RQ2, we measured the ratios of useful CQs that ask for 30 types of technical details (see Table 12). We further measured the contributions of the participants' feedback to the

TABLE 13
Evaluation of the SO Questions Retrieved by Ten Approaches

Approach	Pre@1	Pre@5	Pre@10	NDCG@1	NDCG@5	NDCG@10
Lucene	0.414	0.332	0.279	0.369	0.369	0.396
Lucene+WN	0.308	0.237	0.216	0.300	0.283	0.315
Lucene+QECK	0.278	0.190	0.156	0.251	0.245	0.260
Lucene+TR	0.250	0.203	0.169	0.243	0.246	0.265
Lucene+IQR	0.540	0.434	0.343	0.480	0.478	0.496
WE	0.530	0.416	0.348	0.484	0.473	0.500
WE+WN	0.300	0.236	0.188	0.285	0.281	0.299
WE+QECK	0.310	0.232	0.201	0.269	0.269	0.293
WE+TR	0.352	0.232	0.209	0.319	0.289	0.318
Chatbot4QR	0.838	0.670	0.548	0.765	0.731	0.760

TABLE 14
Improvement Degrees and the Maximum Significant Ratios of Chatbot4QR Over Nine Baselines

Baseline	Pre@1		Pre@5		NDCG@1		NDCG@5	
	ImpD(%)	(p, SigR(%))	ImpD(%)	(p, SigR(%))	ImpD(%)	(p, SigR(%))	ImpD(%)	(p, SigR(%))
Lucene	102.4	(0.05, 100.0)	102.1	(0.05, 100.0)	107.0	(0.01, 95.0)	97.8	(0.01, 100.0)
Lucene+WN	172.1	(0.05, 100.0)	182.9	(0.01, 100.0)	154.5	(0.05, 100.0)	158.5	(0.01, 100.0)
Lucene+QECK	201.4	(0.01, 100.0)	251.9	(0.01, 100.0)	205.2	(0.05, 100.0)	197.6	(0.001, 100.0)
Lucene+TR	235.2	(0.01, 100.0)	229.7	(0.001, 100.0)	214.0	(0.01, 100.0)	197.3	(0.001, 100.0)
Lucene+IQR	55.2	(0.05, 85.0)	54.2	(0.05, 95.0)	59.4	(0.05, 90.0)	52.7	(0.05, 100.0)
WE	58.1	(0.05, 70.0)	60.9	(0.05, 95.0)	57.8	(0.05, 80.0)	54.6	(0.01, 95.0)
WE+WN	179.3	(0.05, 100.0)	183.9	(0.01, 100.0)	168.4	(0.01, 100.0)	160.0	(0.001, 100.0)
WE+QECK	170.3	(0.05, 100.0)	189.3	(0.01, 100.0)	184.5	(0.01, 100.0)	171.9	(0.001, 100.0)
WE+TR	138.1	(0.05, 100.0)	189.3	(0.001, 100.0)	139.3	(0.05, 100.0)	152.8	(0.001, 100.0)

"ImpD" is the improvement degree. "(p, SigR)" are the maximum significant ratio and the corresponding p-value.

CQs that ask for different types of technical details, in order to retrieve relevant questions. More specifically, for every feedback to a CQ of a query, we produced a list of the top ten SO questions by adjusting the initial semantic similarities of the 10,000 questions (see Section 4.2) using the single feedback. The participants evaluated the relevance of the questions that were not evaluated before for each allocated query. Then, for a query q , we computed the performance of a specific Pre@k and NDCG@k metric m improved by each feedback fb , denoted as $Imp_m(q, fb)$, as follows.

- If fb is a technique (e.g., a programming language) or a version given to an initially generated CQ, then $Imp_m(q, fb)$ is measured as $m_{Chatbot4QR}(q, fb) - m_{Initial}(q)$. $m_{Chatbot4QR}(q, fb)$ is the m value of the top- k questions retrieved for q using Chatbot4QR by leveraging fb ; and $m_{Initial}(q)$ is the m value of the initial top- k questions retrieved for q using our two-phase method.
- If fb is a version of a technique feedback fb' , then $Imp_m(q, fb)$ is measured as $m_{Chatbot4QR}(q, fb) - m_{Chatbot4QR}(q, fb')$.

Finally, we measured the average performance improvement achieved using the participants' feedback that belongs to a specific type of technical details. For each type, we also measured the average performance improvement achieved using the participants' feedback to all CQs of the queries that contain any feedback of the type.

Results. Table 13 presents the performance of the top ten SO questions retrieved using ten approaches. Table 14 presents

the improvement degrees and the maximum significant ratios of Chatbot4QR over the nine baselines. "ImpD(%)" is the improvement degree expressed as a percentage; and "(p, SigR (%))" are the maximum significant ratio expressed as a percentage and the corresponding p-value p . From the two tables, we have the following findings:

- Chatbot4QR achieves the best performance in terms of both Pre@k and NDCG@k. The result demonstrates that the queries refined by IQR (i.e., our interactive query refinement approach) can improve the quality of SO questions retrieved by WE.
- Chatbot4QR improves the two popular baselines WE and Lucene by at least 54.6 percent and 97.8 percent, respectively. Chatbot4QR significantly outperforms Lucene for all the participants in terms of Pre@1, Pre@5, and NDCG@5. Although Chatbot4QR does not significantly outperform WE for all the participants, the significant ratios are all higher than 70 percent. This result indicates that the improvement of Chatbot4QR over WE is significant for at least 14 of the 20 participants.
- Lucene+IQR improves Lucene by at least 22.91 percent, which further demonstrates the effectiveness of our IQR approach in helping users refine queries and retrieve more relevant questions using Lucene.
- WE outperforms Lucene by at least 24.48 percent. This is because that WE can retrieve semantically similar questions for queries, while Lucene cannot due to the lexical gaps issue.

TABLE 15
The Average Performance Improvement of SO Question Retrieval Achieved Using the Participants' Feedback to the CQs Generated by Chatbot4QR That Ask for Different Types of Technical Details

Type	#Cases	Pre@1		Pre@5		NDCG@1		NDCG@5	
		Avg. Imp	Avg. Imp by All	Avg. Imp	Avg. Imp by All	Avg. Imp	Avg. Imp by All	Avg. Imp	Avg. Imp
programming language	355	0.318	0.363	0.243	0.299	0.263	0.333	0.248	0.307
programming language (v)	454	0.075	0.326	0.040	0.264	0.065	0.285	0.039	0.265
database	15	0.333	0.333	0.173	0.240	0.236	0.271	0.179	0.268
database (v)	8	0.000	0.375	0.000	0.325	0.000	0.317	0.000	0.334
operating system	92	-0.207	0.217	0.039	0.241	-0.120	0.266	0.003	0.264
operating system (v)	45	0.000	0.200	0.018	0.276	0.000	0.276	0.016	0.290
library	183	0.279	0.421	0.118	0.280	0.224	0.381	0.133	0.293
library (v)	37	-0.081	0.324	-0.022	0.254	-0.011	0.275	-0.008	0.242
technique	56	0.054	0.464	-0.004	0.289	0.076	0.397	0.019	0.297
class	35	-0.057	0.343	-0.063	0.217	-0.094	0.306	-0.071	0.240
class (v)	13	0.000	0.077	0.000	0.431	0.000	0.021	-0.014	0.262
non-PL language	33	-0.364	0.273	0.012	0.406	-0.322	0.227	-0.063	0.307
non-PL language (v)	20	0.000	0.450	0.010	0.350	0.000	0.378	-0.028	0.259
format	40	-0.175	0.175	-0.035	0.200	-0.108	0.140	-0.036	0.202
format (v)	61	0.000	0.197	0.013	0.216	0.000	0.165	0.004	0.186
model/algorithm	5	-0.400	0.200	-0.120	-0.000	-0.107	0.333	-0.109	0.057
model/algorithm (v)	21	0.000	-0.143	0.010	0.124	0.000	-0.113	0.008	0.099
tool	15	0.000	0.200	0.040	0.213	0.036	0.196	0.047	0.233
framework	63	0.127	0.286	0.029	0.235	0.116	0.312	0.052	0.245
framework (v)	17	-0.176	0.235	-0.047	0.141	-0.024	0.267	-0.017	0.212
design pattern	3	0.000	0.333	0.067	0.467	0.000	0.311	0.066	0.521
environment	14	-0.143	0.143	-0.014	0.171	-0.062	0.248	-0.024	0.199
environment (v)	1	0.000	0.000	0.000	0.200	0.000	0.533	0.000	0.276
non-OS system	14	0.000	0.500	0.086	0.400	-0.038	0.479	0.046	0.403
non-OS system (v)	1	0.000	1.000	0.000	0.400	0.000	0.933	0.000	0.401
platform	3	0.000	1.000	0.000	0.200	0.000	1.000	0.000	0.333
engine	1	0.000	0.000	0.000	0.200	0.000	0.000	-0.054	0.025
server	1	0.000	0.000	0.000	0.400	0.000	0.000	0.000	0.203
browser	2	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.083

Each type with "(v)" means the version of the corresponding technique type. For each type, "#Cases" is the number of times a participant's feedback of the type is used to adjust the initial semantic similarities of SO questions of a query; "Avg. Imp" is the average Pre@k or NDCG@k improvement achieved using the participants' feedback of the type; "Avg. Imp by All" is the average Pre@k or NDCG@k improvement achieved using the participants' feedback to all CQs of the queries that contain any feedback of the type.

- WE outperforms WE+WN, WE+QECK, and WE+TR by at least 50.57 percent. Lucene outperforms Lucene+WN, Lucene+QECK, and Lucene+TR by at least 22.93 percent. These results may indicate that the three automatic query expansion approaches (i.e., WN, QECK, and TR) are not suitable for reformulating queries to improve the performance of question retrieval.

Table 15 presents the average performance improvement of question retrieval achieved using the participants' feedback to the CQs that ask for 29 types of technical details. For each type, "# Cases" is the number of times a participant's feedback of the type is used for question retrieval; "Avg. Imp" is the average Pre@k or NDCG@k improvement achieved using the participants' feedback of the type; and "Avg. Imp by All" is the average Pre@k or NDCG@k improvement achieved using the participants' feedback to all CQs of the queries that contain any feedback of the type. The type 'tool (v)' in Table 12 has no improvement result in Table 15, since none of the three CQs that ask for the type of technical details are evaluated as useful and thus there is no feedback of the type used for question retrieval. We find that the feedback of some types has positive improvement while the feedback of other types has no or negative improvement. For example, in terms of Pre@1, the improvement of the three types 'programming language', 'database (v)', and 'operating system' are 0.318, 0.000, and -0.207, respectively. We also find that the improvement achieved using the feedback of a type can be different in terms of different metrics. For example,

the feedback of 'operation system' has a negative impact on the Pre@1 and NDCG@1 performance, however, it has positive improvement in terms of Pre@5 and NDCG@5. The version types have very low improvement. One of the possible reasons could be that many SO questions do not explicitly specify the versions of the involved techniques, especially in the question title and tags. To effectively leverage the feedback of versions, there needs a method for inferring the versions of techniques from the content (e.g., code snippets) of questions. Moreover, the values of "Avg. Imp by All" are generally higher than those of "Avg. Imp", indicating that better performance is achieved by integrating the feedback to all CQs of a query. The top three types with the maximum performance improvement are the same, i.e., 'programming language', 'database', and 'library', in terms of all metrics.

Compared with the nine baselines that involve two question retrieval approaches and four query expansion approaches, Chatbot4QR retrieves more relevant SO questions for queries. The improvement degree of Chatbot4QR over the word embedding-based question retrieval approach (WE) is at least 54.6 percent. Furthermore, the improvement of Chatbot4QR over WE is statistically significant for more than 70 percent of the participants. Moreover, the participants' feedback to the CQs that ask for different types of technical details has different contributions to question retrieval. The top three types with the maximum contributions are 'programming language', 'database', and 'library'.

5.4 RQ4: How Efficient is Chatbot4QR?

Motivation. In Chatbot4QR, several resources need to be built offline, including the Lucene index of SO questions, two language models, and the categorization and version-frequency information of SO tags. Although the offline processing takes a substantial amount of time, the built resources are reusable. We are interested in finding out the response time that Chatbot4QR can respond to a user once the user submits a query. If the response time is too long, our approach may not be acceptable even if it is effective in generating useful CQs and retrieving relevant SO questions. Therefore, it is essential to examine whether Chatbot4QR is efficient for practical uses.

Approach. We recorded the amount of time that Chatbot4QR, WE, and Lucene spent on the offline processing of SO data and online question retrieval during our experiments. After the *user study 3*, we asked the participants to report their time spent on interacting with our chatbot. We did not consider the time costs of the other seven baselines because *Lucene+IQR* is based on our *IQR* and the performance of other baselines is too low (see Table 13).

Results. Table 16 presents the time costs of the three approaches. From the table, we have the following findings:

- As for the offline processing, the processing time of Chatbot4QR is 91.15 hours, which is much higher than those of *Lucene* and *WE*. This is because that the offline processing of Chatbot4QR contains three main parts: (1) the semi-automatic categorization of SO tags (74 hours); (2) the building of the Lucene index of SO questions and two language models ($8.52 + 7.38 = 15.9$ hours); and (3) the tag identification from SO questions (1.25 hours). Since the resources are reusable and can be incrementally updated (as explained in Section 3.1.5), the relatively high time cost of the offline processing of Chatbot4QR may not be a problem for practical uses.
- As for the online question retrieval for a query, the processing time of Chatbot4QR contains three parts (as shown in Table 16): (1) *Response* is the amount of time required to respond to a participant (1.30 seconds), including the two-phase question retrieval and CQ generation; (2) *Interaction* is the amount of time that a participant spent on the interaction with our chatbot (about 42 seconds); and (3) *Recommendation* is the amount of time required to adjust the similarities of 10,000 SO questions and produce the top ten recommended questions (0.02 seconds). The response time is 1.30 seconds, meaning that Chatbot4QR can responsively start interacting with the user after receiving a query. After the interaction, the question recommendation list can be produced within 0.02 seconds. These results demonstrate the efficiency of Chatbot4QR.
- The time spent by *WE* on question retrieval is 49.96 seconds per query, which is high because *WE* measures the semantic similarities between a query and the 1,880,269 SO questions in our repository. In contrast, the two-phase question retrieval approach used in Chatbot4QR is scalable. The reason is that the first phase uses *Lucene*, which is efficient to

TABLE 16
Time Costs of Three Approaches

Approach	Offline Processing	Online Question Retrieval
Lucene	8.52h	0.02s
WE	7.38h	49.96s
Chatbot4QR	91.15h	Response: 1.30s Interaction: ≈ 42 s Recommendation: 0.02s

handle a large-scale repository, as shown in Table 16; and by fixing the parameter N to a relatively large value (e.g., 10,000 in this work), the time cost of the second phase is stable.

It is worth to mention that the average time spent on the interaction with our chatbot for a query is 42 seconds. For a few queries, some participants took 2-3 minutes because they needed to search for unfamiliar technical terms asked in the CQs online. As confirmed by the participants, the amount of time spent on the interaction is practically acceptable since the feedback to CQs can contribute to more relevant SO questions and reduce the time required for the manual examination of undesirable questions. Although the quality of retrieved questions relies on the user's feedback to CQs, Chatbot4QR does not require the user to answer every CQ. The amount of the interaction time depends on (1) the user's programming experience and (2) whether the user wants to search for unfamiliar technical terms online, in order to provide more precise feedback to CQs and obtain more relevant questions.

Chatbot4QR takes approximately 1.30 seconds to respond to a user after the user submits a query and 0.02 seconds to produce the SO question recommendation list after interacting with the user, indicating that Chatbot4QR is efficient for practical uses.

5.5 RQ5: Can Chatbot4QR Help Obtain Better Results Than Using Web Search Engines Alone?

Motivation. In practice, developers often use the SO search engine and general-purpose search engines (e.g., Google) to look for desired information [13], [40]. To further validate the effectiveness of Chatbot4QR, we investigate whether Chatbot4QR can help users obtain better results than using Web search engines (including the SO search engine, Google, etc.) alone. Here, a result refers to a SO question or any other resources returned by Web search engines, e.g., a blog or a tutorial.

Approach. We conducted four user studies (i.e., the *user studies 2, 4, 5, and 6* shown in Fig. 5) for answering RQ5. Before the interaction with Chatbot4QR, we asked the 20 participants in PG1 and PG2 to obtain the top ten results using Web search engines of their choices for each allocated query. The participants can modify a query according to the returned results until they are satisfied with the results (excluding the SO question whose title is the same as the original query) listed in a webpage. For each query, we asked the participants to record the final query and the top ten results in the returned webpage. After interacting with Chatbot4QR, the participants can obtain new results for a query using Web search engines by reformulating the query

TABLE 17
Evaluation of the Results Obtained Using Web Search Engines Before/After Interacting With Chatbot4QR

	Pre@1	Pre@5	Pre@10	NDCG@1	NDCG@5	NDCG@10
WS	0.634	0.483	0.401	0.532	0.500	0.502
WS+IQR	0.664	0.524	0.433	0.555	0.528	0.531
Best	0.900	0.725	0.585	0.798	0.746	0.749
ImpD(%) of Best over WS	22.4	29.4	26.9	27.5	26.9	29.8
(p, SigR (%)) of Best over WS	(0.05, 80.0)	(0.05, 100.0)	(0.05, 90.0)	(0.05, 90.0)	(0.01, 100.0)	(0.01, 100.0)
ImpD(%) of Best over WS+IQR	16.9	19.3	17.3	22.3	20.0	22.5
(p, SigR (%)) of Best over WS+IQR	(0.05, 70.0)	(0.05, 95.0)	(0.05, 85.0)	(0.05, 85.0)	(0.01, 100.0)	(0.05, 100.0)

"ImpD" is the improvement degree. "(p, SigR)" are the maximum significant ratio and the corresponding p-value.

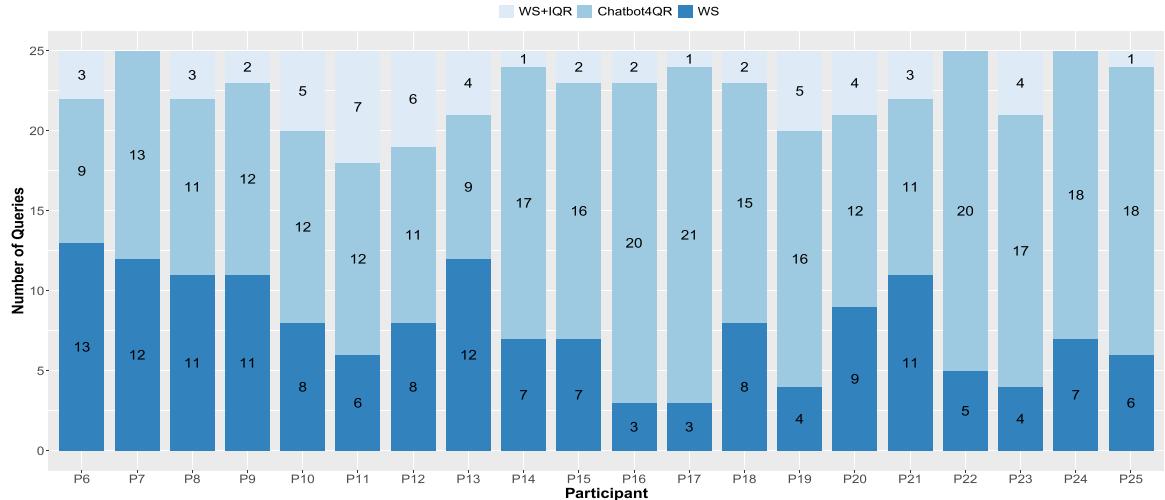


Fig. 9. The numbers of queries that achieve the best results using WS, WS+IQR, and Chatbot4QR by 20 participants.

with interesting technical terms in their feedback to the CQs. Take the query Q18 shown in Table 11 as an example, the participant P16 can reformulate Q18 by adding some technical terms, e.g., 'node.js', in his/her positive feedback. Then, the participants evaluated the relevance of the two kinds of Web search results. Finally, the participants chose the preferred/best results for each allocated query from the three kinds of results: the top ten SO questions retrieved by Chatbot4QR and the two top ten Web search results. After the user studies, we interviewed the participants to get their opinions on the value of Chatbot4QR.

For each participant, we measured the performance of the following three top ten results for each allocated query:

- WS: the top ten results obtained using Web search engines before the interaction with Chatbot4QR.
- WS+IQR: the top ten results obtained using Web search engines after the interaction with Chatbot4QR.
- Best: the best results chosen by the participant.

We viewed WS, WS+IQR, and Best as three retrieval approaches. For a specific Pre@k or NDCG@k metric, we measured the overall performance of each approach as the average performance evaluated by the 20 participants. Moreover, we measured the improvement degrees and the maximum significant ratios of Best over WS and WS+IQR. The detailed measurement process can refer to Section 5.3.

Results. Table 17 presents the overall performance of WS, WS+IQR, and Best, as well as the improvement degrees and

the maximum significant ratios of Best over WS and WS+IQR. From the table, we have the following findings:

- Best outperforms WS and WS+IQR in terms of both Pre@k and NDCG@k with an improvement of at least 22.4 and 16.9 percent, respectively. The significant ratios of Best over WS are all higher than 80 percent, indicating that the improvement of Best over WS is statistically significant for at least 16 of the 20 participants. This result shows that more desired results are obtained by the participants after interacting with Chatbot4QR than directly using Web search engines.
- WS+IQR is slightly better than WS. This means that for some queries, the participants obtained better results using Web search engines again by reformulating the queries using information that they learned from the interaction with our chatbot.

We counted the numbers of queries that achieve the best results by WS, WS+IQR, and Chatbot4QR for each participant, as shown in Fig. 9. From the figure, we have the following findings:

- For 12-22 of the 25 (i.e., 48-88 percent) assigned queries, the participants preferred the results obtained by Chatbot4QR or WS+IQR. For 16 participants (except P6, P8, P13, and P21), Chatbot4QR achieves the best results for the largest number of queries. Moreover, there are 1-7 queries whose best results are

TABLE 18
The Numbers of Participants Who Reformulated the 50 Queries

Query Nos.	#Participants Who Reformulated the Queries
2, 26, 45	8
13, 20, 41, 48	7
1, 21, 22, 25, 32, 37, 39, 46	6
3, 8, 12, 18, 19, 27, 28, 31, 36	5
5, 10, 15, 17, 29, 33, 42, 50	4
6, 14, 34, 35, 38, 40, 49	3
7, 11, 16, 24, 30, 44	2
23, 43	1
4, 9, 47	0

obtained by *WS+IQR* for 17 participants (except P7, P22, and P24). For example, for the query Q22, the participant P3 reformulated it by adding the feedback ‘*mysql*’ given to the CQ “*I want to know whether you are using a database, e.g., mysql or mongodb. Can you provide it?*”, which contributes to the best results retrieved by Google. All these results are consistent with the overall performance shown in Table 17, which further demonstrate that for a considerable number of queries, Chatbot4QR helps the participants obtain better results than using Web search engines alone.

- For each of the two participant groups PG1 (=P6-P15) and PG2 (=P16-P25), there are notable differences among the participants with respect to the numbers of queries whose best results are obtained by *WS*, *WS +IQR*, and Chatbot4QR. By interviewing the participants, we found that the differences are mainly caused by the participants’ different preferences of techniques and programming experience. For example, for the query Q36, the participants P23 and P24 preferred Java while P25 preferred Python. Before using Chatbot4QR, P24 reformulated the query by adding ‘*jsoup*’ [41] (a Java HTML parser) while P23 simply added ‘*java*’. Consequently, they obtained different results for Q36.
- For all the 20 participants, *WS* achieves the best results for 3-13 queries. We found that most of those queries are relatively simple and have specified technical terms, e.g., Q1 and Q7. The result shows the good performance of Web search engines when the query is clearly specified. Although Chatbot4QR cannot achieve the best results for some queries, all the participants expressed their willingness to use our chatbot as a complement to Web search engines.

Moreover, we examined the query reformulation records of the participants by leveraging the final queries that they used for obtaining the results of *WS* and *WS+IQR*. Table 18 presents the queries according to the number of participants who had reformulated them. We observe that 24 (= 3+4+8+9) queries were reformulated by 5-8 participants, while 11 (= 3+2+6) queries were reformulated by 0-2 participants. We further analyzed the participants’ feedback to CQs used in their reformulated queries. Table 19 lists the statistics of the technical terms added to five frequently reformulated queries and five less frequently reformulated queries. The number in a parenthesis indicates the frequency of the corresponding technical term used to reformulate a query. As

TABLE 19
Technical Terms Used to Reformulate Ten Queries

Query No.	Technical Terms Used to Reformulate the Query
2	numpy(3), python(2), matplotlib(2), python 3.x(1)
26	java(3), collections(1), c(1), python 3(1), linux(1)
45	pandas(7), python(1)
13	python(2), numpy(2), java(1), c#(1), python 3.x(1)
48	regex(4), python 3.x(1), jquery(1), django(1)
7	java(2)
11	java(2)
16	xml(1), html(1)
23	python 3(1)
43	neural-network(1)

Each number in a parenthesis is the frequency of the technical term used to reformulate the corresponding query.

shown in Table 19, the queries reformulated by more participants often involve multiple techniques. For example, the technical terms used to reformulate the query Q26 include three programming languages {‘*java*’, ‘*c*’, ‘*python 3*’}, one operating system {‘*linux*’}, and one library {‘*collections*’}.

For 12-22 of the 25 (i.e., 48 percent-88 percent) assigned queries, the participants preferred the results obtained by Chatbot4QR or using Web search engines with the queries reformulated after interacting with Chatbot4QR. This demonstrates that Chatbot4QR can help obtain better results than using Web search engines alone. During the interview with the participants, all the participants expressed their willingness to use Chatbot4QR as a complement to Web search engines.

6 DISCUSSION

6.1 Why Chatbot4QR Can Help Users Retrieve Better SO Questions and Web Search Results?

Tables 13 and 14 show that Chatbot4QR significantly outperforms the two popular retrieval approaches: *WE* and *Lucene*, as well as their variants combined with three query expansion approaches: *WN*, *QECK*, and *TR*. Moreover, *WE* is better than *Lucene* because *WE* can bridge the lexical gaps between SO questions and queries while *Lucene* cannot. The variants perform worse than *WE* and *Lucene* due to the fact that *WN*, *QECK*, and *TR* may introduce noise terms and decrease the quality of retrieved SO questions. As an example, for the query Q9, i.e., “*how to encrypt data using AES in Java*”, the terms expanded by *QECK* are: {*ruby*, *iv*, *php*, *openssl*, *algorithm*, *i.e*, *fast*, *disk*, *byte*, *decrypt*}. Since Q9 has a programming language ‘*java*’, the two terms ‘*php*’ and ‘*ruby*’ may probably be unexpected by users.

Based on the above analysis, Chatbot4QR uses *WE* as the question retrieval model. However, the performance of *WE* is limited by the quality of queries. When a query is vague, e.g., missing important technical details, *WE* cannot retrieve accurate questions. As users may have varied technical background, Chatbot4QR uses an interactive approach to assisting users in refining queries by asking CQs that are generated according to the missing technical details in a query. The user’s feedback to CQs can accurately represent their technical requirements on the queries and contribute to retrieving relevant SO questions.

TABLE 20

The Average Numbers of CQs and the Average Ratios of Useful CQs That are Generated by Chatbot4QR and *ConstantBot* for the 50 Queries

Approach	Avg. #CQs	Avg. Ratio of Useful CQs
ConstantBot	2.7	0.446
Chatbot4QR	5.1	0.608

Table 17 and Fig. 9 show that Chatbot4QR helps the participants obtain much better results than using Web search engines alone for at least 48 percent of their allocated queries. The SO questions retrieved by Chatbot4QR were chosen as the best results by 16 of the 20 participants for the largest proportion of queries. For some queries, the participants obtained the best results using Web search engines by reformulating the queries with their feedback to CQs. These results demonstrate that Chatbot4QR can (1) retrieve desired SO questions for users after helping them refine the queries and (2) help users better understand their queries and obtain better results using Web search engines.

6.2 Why Not Use A Constant Chatbot?

Chatbot4QR is designed to generate different CQs for queries based on the existing technical details mentioned in a query and an initial set of similar SO questions retrieved for the query. *Is this design necessary? Can we use a constant chatbot that always asks several fixed CQs for queries?* To answer these questions, we implemented a constant chatbot, denoted as *ConstantBot*, which focuses on asking for four types of technical details given a query, namely the programming language, framework, and the versions of the two technique types. More specifically, *ConstantBot* first asks a CQ “*What programming language does your problem refer to?*”. If a user provides a programming language, e.g., Java, then *ConstantBot* further asks for the version of Java using a CQ “*Can you specify the version of java?*”. Then, *ConstantBot* asks for a framework using “*If you are using a framework, please specify:*”, as well as the version of a possible framework given by the user.

We asked the 20 participants in PG1 and PG2 to evaluate the CQs asked by *ConstantBot* for each allocated query. Similar to the *user study 3* conducted in Section 5.2, the participants rated each CQ by five grades 0-4 (as defined in Section 5.1) and gave feedback to the useful CQs. After the evaluation, we asked the participants to provide some comments about *ConstantBot*. Then, we retrieved the top ten similar SO questions using Eq. (1) by leveraging each participant’s feedback to a query. The participants evaluated the relevance of the questions that were not evaluated before by five grades 0-4, as defined in Section 5.1.

Table 20 presents the average number of CQs and the average ratio of useful CQs that are asked by *ConstantBot* for the 50 queries. Table 21 presents the Pre@k and NDCG@k performance of the retrieved SO questions, as well as the improvement degrees and the maximum significant ratios of Chatbot4QR over *ConstantBot*. From Table 20, we find that on average *ConstantBot* asked 2.7 CQs for a query; and the ratio of useful CQs is 44.6 percent, which is much lower than that of Chatbot4QR (i.e., 60.8 percent). By analyzing the evaluation results of the 50 queries, there are 11 queries (i.e., Q4,

TABLE 21

Evaluation of the SO Questions Retrieved by Chatbot4QR and *ConstantBot*, and the Improvement Degrees and the Maximum Significant Ratios of Chatbot4QR Over *ConstantBot*

	Pre@1	Pre@5	NDCG@1	NDCG@5
ConstantBot	0.766	0.596	0.687	0.661
Chatbot4QR	0.838	0.670	0.765	0.731
ImpD(%) (p, SigR(%))	9.4% (0.05, 20.0%)	12.3% (0.05, 55.0%)	11.3% (0.05, 30.0%)	10.5% (0.05, 35.0%)

Q9, Q17, Q23, Q27, Q29, Q32, Q33, Q39, Q46, and Q50) that have no useful CQ as evaluated by the participants. The 11 queries contain a specific programming language, e.g., Java in Q4; and the participants are not interested in looking for a framework. Two major comments about *ConstantBot* given by the participants are: (1) *ConstantBot* still asks for a programming language when a query already has a programming language; and (2) unlike Chatbot4QR, *ConstantBot* cannot help recognize some technical details that are useful but missed in a query, e.g., databases and libraries. From Table 21, we find that in terms of Pre@k and NDCG@k (k = 1 and 5) metrics, Chatbot4QR improves *ConstantBot* by 9.4-12.3 percent; and the improvement is statistically significant for 20-55 percent participants. Based on the analysis results, we can conclude that it is not appropriate to use a constant chatbot for the interactive query refinement and question retrieval.

6.3 Learning Effect From Interacting With Chatbot4QR

In Section 5.5, considering that the participants can learn to recognize some missing technical details in queries from the interaction with Chatbot4QR, we first asked the participants to search results for queries before interacting with Chatbot4QR. This can avoid the impact of the participants’ learning effect on their Web search results using the original queries.

It is worth mentioning that the learning effect is good for users in practice. After interacting with Chatbot4QR for several times, users, especially the novices, can learn to formulate high-quality queries with necessary technical details for retrieving questions from SO or other resources from general-purpose Web search engines (e.g., Google). Because of the learning effect, users can ask better questions in Q&A sites by describing their problems with a clear technical context, which can lead to better answers. Moreover, there are too many techniques (e.g., libraries) available on the Web; and it is difficult for users, even for experienced developers, to know every possible technique. Chatbot4QR may help users, including both novices and experienced developers, discover unknown or better techniques for some programming tasks.

6.4 Application Scenarios of Chatbot4QR

Chatbot4QR can be applied in the following two scenarios:

- Chatbot4QR can be implemented as a browser plugin. When a user inputs a query to Web search engines, the plugin detects the missing technical details in the

query. If there are missing technical details, the plugin informs the user that the query has a quality issue. Then, the user can choose to interact with our chatbot. After the interaction, our chatbot recommends the top ten similar SO questions. Moreover, the user can use their feedback to CQs to reformulate the query for Web search.

- In the literature, many technical tasks, such as answer summarization [6] and API recommendation [5] rely on the quality of the top similar SO questions retrieved for queries. Chatbot4QR could be used to improve the performance of question retrieval, which will contribute to better results of the tasks.

6.5 Participants' Comments About Chatbot4QR

In the experiments, we encouraged the participants to provide comments about Chatbot4QR. We summarize several major positive and negative aspects of the comments.

- *Positive Comments*

- PC1. *The chatbot is good! It can really help me figure out some important technical details missed in the queries. And, the final retrieved results are more satisfactory.*
- PC2. *Using the generated CQs to refine queries is more straightforward and systematic than manually picking up relevant information scents in the search results.*
- PC3. *The chatbot is flexible and fast, and most of the asked CQs are really closely related to a query.*
- PC4. *Although it may ask some unfamiliar techniques for me, it still helps me get a better understanding of the query as well as some other possibly useful libraries. I'd like to try it later.*

- *Negative Comments*

- NC1. *Some CQs are unnecessary because the asked information can be inferred from some keywords in the query. For example, the CQ "What programming language, e.g., java or c#, does your problem refer to?" asked for the query Q35 "Using LINQ to extract ints from a list of strings" is useless because 'LINQ' is based on C#.*
- NC2. *There are a bit too many CQs for some queries. Although the chatbot allows me to skip and terminate, I suggest that you can limit the number of CQs for a query, e.g., five.*

According to the comments, Chatbot4QR can assist the participants in refining queries and retrieving more desired results (PC1 and PC2). Additionally, Chatbot4QR can help the participants better understand the queries and discover some possibly useful techniques (PC4). The efficiency of Chatbot4QR is also acceptable (PC3). However, there still remain some issues. For example, Chatbot4QR cannot filter unnecessary CQs based on the existing information in queries (NC1). To solve this issue, we need to mine the relationships among techniques, e.g., what frameworks and libraries are related to a specific programming language. Moreover, the participants suggest us to limit the number of CQs asked for a query (NC2).

To validate the suggestion in NC2, we generated a list of the CQs evaluated by a participant for a query. The CQ list was ranked by the orders of the CQs prompted to the

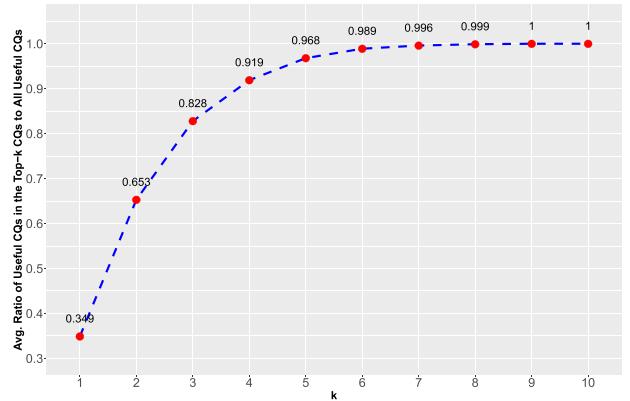


Fig. 10. The average ratios of useful CQs that are contained in the top- k CQs prompted to the participants to all useful CQs for the 50 queries.

participant during the interaction with Chatbot4QR. We counted the number of useful CQs in the top- k CQs of the list, and measured the ratio of useful CQs in the top- k to all useful CQs that are evaluated by the participant for the query. We set k from 1 to 10 (the maximum number of CQs in all CQ lists). For each k , we measured the average ratio of useful CQs in all CQ lists, as shown in Fig. 10. On average, 96.8 percent of the useful CQs of a query are contained in the top five CQs prompted to a participant, indicating that it is suitable to limit the number of CQs asked for a query as 5.

6.6 Error Analysis of Chatbot4QR

Although it has been validated that Chatbot4QR can effectively generate useful CQs and recommend relevant SO questions for queries, we find two error scenarios of Chatbot4QR from the participants' evaluation results as follows.

1. As reported in the comment NC1, Chatbot4QR may generate wrong CQs for a query. Despite the wrong CQ asked for the query Q35 in NC1, the CQ "What programming language, e.g., javascript or python, does your problem refer to?" generated for the query Q18 (see Table 11) is also useless since the technical term 'NPM' in Q18 is highly related to JavaScript. During the interaction with a participant, Chatbot4QR cannot dynamically filter unsuitable CQs or technical terms appearing in CQs based on the participant's feedback. For example, for the query Q42 shown in Table 11, after the participant P7 answered the CQ "What programming language, e.g., java or c#, does your problem refer to?" with Python, the subsequent CQ "Are you using .net? (y/n), or some other frameworks." became unsuitable as '.net' is a C# framework. The CQ should be revised by replacing '.net' with a Python framework appearing in the initial top- n similar questions retrieved for Q42. If there is no such a Python framework, the CQ can be removed. Through our analysis, the errors are caused by the fact that Chatbot4QR currently has no knowledge about the relationships between techniques, e.g., NPM is related to JavaScript and .net is related to C#. In the future, we plan to mine knowledge about the relationships among SO tags and integrate the knowledge to Chatbot4QR.
2. Chatbot4QR may produce worse question recommendation lists than the two-phase method for some

TABLE 22

Two Kinds of the Top Five SO Questions Retrieved for the Query Q5; and the Relevance of Questions Evaluated by the Participant P17

Result Type	The Top Five SO Questions			Relevance
	Question ID	Question Title	Question Tags	
Initial	20045940	Inserting multiple rows in database	java, database	4
	47244614	Inserting Data in Multiple Tables in Hibernate	java, hibernate, jpa	3
	22553920	Insert into two tables in two different database	java, spring, hibernate, jpa	2
	39383049	How to insert data to multiple table at once in hibernate using java	java, mysql, hibernate	3
	22472292	How to insert new items with Hibernate?	java, mysql, hibernate	2
Final	25485086	how to insert new row in hibernate framework?	java, mysql, sql, hibernate	3
	23200729	Records in DB are not one by one. Hibernate	java, mysql, sql, hibernate	1
	39383049	How to insert data to multiple table at once in hibernate using java	java, mysql, hibernate	3
	31583737	hibernate: how to select all rows in a table	java, mysql, sql, hibernate, postgresql	2
	22472292	How to insert new items with Hibernate?	java, mysql, hibernate	2

"Initial" represents the top five questions retrieved using our two-phase method. "Final" represents the top five questions retrieved using Chatbot4QR by leveraging P17's feedback to the CQs of Q5.

queries. For example, for the query Q5 "*How to insert multiple rows into database using hibernate?*", the participant P17 provided three kinds of positive feedback, i.e., {'java 8', 'mysql', 'sql'}, to the CQs. However, the final top ten SO questions refined by incorporating the feedback are worse than the initial top ten questions. Table 22 presents the initial and the final top five questions retrieved for Q5, as well as the relevance of the questions evaluated by P17. By analyzing the results, the performance of the final top five questions is decreased as some questions (e.g., the question '23200729') are irrelevant to the query task, but they match all the feedback, and therefore the rankings of such questions are over-promoted. To correct such errors, our future work will aim to optimize the weights of different types of technical feedback in Eq. (1) according to their contributions to the question retrieval (see Table 15).

6.7 Threats to Validity

Threats to internal validity relate to two aspects in this work: (1) the errors in the implementation of Chatbot4QR and the baseline approaches and (2) the participants' bias during the experiments.

As for the aspect (1), we carefully checked the implementation code of our Chatbot4QR prototype. Considering that there could be noises in the tag assignments of SO questions, which may affect the CQ generation of Chatbot4QR, we built the question repository by requiring that each question has an accepted answer and a positive score. Moreover, we ensured that the experimental queries and their duplicates were not included in the repository. Although our experimental queries were built from the titles of SO questions, it may not be a serious problem as it is a common experimental setup used in previous work [5], [6], [8], [20], [32], [33]. For the four baselines *EVPI*, *Lucene*, *WE*, and *TR*, we directly used the open-source code. For the other two baselines *WN* and *QECK*, we carefully re-implemented them according to the details presented in the papers [8], [12]. Therefore, there is little threat to the implementation of the approaches.

As for the aspect (2), we recruited the participants who are interested in our work and have 2-11 years of programming experience. We adopted several strategies to mitigate the participants' bias in the steps that require manual efforts. For the categorization of SO tags, we used two iterations of a card sorting approach. Each iteration step was independently conducted by the first two co-authors of the paper; then they worked together with an invited postdoc to discuss the disagreements to obtain the final results. We asked the participants to search results for the queries using Web search engines before interacting with Chatbot4QR, in order to avoid the participants transferring the knowledge learned from our chatbot to enhance the original queries when they use Web search engines. Before evaluating CQs in the user studies 1 and 3, we launched a video conference with the participants to introduce our Chatbot4QR prototype, to ensure that they understood how to use the prototype for evaluation. In the video conference of the user study 1, we also explained the relevance judgement of SO questions to a query with a technical context. Moreover, at the beginning of our user studies, we explained to the participants about how to perform the user studies based on the existing technical details in queries and/or their technical background. It is possible that the participants may have difficulties in building the technical context for some queries as they may not be interested in the problems. In the future, we plan to develop Chatbot4QR as a plugin and deploy the plugin in companies, such as Hengtian, to validate whether Chatbot4QR can help developers retrieve better SO questions or other resources for technical problems.

Threats to external validity relate to the generalizability of experiment results. To alleviate this threat, we built a large-scale repository of 1.88 million SO questions. To conduct our user studies, we recruited 25 participants. Considering that the user studies require significant manual efforts, we built 50 experimental queries. The number of participants and the number of experimental queries are close to the existing user studies in the previous work [30], [32], [33], [34], [35]. The 25 participants have different years of programming experience and diverse familiar programming languages, as shown in Table 5. The 50 experimental queries

have diversity in the involved techniques, the complexity of problems, and the quality of expression (i.e., whether there are specified techniques or not), as explained in Section 4.2. The diversity of participants and queries can help improve the generalizability of our experiment results. In the future, we plan to further reduce this threat by extending the user studies with more participants and queries.

Threats to construct validity relate to the suitability of evaluation metrics. To reduce this threat, we used two popular metrics: Pre@k and NDCG@k, which are widely used to evaluate the ranking results in the fields of IR and software engineering [5], [6], [31], [32], [33], [42].

7 RELATED WORK

Question Retrieval in SO. Question retrieval is a key step for many knowledge search tasks in SO. A number of work retrieves similar SO questions for queries by leveraging the Lucene search engine [8] or word embedding techniques [3], [5], [6], [11]. For example, Nie *et al.* [8] proposed a code search approach by expanding queries with important keywords extracted from relevant SO question-and-answer pairs. Lucene is used for indexing and retrieving SO question-and-answer pairs. Xu *et al.* [6] proposed an approach named *Answerbot* to generating a summarized answer for a query by extracting important sentences from the answers of similar SO questions. It retrieves similar SO questions using a word embedding-based approach. Huang *et al.* [5] proposed an API recommendation approach named *BIKER*. A word embedding-based approach is also used for retrieving SO questions similar to a query. The recommended APIs are extracted from the answers of the top ten similar SO questions. The Lucene search engine is efficient but cannot handle the lexical gaps between SO questions and queries. Recently, the word embedding-based approach is widely used to bridge the lexical gaps and can achieve better performance. However, the existing work on question retrieval rarely considers an important issue in practice that the query can be inaccurately specified, which will lead to undesirable questions.

We propose a novel question retrieval approach which improves the word embedding-based approach in two main aspects: (1) a two-phase question retrieval approach is used to improve the efficiency by reducing the search space using Lucene before applying the word embedding-based approach; and (2) a chatbot is designed to interactively help users refine queries by asking several CQs related to the missing technical details in a query. The refined queries can contribute to retrieving more relevant SO questions.

Tag Recommendation in SO. SO encourages users to attach several (no more than five) tags to a question, which can help organize the tremendous amount of questions and facilitate the question retrieval [43]. However, the large set of more than 50 thousand SO tags imposes a huge burden for users to select a few appropriate tags for a question. Much attention has been paid to recommending relevant tags for SO questions [36], [37], [38], [43]. For example, Xia *et al.* [36] proposed an approach called *TagCombine* to finding relevant tags by composing three ranking components. Wang *et al.* [37] proposed a tag recommendation system by using the labeled Latent Dirichlet Allocation (LDA) modeling technique [44].

They analyzed the historical tag assignments and users of SO questions and the original tags provided by users. Zhou *et al.* [38] proposed a neural network approach to recommending tags, which leverages both textual descriptions and tags of SO questions.

Different from the tag recommendation (TR) work, our work focuses on determining missing technical details in a query based on the tags of similar SO questions. As evaluated in Section 5.3, the existing TR approaches may not be suitable for determining relevant tags for queries because of two main reasons. First, unlike a SO question that has a rich description (including the title, original tags, and body), a query typically consists of a few keywords, which makes it difficult to find relevant tags precisely. Second, even for the same query, different users may have personalized preferences of tags considering their different technical background (e.g., the preferred programming languages) or programming context (e.g., the platform the software is developed for). To address these challenging issues, given a query, we use a chatbot to interact with the user by asking several CQs with a candidate set of relevant tags extracted from the top- n similar SO questions, allowing the user to tell what tags they want.

Query Reformulation. The quality of queries has a great impact on the performance of IR systems. However, it is not an easy task to formulate a good query, which largely depends on the user's experience and their knowledge about the IR system [45]. A lot of work has been proposed to automatically reformulate queries by expanding them with relevant terms extracted from lexical databases (e.g., WordNet) or similar resources [8], [12], [15], [46], [47]. For example, Lu *et al.* [12] proposed to expand a query with the synonyms in WordNet for code search. Nie *et al.* [8] also proposed a code search approach by expanding queries with important keywords extracted from relevant SO question-and-answer pairs. A major limitation of automatic query expansion approaches is that there can be unexpected terms added to the query without user involvement, which will affect the quality of results.

To overcome the limitation, several work has recently been proposed to interactively help users refine queries [31], [48], [49]. For example, Zou *et al.* [48] proposed a personalized Web service recommendation approach, which can assist users in refining their requirements. The approach is based on a process knowledge base built from the available online resources. Guo *et al.* [49] proposed an interactive image search approach which uses a reinforcement learning model to capture the user's feedback on their desired image. The approach relies on the predefined feature set of images. It has been demonstrated that these interactive query refinement approaches can help find desired results for users. In contrast to these work, we propose an interactive query refinement approach to assisting users in clarifying the missing technical details in queries, in order to improve the performance of question retrieval from technical Q&A sites. For this purpose, we build two technical knowledge bases, i.e., the categorization and multiple version-frequency information of SO tags.

8 CONCLUSION AND FUTURE WORK

Question retrieval plays an important role in acquiring knowledge from technical Q&A sites, e.g., SO. The existing

search engines provided in the Q&A sites and the state-of-the-art question retrieval approaches are insufficient to retrieve desired questions for users when the query is inaccurately specified. In this paper, we propose a chatbot, named Chatbot4QR, to interactively help users refine their queries for question retrieval. Chatbot4QR can accurately detect missing technical details in a query and interacts with the user by asking several CQs. The user's feedback to CQs is used to retrieve more relevant SO questions. The evaluation results of six user studies demonstrate the effectiveness and efficiency of Chatbot4QR.

To the best of our knowledge, it is the first work on the interactive query refinement for technical question retrieval. However, the current Chatbot4QR is still in infancy with some limited capability. In the current stage, Chatbot4QR focuses on helping users clarify 20 major types of techniques (see Table 1) and the versions of the techniques missed in a query. In the future, we will improve Chatbot4QR in two main directions: (1) we plan to use the possible solutions discussed in Section 6.6; and (2) we will mine knowledge on the differences (e.g., the frequency of use and performance) between the similar techniques (e.g., HashMap is more efficient than Hashtable [6]), so that Chatbot4QR could suggest better techniques when users intend to search for a less frequently used technique or an obsolete technique (e.g., Hashtable). Moreover, we plan to implement Chatbot4QR as a browser plugin to assist users in searching results for technical problems. When a user inputs a query to a Web search engine (e.g., the SO search engine or Google), the plugin can notify the user if there are missing technical details in the query. The user can interact with our chatbot to obtain the top ten similar SO questions and get insights for reformulating the query to search on the Web.

ACKNOWLEDGMENTS

This research was partially supported by the National Key R&D Program of China (No.2019YFB1600700), NSFC Program (No. 61972339), the Australian Research Councils Discovery Early Career Researcher Award (DECRA) (DE200100021), Ministry of Education, Singapore, under its Academic Research Fund Tier 2 (Award No.: MOE2019-T2-1-193), Natural Sciences and Engineering Research Council of Canada (NSERC), and Alibaba-Zhejiang University Joint Institute of Frontier Technologies.

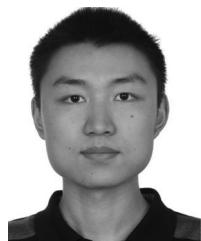
REFERENCES

- [1] C. Chen, X. Chen, J. Sun, Z. Xing, and G. Li, "Data-driven proactive policy assurance of post quality in community Q&A sites," *Proc. ACM Human-Comput. Interaction*, 2018, Art. no. 10.
- [2] D. Ye, Z. Xing, and N. Kapre, "The structure and dynamics of knowledge network in domain-specific q&a sites: A case study of stack overflow," *Empir. Softw. Eng.*, vol. 22, pp. 375–406, 2017.
- [3] B. Xu, Z. Xing, X. Xia, D. Lo, and S. Li, "Domain-specific cross-language relevant question retrieval," *Empir. Softw. Eng.*, vol. 23, pp. 1084–1122, 2018.
- [4] E. C. Campos, L. B. de Souza, and M. D. A. Maia, "Searching crowd knowledge to recommend solutions for API usage tasks," *J. Softw.: Evol. Process*, vol. 28, no. 10, pp. 863–892, 2016.
- [5] Q. Huang, X. Xia, Z. Xing, D. Lo, and X. Wang, "API method recommendation without worrying about the task-API knowledge gap," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 293–304.
- [6] B. Xu, Z. Xing, X. Xia, and D. Lo, "Answerbot: Automated generation of answer summary to developers' technical questions," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 706–716.
- [7] H. Yin, Z. Sun, Y. Sun, and W. Jiao, "A question-driven source code recommendation service based on stack overflow," in *Proc. IEEE World Congress Serv.*, 2019, pp. 358–359.
- [8] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, "Query expansion based on crowd knowledge for code search," *IEEE Trans. Services Comput.*, vol. 9, no. 5, pp. 771–783, Sep./Oct. 2016.
- [9] S. Haiduc, G. Bavota, A. Marcus, R. Oliveto, A. De Lucia, and T. Menzies, "Automatic query reformulations for text retrieval in software engineering," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 842–851.
- [10] T. Mikolov, I. Sutskever, K. Chen, G. S. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. Advances Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.
- [11] T. Nandi *et al.*, "IIT-UHH at semeval-2017 task 3: Exploring multiple features for community question answering and implicit dialogue identification," in *Proc. 11th Int. Workshop Semantic Eval.*, 2017, pp. 90–97.
- [12] M. Lu, X. Sun, S. Wang, D. Lo, and Y. Duan, "Query expansion via wordnet for effective code search," in *Proc. IEEE 22nd Int. Conf. Softw. Anal. Evol. Reengineering*, 2015, pp. 545–549.
- [13] M. M. Rahman *et al.*, "Evaluating how developers use general-purpose web-search for code retrieval," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 465–475.
- [14] G. A. Miller, *WordNet: An Electronic Lexical Database*. Cambridge, MA, USA: MIT Press, 1998.
- [15] F. Pérez, J. Font, L. Arcega, and C. Cetina, "Collaborative feature location in models through automatic query expansion," *Automated Softw. Eng.*, vol. 26, no. 1, pp. 161–202, 2019.
- [16] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: Mining and searching internet-scale software repositories," *Data Mining Knowl. Discovery*, vol. 18, no. 2, pp. 300–336, 2009.
- [17] S. Rao and H. DauméIII, "Learning to ask good questions: Ranking clarification questions using neural expected value of perfect information," in *Proc. 56th Annu. Meeting Assoc. Computat. Linguistics*, 2018, pp. 2737–2746.
- [18] "Google search," 2020. [Online]. Available: <https://www.google.com>
- [19] "Chatbot4qr release," 2019. [Online]. Available: <https://tinyurl.com/y6dgwwy5>
- [20] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 404–415.
- [21] "Tagwiki," 2019. [Online]. Available: <https://stackoverflow.com/tags>
- [22] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python: Analyzing Text with the Natural Language Toolkit*. Sebastopol, CA, USA: O'Reilly Media, 2009.
- [23] R. Rehurek and P. Sojka, "Software framework for topic modelling with large corpora," in *Proc. LREC 2010 Workshop New Challenges NLP Frameworks*, 2010, pp. 46–50.
- [24] D. Ye, Z. Xing, C. Y. Foo, Z. Q. Ang, J. Li, and N. Kapre, "Software-specific named entity recognition in software engineering social content," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reeng.*, 2016, pp. 90–101.
- [25] C. Chen, S. Gao, and Z. Xing, "Mining analogical libraries in q&a discussions—incorporating relational and categorical knowledge into word embedding," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reeng.*, 2016, pp. 338–348.
- [26] M. Nassif, C. Treude, and M. Robillard, "Automatically categorizing software technologies," *IEEE Trans. Softw. Eng.*, vol. 46, no. 1, pp. 20–32, Jan. 2020.
- [27] Q. Huang, X. Xia, D. Lo, and G. C. Murphy, "Automating intention mining," *IEEE Trans. Softw. Eng.*, to be published, doi: [10.1109/TSE.2018.2876340](https://doi.org/10.1109/TSE.2018.2876340).
- [28] J. L. Fleiss, "Measuring nominal scale agreement among many raters," *Psychol. Bull.*, vol. 76, no. 5, 1971, Art. no. 378.
- [29] M. Hearst, "Models of the information seeking process," Cambridge Univ. Press, pp. 64–90, 2013, doi: [10.1017/cbo9781139644082.004](https://doi.org/10.1017/cbo9781139644082.004).
- [30] H. Niu, I. Keivanloo, and Y. Zou, "Learning to rank code examples for code search engines," *Empir. Softw. Eng.*, vol. 22, no. 1, pp. 259–291, 2017.
- [31] N. Zhang, J. Wang, Y. Ma, K. He, Z. Li, and X. F. Liu, "Web service discovery based on goal-oriented query expansion," *J. Syst. Softw.*, vol. 142, pp. 73–91, 2018.

- [32] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.*, 2018, pp. 933–944.
- [33] X. Li, Z. Wang, Q. Wang, S. Yan, T. Xie, and H. Mei, "Relationship-aware code search for javascript frameworks," in *Proc. 24th ACM SIGSOFT Int. Symp. Foundations Softw. Eng.*, 2016, pp. 690–701.
- [34] J. Zhang, H. Jiang, Z. Ren, T. Zhang, and Z. Huang, "Enriching API documentation with code samples and usage scenarios from crowd knowledge," *IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2019.2919304.
- [35] A. Alami, M. L. Cohn, and A. Wasowski, "Why does code review work for open source software communities?" in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 1073–1083.
- [36] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *Proc. 10th Working Conf. Mining Softw. Repositories*, 2013, pp. 287–296.
- [37] S. Wang, D. Lo, B. Vasilescu, and A. Serebrenik, "Entagrec++: An enhanced tag recommendation system for software information sites," *Empir. Softw. Eng.*, vol. 23, pp. 800–832, 2018.
- [38] J. Liu, P. Zhou, Z. Yang, X. Liu, and J. Grundy, "FastTagRec: Fast tag recommendation for software information sites," *Automated Softw. Eng.*, vol. 25, no. 4, pp. 675–701, 2018.
- [39] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bull.*, vol. 1, no. 6, pp. 80–83, 1945.
- [40] S. E. Sim, M. Umarji, S. Ratanatayanon, and C. V. Lopes, "How well do search engines support code retrieval on the web?" *ACM Trans. Softw. Eng. Methodol.*, vol. 21, no. 1, 2011, Art. no. 4.
- [41] "Jsoup," 2019. [Online]. Available: <https://jsoup.org>
- [42] N. Zhang, J. Wang, K. He, Z. Li, and Y. Huang, "Mining and clustering service goals for restful service discovery," *Knowl. Inf. Syst.*, vol. 58, no. 3, pp. 669–700, 2019.
- [43] P. Zhou, J. Liu, Z. Yang, and G. Zhou, "Scalable tag recommendation for software information sites," in *Proc. IEEE 24th Int. Conf. Softw. Anal. Evol. Reeng.*, 2017, pp. 272–282.
- [44] G. Boudaer and J. Loeckx, "Enriching topic modelling with users' histories for improving tag prediction in q&a systems," in *Proc. 25th Int. Conf. Companion World Wide Web*, 2016, pp. 669–672.
- [45] J. A. Rodriguez Perez and J. M. Jose, "Predicting query performance in microblog retrieval," in *Proc. 37th Int. ACM SIGIR Conf. Res. Develop. Inf. Retrieval*, 2014, pp. 1183–1186.
- [46] C. Lucchese, F. M. Nardini, R. Perego, R. Trani, and R. Venturini, "Efficient and effective query expansion for web search," in *Proc. 27th ACM Int. Conf. Inf. Knowl. Manage.*, 2018, pp. 1551–1554.
- [47] M. M. Rahman and C. Roy, "Effective reformulation of query for code search using crowdsourced knowledge and extra-large data analytics," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 473–484.
- [48] P. K. Venkatesh, S. Wang, Y. Zou, and J. W. Ng, "A personalized assistant framework for service recommendation," in *Proc. IEEE Int. Conf. Serv. Comput.*, 2017, pp. 92–99.
- [49] X. Guo, H. Wu, Y. Cheng, S. Rennie, G. Tesauro, and R. Feris, "Dialog-based interactive image retrieval," in *Proc. Advances Neural Inf. Process. Syst.*, 2018, pp. 678–688.



Neng Zhang received the bachelor's and PhD degrees in software engineering from Wuhan University, China, in 2012 and 2017. He is currently a postdoctoral assistant researcher with Zhejiang University, China. His research interests include mining software repositories, empirical software engineering, services computing, human-computer interaction, and knowledge-based systems.



Qiao Huang received the bachelor's and master's degrees in computer science and software engineering from Zhejiang University, Hangzhou, Zhejiang, in 2012 and 2016. He is currently working toward the PhD degree in the College of Computer Science and Technology, Zhejiang University, China. His current research interests include mining software repositories and empirical software engineering.



Xin Xia received the bachelor's and PhD degrees in computer science and software engineering from Zhejiang University, in 2009 and 2014, respectively. He is an ARC DECRA fellow and a lecturer with the Faculty of Information Technology, Monash University, Australia. Prior to joining Monash University, he was a postdoctoral research fellow with the software practices lab, University of British Columbia, in Canada, and a research assistant professor with Zhejiang University, in China. To help developers and testers improve their productivity, his current research focuses on mining and analyzing rich data in software repositories to uncover interesting and actionable information. For more information, please visit <https://xin-xia.github.io>.



Ying Zou is the Canada research chair in Software Evolution. She is a professor with the Department of Electrical and Computer Engineering, and cross-appointed to the School of Computing at Queens University in Canada. She is a visiting scientist of IBM Centers for Advanced Studies, IBM Canada. Her research interests include software engineering, software reengineering, software reverse engineering, software maintenance, and service oriented architecture. For more information, please visit <http://post.queensu.ca/zou/>.



David Lo received the PhD degree in computer science from the National University of Singapore, in 2008. He is a ACM Distinguished member and an associate professor of Information Systems at Singapore Management University. His research interests include the intersection of software engineering and data science, encompassing socio-technical aspects and analysis of different kinds of software artefacts, with the goal of improving software quality and developer productivity. His work has been published in premier and major conferences and journals in the area of software engineering, AI, and cybersecurity.



Zhenchang Xing is an associate professor with the Research School of Computer Science, Australian National University. Previously, he was an assistant professor with the School of Computer Science and Engineering, Nanyang Technological University, Singapore, from 2012–2016. Before joining NTU, he was a Lee Kuan Yew research fellow with the School of Computing, National University of Singapore from 2009–2012. His current research interests include interdisciplinary areas of software engineering, human-computer interaction, and applied AI. He has more than 130 publications in peer-refereed journals and conference proceedings, and has received several distinguished paper awards from top software engineering conferences, including two ACM SIGSOFT distinguished paper awards and two IEEE TCSE distinguished paper awards. He regularly serves on the organization and program committees of the top software engineering conferences including ICSE, FSE, ASE, ICSME, and he is the program committee co-chair for ICSME2020. He is the associate editor of *Journal of Software: Evolution and Process*.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

Conditional Quantitative Program Analysis

Mitchell Gerrard^{ID}, Mateus Borges, Matthew B. Dwyer^{ID}, *Fellow, IEEE*, and Antonio Filieri^{ID}

Abstract—Standards for certifying safety-critical systems have evolved to permit the inclusion of evidence generated by program analysis and verification techniques. The past decade has witnessed the development of several program analyses that are capable of computing guarantees on bounds for the probability of failure. This paper develops a novel program analysis framework, CQA, that combines evidence from different underlying analyses to compute bounds on failure probability. It reports on an evaluation of different CQA-enabled analyses and implementations of state-of-the-art quantitative analyses to evaluate their relative strengths and weaknesses. To conduct this evaluation, we filter an existing verification benchmark to reflect certification evidence generation challenges. Our evaluation across the resulting set of 136 C programs, totaling more than 385k SLOC, each with a probability of failure below 10^{-4} , demonstrates how CQA extends the state-of-the-art. The CQA infrastructure, including tools, subjects, and generated data is publicly available at bitbucket.org/mgerrard/cqa.

Index Terms—Program analysis, model counting, symbolic execution, conditional analysis, software reliability, software certification

1 INTRODUCTION

MODERN safety-critical systems are software-intensive. While such systems undergo traditional verification and validation processes to detect and remove faults, they also go through a certification process that aims to demonstrate their absence. International standards for such systems establish requirements for certifying the software's contribution to overall system safety across a range of domains including: avionics [1], industrial robotics [2], personal care robotics [3], railway [4], automotive [5], and medical software [6]. Meeting these standards is essential, but they present substantial verification and validation challenges above and beyond those of traditional software [7].

Safety certification standards vary, but all represent a complex undertaking that includes, for example, demonstration of bi-directional traceability between requirements and implementation elements and achieving rigorous forms of implementation coverage. It comes as no surprise that the primary means of demonstrating that an implementation meets a safety requirement is achieved through testing. In fact, testing is used in myriad ways across the breadth of application domains and associated standards—Nair *et al.* [8] identify 13 different forms of testing evidence that can be incorporated into safety arguments. For example, structural coverage evidence, such as MC/DC that is required for avionics software [1], robustness evidence, such as that which is achieved using fault-injection to meet automotive standards [5], and reliability evidence, such as that which is

required to certify functions to IEC 61508 safety integrity levels (SIL) [9].

The increasing cost-effectiveness of automated formal methods and static analyses has led certification standards, e.g., DO-333 [10], and researchers to explore the types of evidence they can contribute to safety arguments to complement evidence from testing, e.g., [11]. In the context of a safety argument, such methods tend to provide *all or nothing* evidence—they can prove a property, e.g., through sound overapproximating model checking [12], or they cannot.

In this paper, we investigate combinations of static analysis methods that can provide a more gradual, quantitative form of evidence that can contribute to safety arguments. Our work is motivated by Ladkin and Littlewood's call for the increasing use of statistical evaluation in the certification of critical software [13], [14]. Their perspective is motivated by the fact that IEC 61508 defines SIL levels in statistical terms, e.g., a SIL level 4 function has an average probability of failure of less than 10^{-4} per invocation,¹ yet few cost-effective test methods exist to directly provide such evidence.

The challenges of testing ultra-reliable systems have long been known. Butler and Finelli [15] observed that achieving confidence in a very low probability of failure requires an exorbitant amount of testing. This challenge has been mitigated to an extent by advances in underlying technologies, e.g., high-fidelity simulation systems that can run in faster than real time and that can be executed in parallel [16], yet testing for high, much less ultra, reliability remains a significant obstacle.

Our insight is that two complementary forms of static analysis, when combined synergistically, yield a cost-effective method for demonstrating that functions achieve extremely low probability of failure. For completing subjects across the study, one instantiation of this technique

• Mitchell Gerrard and Matthew B. Dwyer are with the Department of Computer Science, University of Virginia, Charlottesville, VA 22904 USA.
E-mail: {mitchelljgerrard, matthewbdwyer}@virginia.edu.

• Mateus Borges and Antonio Filieri are with the Department of Computing, Imperial College London, London SW7 2AZ, U.K.
E-mail: {m.borges, a.filieri}@imperial.ac.uk.

Manuscript received 23 Apr. 2020; revised 30 July 2020; accepted 10 Aug. 2020. Date of publication 14 Aug. 2020; date of current version 18 Apr. 2022. (Corresponding author: Mitchell Gerrard.)
Recommended for acceptance by M. Pradel.
Digital Object Identifier no. 10.1109/TSE.2020.3016778

1. This is for functions that are invoked relatively infrequently which is referred to as *low demand* in the standard; functions invoked frequently or continuously frame requirements in terms of the number of failure-free hours of operation.

computes a mean and median probability of failure below 10^{-10} and 10^{-38} , respectively. This would be sufficient to easily discharge the evidentiary requirements for a low demand SIL 4 function.

The first analysis targets the fact that a key component of the cost of reliability testing comes from the need to *resample* equivalent program behavior. It is not obvious that two inputs will lead to equivalent behavior from a black box perspective, but when testing is permitted to observe the internal behavior of software, equivalence can be detected. This is precisely what symbolic execution techniques do [17] and reliability-focused extensions to symbolic execution can quantify the probability mass of a set of equivalent inputs [18], [19], [20]. This allows a single non-failing test input to accumulate all of the probability mass associated with its equivalent behaviors, which can greatly accelerate the process of reaching a reliability threshold.

The second analysis targets the fact that when systems enter the certification process they have already been thoroughly validated [7]. Our insight is that in this setting one can formulate a sound static analysis to partition the program input space into two subspaces—one that *may lead to failure* and one that *definitely does not lead to failure* [21], [22]. The latter of these can be skipped entirely when performing the above reliability analysis and the former can be used to *condition* the application of the reliability analysis, allowing it to focus on a smaller region of program behavior to maximize its cost-effectiveness.

In this paper, we study how these two analyses can be blended to create a new form of quantitative static analyses that can produce *guaranteed bounds on the probability of violating a safety property*. Unlike statistical methods [23], [24], which can only produce a probabilistic *confidence* on the soundness of the results based on statistics on the outcome of many test runs of the program,² the static analyses we focus on in this paper provide mathematically sound *guarantees* on the bounds for the probability of violations, and thus meet the strict evidentiary requirements for the above standards, e.g., [10].

Quantitative static program analysis has been studied for more than two decades, e.g., [27], [28], but only recently have fully automated techniques been developed that can scale to non-trivial code bases. Researchers have built on developments in increasingly scalable path-sensitive analyses, e.g., [29], [30], [31], [32], and increasingly scalable techniques for model counting of logical formulae, e.g., [33], [34], [35], [36], [37], [38], [39], to produce several families of techniques which we term *probabilistic symbolic execution* (PSE) [18], [19] and *statistical symbolic execution* (SSE) [20].

These techniques hint at the potential of combining non-quantitative program analyses, like symbolic execution, with quantitative analysis techniques, like model counting. We take this a step further in presenting a novel algorithmic framework for *conditional quantitative program analysis* (CQA) that blends evidence from multiple static analyses to extend the scalability, accuracy, and applicability of quantitative program analysis.

The history of combining non-quantitative static analyses to improve cost-effectiveness dates back at least three decades, e.g., [40]. In recent work, the open-source ALPACA framework [22], [41] implements an alternating conditional analysis (ACA) that combines 9 different C static analyzers to precisely characterize the regions of a program’s execution space that always satisfy (or always violate) a given property. Whereas individual analyzers may be limited in their ability to cope with aspects of a program or state-space structure, ACA harvests and blends their partial results to produce a comprehensive description of program behavior. The key to CQA is the insight that ACA-computed descriptions—rendered as logical constraints formulated over program input variables—can be leveraged to focus the application of different forms of quantitative analyses, which has the potential to make them more efficient and more accurate.

Understanding the potential improvements that the algorithmic variants of the CQA framework offer relative to existing state-of-the-art quantitative static analyses, such as [18], [19], [20], requires empirical evaluation. Unfortunately, no benchmarks exist that focus on the specific challenges in evidence generation for certification of safety-critical software systems.

Developing a broad and representative benchmark for this class of problems is a worthwhile pursuit, but in this work we only take a modest first step by customizing an existing verification benchmark—SV-COMP [42]. The benchmark is designed to stress automated static analysis and verification tools, but to reflect certification challenges, benchmark programs should exhibit low-probability property violations—like those that might slip through development into a certification process. In Section 4 we describe the systematic selection of 136 C programs, comprising more than 385,000 SLOC, for which the probability of a property violation is less than 10^{-4} . This threshold was chosen because it corresponds to the failure probability threshold required to meet IEC 61508’s SIL 4 standard. As our evaluation reveals, CQA is capable of establishing a much lower probability of failure than the SIL 4 requirement and can produce probability guarantees that were previously thought to be completely infeasible to achieve [15], e.g., less than 10^{-35} in under 15 minutes on Problem10_1label148.

The next section presents background and the prior work on quantitative and conditional program analysis on which we build. Section 3 presents the foundations of the CQA framework. Section 4 presents an evaluation that explores the algorithmic tradeoffs between CQA and existing approaches and demonstrates that CQA extends the state-of-the-art. We discuss related work in Section 5 and future work in Section 6.

2 BACKGROUND

2.1 Reachability Guarantees

A program’s semantics can be formalized as a transition system $\langle Q, q_0, \rightarrow \rangle$, where Q is a set of states (mappings from variables to values), $q_0 \in Q$ is the initial state, and $\rightarrow \subseteq Q \times Q$ is a transition relation between states and their possible successors. The *state space* is the set of all possible configurations in this transition system. The reachability problem is that of determining whether there exists a path in a

² For a property ψ , a probabilistic guarantee is of the form $Pr(p_{\neg\psi} \in [a, b]) \geq \delta$, where $p_{\neg\psi}$ estimates the probability of violating ψ and $\delta < 1$ is a *confidence* value bounding the probability of the produced interval being incorrect (e.g., [24], [25], [26]).

program's transition system from the initial state to a target state that satisfies property ψ ; we will call these target states ψ -states. Most program analysis questions can be framed as reachability problems [43].

In this paper we will refer to two kinds of reachability guarantees using different terms that refer to these distinct kinds; we will group and italicize the related terms in this paragraph. The first kind of guarantee is based on an *overapproximation* of the program state space, and gives a proof that a ψ -state is unreachable. We refer to this as a *safety* proof, produced by a *may* analysis that reasons about the *necessary* conditions on reaching a ψ -state. The second kind of guarantee is based on an *underapproximation* of the state space, and gives a proof that a ψ -state is in fact reachable. This is a *soundness* proof, produced by a *must* analysis that reasons about the *sufficient* conditions on reaching a ψ -state. More formally, given a program p , the first kind of guarantee is equivalent to the statement $p \models \neg\psi$, while the second to $p \not\models \neg\psi$.³

While the classical reachability problem is given as an existential query (does a path exist or not?), this work considers its generalization: find *all* paths from q_0 —the start state—that could reach a ψ -state. Because there may be many program paths that lead to a ψ -state, we will characterize these sets of paths in terms of *intervals*, defined in the following subsection.

2.2 Logical Intervals

Any given program can contain large state spaces, such that portions of it cannot be exactly characterized in an efficient manner. Overapproximating the state space can simplify the model of complex state spaces in ways that make this more amenable to efficient reasoning [45], [46]. In general, program analyzers can approximate a program's ψ -state reachability within over- and underapproximate bounds, which we define as a logical interval.⁴

Informally, a logical interval I is a characterization of the input subdomain that exhibits behaviors reaching ψ -states. This characterization consists of a *lower bound* (\underline{I}) guaranteed to be subsumed by a program's true ψ -state reachability, R_ψ , and an *upper bound* (\bar{I}) guaranteed to subsume R_ψ . A sufficient condition on p 's inputs reaching a ψ -state is given by \underline{I} , while \bar{I} comprises a necessary condition for the same. In the remainder of the text, we will use the term *interval* and logical interval interchangeably.

Definition 1 (Logical Interval). A logical interval, $I_\psi = [\underline{I}_\psi, \bar{I}_\psi]$, is a pair of logical predicates on the inputs that semantically bound a program's ψ -state reachability, R_ψ , such that $\underline{I}_\psi \Rightarrow R_\psi \Rightarrow \bar{I}_\psi$.

This interval lies between two extremes: $\underline{I}_\psi = \bar{I}_\psi = R_\psi$ (most informative, exact characterization) and $[\text{false}, \text{true}]$ (non-informative).

3. A reachability property is a property stating that a particular state can be reached, while a safety property states that a “bad” state is *never* reached; so reachability properties can be seen as the negation of a safety property [44]. Because the focus of this work is on characterizing reachable states, safety is discussed in a negated sense in relation to ψ .

4. A logical interval that semantically bounds program behavior is discussed in [21], where it is referred to as a *comprehensive failure characterization*.

A partition on I_ψ induces a set of *disjoint intervals*. For the remainder of the text, the ψ subscript will denote the logical interval over the entire program space, and those without the ψ subscript will denote disjoint intervals.

Definition 2 (Disjointness). Two intervals are considered disjoint when their upper bounds are disjoint, i.e., $\bar{I}_i \wedge \bar{I}_j \equiv \emptyset$; because upper bounds subsume their lower bounds, all lower bounds will also be disjoint.

Note that the upper bound of each individual disjoint interval is not a necessary condition on reaching a ψ -state over the entire program space: there may exist other intervals, so the negation of one upper bound would by definition include the other intervals. However, it is a necessary condition on reaching a ψ -state within its respective partition of the input domain. The lower bound of a disjoint interval, in contrast, is a sufficient condition in reaching a ψ -state both in the entire program space as well as within its respective partition of the input domain.

2.3 Conditional Program Analysis

The idea of conditional quantitative analysis builds on the principle of conditional verification [47]. Conditional verification combines the strengths of multiple verification procedures by excluding the portion of state space that one procedure has verified as safe from the search space of the others. This allows each application of a verification procedure to focus only on parts of the state space which have not yet been covered by any of the others. Because most verification procedures are specialized (or optimized) on specific program features (e.g., via specific abstractions or memory models), composing their partial results may enable each to leverage their respective strengths on portions of a program exhibiting specific features, leaving simpler residual problems after each application.

2.4 Basic Probability Definitions

The possible outcomes of an experiment are called *elementary events*. For example, flipping a coin can produce one of two elementary events: heads or tails. Elementary events are mutually exclusive, and the set of all elementary events is called the *sample space*. An *event* is a set of elementary events.

Definition 3 (Probability distribution). Let Ω be the sample space of an experiment. A probability distribution on Ω is a function associating to each subset of Ω a real value between 0 and 1: $Pr : 2^\Omega \rightarrow [0, 1]$

that satisfies the Kolmogorov's probability axioms [48]:

- $Pr(e) \geq 0$ for every elementary event e
- $Pr(\Omega) = 1$
- $Pr(A \cup B) = Pr(A) + Pr(B)$ for all events A, B where $A \cap B = \emptyset$ (Ω, Pr) is called the probability space.

Definition 4 (Conditional probability). Let (Ω, Pr) be a probability space. Let A and B be events with $Pr(B) > 0$. The conditional probability of A given B (i.e., the probability of A assuming B has occurred) is defined as $Pr(A | B) = \frac{Pr(A \cap B)}{Pr(B)}$.

Definition 5 (Law of total probability). Let (Ω, Pr) be a probability space and $\{E_i | i = 1, 2, 3, \dots, n\}$ be a finite

partition of Ω , where $\forall i. Pr(E_i) > 0$. Then, for any event A , $Pr(A) = \sum_{i=1}^n Pr(A | E_i) \cdot Pr(E_i)$.

The probability mass function yields the probability that a discrete random variable is equal to some value. The *probability mass* of a set of values is the summation of the probability mass function applied to its elements. A logical formula is the characteristic function of the set of its models, thus the probability mass of a logical formula is the probability mass of each of its models.

2.5 Quantifying Logical Formulae

Given a logical formula and a probability distribution over the free variables in the formula, there are a growing number of cost-effective methods to estimate the probability mass contained in the formula. Some of these estimates are exact, e.g., when the formula lies in the domain of linear integer arithmetic [33]; in other cases the accuracy of estimates are probabilistically bounded [49], [50].

3 CONDITIONAL QUANTITATIVE ANALYSIS

The problem this paper addresses is determining *how likely* it is that a ψ -state is reached within some program. Unlike the classical formulation of reachability, where either a path to a ψ -state exists or not, quantifying the probability of reaching a ψ -state requires considering many paths, in general.

One approach to solving the problem of how to quantify the probability mass of inputs reaching a ψ -state is via brute force, i.e., enumerate all program paths and sum the mass of those reaching a ψ -state, as proposed in [18]. Another approach is to fuzz the input space to get a statistical bound on the probability of reaching a ψ -state [51], [52], [53]. The first approach suffers when the state space is large, while the latter suffers when the probability of reaching a ψ -state is exceedingly rare.

The solution advocated in this paper is to first determine *which* regions of the input space can lead to a ψ -state, and only quantify this reduced portion of the program state space. We call this a *conditional quantitative analysis*.

Algorithm 1 defines the conditional quantitative analysis algorithm using the specified internal functions. CQA takes as input a program, a reachability property (ψ), and a probability distribution over the program's input variables; and outputs a quantitative characterization that bounds the input probability mass reaching ψ . The “lower” quantity (l) provides a sound lower bound on ψ -reaching inputs, i.e. l quantifies the sufficient conditions on inputs reaching ψ , while the “upper” quantity (u) provides a safe upper bound on ψ -reaching inputs, i.e., u quantifies the necessary conditions.

CQA begins by initializing the lower and upper quantifications to zero in line 2. The function *generate_intervals* on line 3 takes a program and a reachability property and returns a nonempty, finite set of intervals that describe the portions of the state space that *may/must* reach a ψ -state. The intervals must satisfy the safety and disjointness properties of Definitions 1 and 2, respectively. A trivial implementation of *generate_intervals* would return the set $\{\text{false}, \text{true}\}$, which contains a single interval that implies *all* program behavior—thus safely but trivially bounding

ψ -state reachability. A more informative implementation of *generate_intervals* could, for instance, return the set $\{[\alpha \wedge \beta, \alpha], [\neg\alpha \wedge \gamma, \neg\alpha \wedge \gamma]\}$, which contains two intervals: the first denotes that a ψ -state *must* be reached when the program inputs satisfy $\alpha \wedge \beta$ (the interval's lower bound), and that a ψ -state *may* be reached when the program inputs satisfy α (the interval's upper bound); the second denotes an interval whose lower and upper bound coincide—this means that a ψ -state *must* be reached when the inputs satisfy $\neg\alpha \wedge \gamma$.

Algorithm 1. Conditional Quantitative Analysis

Input: Program P , reach. property ψ , prob. distribution X
Output: Lower/upper quant. of ψ -reaching inputs $[l, u]$

- 1: **procedure** CQA(P, ψ, X)
- 2: $[l, u] \leftarrow [0, 0]$
- 3: $\mathcal{I} \leftarrow \text{generate_intervals}(P, \psi)$
- 4: **for each** $I \in \mathcal{I}$ **do**
- 5: **if** $\underline{I} \equiv \bar{I}$ **then**
- 6: $e \leftarrow \text{estimate}(\bar{I}, X)$
- 7: $[l, u] += [e, e]$
- 8: **else**
- 9: $[l, u] += \text{quantify_in_bounds}(P, \psi, I, X)$
- 10: **return** $[l, u]$

Specifications for CQA Functions

generate_intervals (P, ψ)

Input: Program P , reachability property ψ

Output: Set of disjoint logical intervals (see Definitions 1 and 2)

estimate (f, X)

Input: Logical formula f , probability distribution X

Output: Estimate of $Pr(f)$

quantify_in_bounds (P, ψ, I, X)

Input: Prog. P , reach. prop. ψ , interval I , prob. dist. X

Output: [Overappr. of $Pr(\underline{I})$, Underappr. of $Pr(\bar{I})$]

Lines 4–9 use these computed intervals to focus quantification efforts within the state space delineated by a given interval. There are two cases to consider when deciding how to quantify ψ -state reachability within an interval. In one case—line 5—the lower and upper bounds coincide, so we can directly quantify the formula given by its upper (or equivalent lower) bound. This is done by the function *estimate*, which takes as input a logical formula and a probability distribution, and computes either an exact or a (probabilistically bounded) approximate estimate— e —of the probability mass that satisfies the given formula. In the case of coinciding bounds, the computed probability mass e is necessary and sufficient, so e is added to both the lower and upper quantifications in line 7.

The other possibility—line 8—is that an interval's bounds do not coincide, in which case we must explore the region of the state space between the lower and upper bounds in order to quantify the ψ -reaching probability mass contained within the interval. The function in line 9—*quantify_in_bounds*—takes as input a program, a reachability property, an interval, and a probability distribution, and returns a pair whose first part quantifies a safe overapproximation of the probability mass reaching \underline{I} —the lower bound of I , and whose second part quantifies a safe underapproximation of the probability mass reaching \bar{I} —the upper bound of interval I . The output in line 10 gives the lower

and upper bounds on the probability mass of reaching a ψ -state.

Theorem 1 (Termination). *Algorithm 1 terminates if generate_intervals, estimate and quantify_in_bounds terminate.*

Proof. The loop in lines 4–9 will run a bounded number of times because \mathcal{I} is a finite set, so if each function terminates, Algorithm 1 will terminate. All functions called within CQA are required to terminate due to both time and space bounds, guaranteeing that CQA terminates. \square

Theorem 2 (Correctness). *Algorithm 1 terminates with l providing a sound lower bound and u a safe upper bound on the probability mass of a program reaching a ψ -state, given some input probability distribution.*

Proof. The correctness of CQA’s output follows from four observations: (1) the function *generate_intervals* requires all program behavior reaching a ψ -state to be contained within \mathcal{I} , implying all probability mass of reaching a ψ -state is also in \mathcal{I} , (2) the same function requires the intervals of \mathcal{I} to be disjoint, so no probability mass is quantified twice, (3) the functions *estimate* and *quantify_in_bounds* are required to yield a sound underapproximation and a safe overapproximation on the probability of reaching a ψ -state within the state space bounded by an interval, and (4) the estimates on each interval’s probability mass are accumulated in l and u in either lines 7 or 9. So upon termination of the loop in line 10, l and u correctly provide lower and upper bounds on the probability mass of reaching a ψ -state. \square

The remainder of this section discusses some of the possible instantiations of the functions used within Algorithm 1. These instantiations are used in the evaluation of CQA, discussed in Section 4.

3.1 Instantiation of *generate_intervals*

One non-trivial instantiation of *generate_intervals* is given by the framework of an *alternating conditional analysis*, which computes a sound characterization of all the ways a program either may or must satisfy some property. This is computed by alternating between over- and underapproximate analyses, conditioning analyses to ignore portions of the program that have already been analyzed, and combining the results of state-of-the-art analysis tools in a portfolio run in parallel. ACA is based on the framework introduced in [21], which was generalized in [22].

We will present the general idea of ACA by way of example. Given the program in Listing 1, ACA begins by running a portfolio of static analysis tools that search for a path reaching a call to *psi()*. Suppose a tool provides evidence of a path to *psi()*, then an underapproximate analyzer uses this evidence to characterize the path as either valid or spurious. This characterized space is now accounted for and does not need to be analyzed again.

Suppose the given evidence describes constraints on x that drive execution to one of the calls to *psi()*: $x < 0$. ACA now checks if there are other paths to *psi()*. To do so, blocking instrumentation is injected into the initial program—via *assume* statements—to condition analyzers to avoid this already-covered space.

Listing 1. Initial Program

```
main()
{
    x = read();
    y = read();

    if (x < 0)
        psi();
    elif ((x > 9) && (x < y*y))
        psi();
}
```

Listing 2. Conditioning 1

```
main()
{
    x = read();
    y = read();

    assume (!(x < 0));

    if (x < 0)
        psi();
    elif ((x > 9) && (x < y*y))
        psi();
}
```

Listing 3. Conditioning 2

```
main()
{
    x = read();
    y = read();

    assume !(x < 0);
    assume !( (x > 9) && (x < y*y) );

    if (x < 0)
        psi();
    elif ((x > 9) && (x < y*y))
        psi();
}
```

Listing 4. Conditioning 3

```
main()
{
    x = read();
    y = read();

    assume !(x < 0);
    assume !(x > 9);

    if (x < 0)
        psi();
    elif ((x > 9) && (x < y*y))
        psi();
}
```

ACA runs the portfolio of analysis tools on the instrumented program of Listing 2. This time an overapproximate analyzer claims that a different path to `psi()` is reachable, with the given evidence of $x > 9 \wedge x < y * y$. A second `assume` statement is injected into the program, and the tool portfolio is run on the program in Listing 3.

This time around, an overapproximate analyzer—upon encountering the relational and nonlinear expression in the second `assume` statement—overapproximates the state space and declares that the call to `psi()` within the `elif` block is still reachable. An underapproximator deems this evidence spurious. In order to reach a fixed point, ACA now generalizes the second blocking clause by relaxing the constraint of $x > 9 \wedge x < y * y$ to be $x > 9$; this relaxed conditioning is shown in the second `assume` statement in Listing 4.

The tool portfolio is run on the program of Listing 4, and an overapproximate analyzer declares that `psi()` is unreachable given the conditioned program. Because the safety proof comes from an overapproximate analysis, it is assumed correct, and ACA terminates with two intervals describing inputs constraints leading to `psi()`. One interval has coinciding lower and upper bounds that exactly describe input constraints leading to `psi()`: $\underline{I}_1 \equiv \bar{I}_1 \equiv (x < 0)$, and another has noncoinciding lower and upper bounds: $\underline{I}_2 \equiv (x > 9 \wedge x < y * y) \Rightarrow \bar{I}_2 \equiv (x > 9)$. Note that, while \underline{I}_2 defines constraints on inputs that *must* reach `psi()`, this is not the case for \bar{I}_2 , which includes concrete inputs that do not reach `psi()`, e.g., $x \equiv 10 \wedge y \equiv 3$. This simple example does not cover all cases of ACA; see [21] for an exhaustive case analysis.

The intervals computed by ACA satisfy the requirements of `generate_intervals`, in that its output consists of a *lower bound* that is guaranteed to be subsumed by all reachable paths (the *must* information), and an *upper bound* that is guaranteed to subsume all reachable paths (the *may* information).

Across the 136 subjects in this study, the instantiation of `generate_intervals` returns intervals with noncoinciding upper and lower bounds on 129 subjects; 15 of these subjects are composed of multiple intervals. The upper and lower bounds coincide on the remaining 7 subjects, one of which is composed of multiple intervals.

3.2 Instantiation of estimate

Inputs can be assumed distributed uniformly over their domains or according to a given input distribution called a *usage profile* [19]. For simplicity, a uniform distribution over the input domains will be assumed throughout the paper; extension to arbitrary usage profiles is orthogonal to our contributions and can be straightforwardly implemented as in [19] and [54].

For a finite input domain D , computing the probabilities $Pr(c)$ of a constraint c can be reduced to computing the ratio between the number of solutions of $\#(c \wedge D)$ and the size of the domain $\#(D)$. Model counting procedures may in general be intractably complex [55]. Nonetheless, as with constraint solving problems, several algorithms are available for the efficient solution of specific fragments of the problem. Linear integer constraints can be efficiently and exactly solved using Barvinok’s algorithm [33] (with off-the-shelf

implementations including Latte [34] and Barvinok [36]). Nonlinear constraints over numerical variables can rely on progress in convex analysis [56], interval constraint paving [35], [57], and the approximate methods developed in both program analysis [54], [58], [59] and statistical machine learning [60]. Model counting over string domains includes exact counters for regular languages [38], exact bound computation [37], and mixed string/numerical counters [61].

More general—though usually more expensive— $\#SAT$ and $\#SMT$ solvers also exist for model counting over mixed theories (e.g., [62], [63], [64]). The growing research interest in model counting for program analysis and artificial intelligence is driving a substantial research effort discovering new fragments of theories where efficient solutions are possible (e.g., [65], [66]) and are expected to directly benefit quantitative program analysis in the coming years.

As model counting is an orthogonal concern for CQA (equally impacting all the existing quantitative analysis techniques), for the implementations reproduced in this paper we will focus on linear integer constraints.

3.3 Instantiations of *quantify_in_bounds*

Following the principle of conditional program analysis, because all behaviors outside the logical interval of the entire program space $I_\psi = [\underline{I}_\psi, \bar{I}_\psi]$ have already been analyzed by `generate_intervals`,⁵ quantification techniques can focus only on the residual behaviors, i.e., program paths satisfying the assumption $\alpha \equiv \neg \underline{I}_\psi \wedge \bar{I}_\psi$.

In probabilistic terms, this can be formalized as computing the conditional probability $Pr(I_\psi | \alpha)$, instead of $Pr(I_\psi)$, as the analysis is restricted to the subspace of the sample space that encloses all inputs satisfying α . Recalling Definition 4 (conditional probability), for each disjoint interval I_i and its corresponding assumption α_i we obtain

$$Pr(I_\psi | \alpha_i) = \frac{Pr(I_\psi \wedge \alpha_i)}{Pr(\alpha_i)}. \quad (1)$$

The total probability $Pr(I_\psi)$ is the result of summing over the conditional probabilities multiplied by the respective $Pr(\alpha_i)$, as in Definition 5.

We now discuss three possible instantiations of *quantify_in_bounds*; the first being a direct application of model counting and the last two based on symbolic execution. Each satisfies the requirements of *quantify_in_bounds* in that it: (1) safely overapproximates its lower bound \underline{I} and safely underapproximates an interval’s upper bound \bar{I} , and (2) is amenable to conditioning. The second requirement is fulfilled simply by each technique respecting the semantics of `assume` statements. As the intervals of \mathcal{I} are guaranteed to be disjoint, the conditioning comes for free, because each technique will reason only about the state space encoded by the disjoint formulae.

⁵ Recall that `generate_intervals` characterizes behaviors that *must* reach a ψ -state, i.e., \underline{I}_ψ ; and upon termination guarantees that all inputs outside the upper bound, i.e., $\neg \bar{I}_\psi$, *must not* reach a ψ -state. These two behaviors lie outside the interval I_ψ , in that they lie “below” the lower bound and “above” the upper bound, and have already been characterized by some analyzer; so they may safely be ignored by *quantify_in_bounds*.

We will refer to CQA whose *quantify_in_bounds* has been instantiated with model counting, probabilistic symbolic execution, and statistical symbolic execution, as $\text{CQA}_{\#}$, CQA_{pse} , and CQA_{sse} , respectively.

3.3.1 Counting Lower and Upper Bounds

The set of logical intervals \mathcal{I} can be passed to a model counting procedure that converts its bounds into a numerical interval describing the contributions to the probability mass, i.e., by summing over the counts of the lower bounds and the counts of the upper bounds.

Applying model counting to an interval's lower bound and upper bound yields quantifications of these formulae that are either exact or (probabilistically bounded) over- and underapproximate estimates on the probability mass defined by \underline{I} and \overline{I} , respectively; this satisfies the postcondition of *quantify_in_bounds*.

This instantiation of *quantify_in_bounds* is straightforward but can be very imprecise depending on the precision of the bounds. The potential imprecision can be improved upon by focusing underapproximate analyses within the lower and upper bounds. Any behavior that is analyzed within the interval is guaranteed to improve the bounds, e.g., if ψ is found, then the lower bound raises, and if $\neg\psi$ is found, the upper bound drops. Below we discuss two techniques that offer this kind of improved precision.

3.3.2 Probabilistic Symbolic Execution

Probabilistic symbolic execution extends symbolic execution by computing the probability of each execution path being triggered by a program input [18]. As in standard symbolic execution, a program execution path (or program path) is uniquely identified by its path condition.

Program paths are classified in one of three ways, as: (a) reaching a target state (denoted with a ψ superscript), (b) missing a target state (denoted with a $\neg\psi$ superscript), or (c) truncated (denoted with a ? superscript) because the execution along a path failed to reach a target state within the prescribed depth or time limit. This classification of the execution paths induces a partition on the path conditions into three sets: (a) $PC^{\psi} = \{PC_1^{\psi}, \dots, PC_i^{\psi}\}$, (b) $PC^{\neg\psi} = \{PC_1^{\neg\psi}, \dots, PC_j^{\neg\psi}\}$, and (c) $PC^? = \{PC_1^?, \dots, PC_k^?\}$. Because each path gives rise to a disjoint path condition, a lower bound on the probability of reaching a target state is given by

$$Pr^{\psi}(P) = \sum_i Pr(PC_i^{\psi}). \quad (2)$$

The probability of missing a target state, $Pr^{\neg\psi}(P)$, and the truncated probability, $Pr^?(P)$, have analogous definitions. As the union of path conditions induces a partition of all execution paths, the sum of these probabilities is 1, entailing that with any two of them the third can be computed arithmetically.

When the analysis of PSE is focused within an interval I , the path conditions within PC^{ψ} will raise the lower bound, or safely overapproximate \underline{I} 's probability mass; and the path conditions within $PC^{\neg\psi}$ will dually drop the upper bound by the probability mass contained in the set, so PSE safely underapproximates \overline{I} 's probability mass. Thus PSE satisfies the postcondition of *quantify_in_bounds*.

3.3.3 Statistical Symbolic Execution

PSE inherits the path explosion issue of symbolic execution, in addition to the cost of quantification procedures, which may prevent it from exploring the entirety of a program's executions.

Statistical symbolic execution [20] addresses the problem of incomplete exploration by prioritizing the exploration of paths based on their probability mass. At each branch point, SSE computes the probability of moving towards each of the possible successor states by quantifying the solution space of the branch condition. The exact probability of a path is computable after its complete traversal.

As a sampled path is completely characterized by its path condition, it does not need to be sampled again, and can be pruned out of the sample space. This pruning allows for faster convergence of the statistical estimator to a prescribed accuracy, deterministically guaranteed termination, and more efficient coverage of rare events (i.e., execution paths with low probability).

The classification of program paths into three distinct sets is the same as with PSE, as is the way in which the probability mass within the sets PC^{ψ} and $PC^{\neg\psi}$ is used to satisfy the postcondition of *quantify_in_bounds*.

4 EVALUATION

In this section, we explore the cost and effectiveness of conditional quantitative analysis compared to the state-of-the-art—namely probabilistic symbolic execution and statistical symbolic execution—as well as how bounds within CQA can focus further analyses. Our goal is to provide information about the runtime, accuracy, and cost of quantification across techniques, when applied in a context that captures challenges for evidence generation for safety certification of software components. To this end, we look at three research questions.

RQ1 *How cost-effective is CQA compared to the state-of-the-art in terms of runtime and accuracy of probabilistic bounds?*

RQ2 *How much quantification can be avoided using CQA?*

RQ3 *How does conditioning within CQA progressively focus quantitative analysis?*

4.1 Algorithm Implementations

To maximize consistency in our evaluation of different algorithmic approaches we implemented them on top of a common set of existing analyses. We use the open-source ALPACA from [41] since it is the only tool we are aware of that implements a nontrivial instantiation of *generate_intervals* for computing sound conditional information to drive CQA. We made minor modifications to invoke a model counter [36] to count computed intervals. ALPACA uses the CIVL symbolic executor for C programs [67]; it also enabled us to use a portfolio of 9 different analyzers that participated in the SV-COMP'19 competition for the synthesis of conditioning intervals, namely: CBMC [68], CPA-BAM-BnB [69], CPA-Seq [70], ESBMC-incr [71], PeSCo [72], Symbiotic [73], UltimateAutomizer [74], UltimateTaipan [75], and VeriAbs [76].

For consistency with ALPACA, our implementations of PSE and SSE both build on CIVL. PSE extends the default depth-first search in CIVL to report the current PC at the end of

each path, which will then be categorized as either PC^ψ , $PC^{-\psi}$, or $PC^?$: PC^ψ for paths that end due to a call to the SV-COMP function `__VERIFIER_error()`, $PC^{-\psi}$ for executions terminating within the time bound without invoking the error function, and $PC^?$ for paths that hit the search depth limit. Only PC^ψ and $PC^{-\psi}$ paths are counted, since $\Pr(PC^?) = 1 - (\Pr(PC^{-\psi}) + \Pr(PC^\psi))$.

SSE is implemented following the design in [20], by annotating each explored node of the symbolic execution tree with the fraction of the input domain reaching it (quantifying the path condition up to the node). Transitions during SSE exploration are randomly chosen according to the relative probability mass of the direct successor nodes, that is, for each direct successor, the ratio between the fraction of domain that can reach it and the cumulative fraction of the domain that can reach any direct successor. On backtrack, either due to termination of a path or reaching the depth limit, SSE subtracts the probability of the final path condition from each node along the way up to the root; nodes with a probability of 0 are pruned from the tree and thus will not be visited again (intuitively, the probability mass annotating a symbolic node represents how much of the executions through the node have not been explored yet; when 0, all such executions have been explored and the node will not be sampled in subsequent runs). We use Barvinok [36] for model counting (default parameters, no timeout). Using Barvinok off-the-shelf limits our prototype to linear integer constraints.

Before counting, path conditions are first simplified using the Z3 solver [77] and then sliced into independent subproblems that do not share symbolic variables (following the slicing rules in [19]). The simplification step helps to mitigate the impact of Barvinok's internal transformation into Disjunctive Normal Form for inputs containing nested disjunctions.

4.2 Artifacts

All artifacts, including tools, subjects, and generated data are publicly available at bitbucket.org/mgerrard/cqa.

International standards establish *safety integrity levels* for safety-related functions. For the highest integrity level the standard imposes a probability of violation per invocation of less than 10^{-4} —*low demand* mode for SIL4 [9].⁶ The selection and filtering of subjects suitable for evaluating CQA was driven by this requirement that each subject has a probability of failure below 10^{-4} .

Our choice of building on ALPACA, which relies on analyzers that competed in SV-COMP, naturally led us to consider the SV-COMP benchmark suite [42] since the analyzers generally are able to process subjects in the benchmark and interpret SV-COMP annotation primitives, for example, `__VERIFIER_error()`, whose reachability defines ψ -states in our evaluation.

More specifically, we considered the sequential subjects from the benchmark that have property violations. From these we selected 1400 for which at least one of the 9 tools used in ALPACA could detect a violation within 15 minutes (SV-COMP competition timeout); note that this does not

6. More stringent probabilities are required for *high demand* contexts.

mean that ALPACA can characterize all violating inputs for subjects making it past this filter. From these, we filtered out subjects that could not be processed by the ALPACA, PSE, and SSE implementations: 412 subjects suffered from the inability to confirm a witness to a violation (a known problem in multiple SV-COMP tools), 34 subjects contained double or float data that is not supported by the model counters we used, 172 could not be handled due to incomplete support for C expressions in the ALPACA implementation, and 420 subjects could not be processed by CIVL because it either enforced strict C standards that were not met by the subjects, or the subject contained an unsupported feature. The remaining 369 subjects were run through ALPACA, PSE, and SSE implementations.

In specific domains or applications, inputs are usually assumed as generated by an input probability distribution. Unfortunately, this information is not available for any SV-COMP benchmark, even when the benchmarks are components of real software systems. Consequently, for our evaluation we assume a uniform input probability.⁷

For the remaining 369 subjects we ran the five considered quantitative analysis techniques to determine whether any could compute a lower bound on the probability of violation that exceeded 10^{-4} . Such subjects do not reflect the type of *rare* violations that make certification evidence challenging to produce, e.g., [13], [14], [15], so we removed them from our study. This resulted in a final set of 136 C subject programs which average 2833 SLOC—only 6 of these subjects have less than 100 SLOC and the largest is 9464 SLOC.

4.3 Results

We report the results of running CQA and (unconditioned) PSE and SEE on the 136 subjects in aggregated data; the full details of our experiments are included in the electronic appendix, which can be found on the Computer Society Digital Library at <http://doi.ieeecomputersociety.org/10.1109/TSE.2020.3016778>.⁸ These details include a variety of measures that capture characteristics of the subjects and their analysis, for example, the number of paths explored, the number of gray paths whose exploration was truncated at depth bounds, and the share of analysis time spent in model counting. We reference these measures in the discussion of our results below.

Setup. We ran our analyses on a 2x 16-core Intel Xeon Gold 6130 server with 64 GBs of RAM running Ubuntu Linux 18.04. We established a timeout of 90 minutes for running any of the implementations on a subject. It is not possible to determine the optimal depth limit for a given subject ahead of time, so for this evaluation we used a depth limit of 1,000 symbolic states for both PSE and SSE.

Results Overview. Table 1 provides an overview of our study results. There is a row for each algorithmic variant, where CQA has three variants depending on the technique used to instantiate `quantify_in_bounds`: $CQA_{\#}$, CQA_{pse} , and

7. Arbitrary discrete distributions can be reduced to mixtures of uniform ones over a partition of the finite domain, as in [19] therefore supporting arbitrary discrete profiles would add a linear complexity factor in the size of the partition; however, no input distribution is specified in SV-COMP.

8. Included with the submission and also available at bitbucket.org/mgerrard/cqa.

TABLE 1
Summary of Evaluation by Technique

	Most acc. \underline{I}	Most acc. \bar{I}	Avg. Time (s)	Run characteristics		
				Complete	T/O	Bounded
CQA _#	80	8	1,269	133	3	0
CQA _{pse}	113	77	2,488	40	32	76
CQA _{sse}	93	59	1,418	21	57	63
PSE	51	54	1,672	34	31	97
SSE	60	57	2,886	11	54	76

CQA_{sse}. The columns classify the performance of each technique by the number of subjects, out of 136, that share a particular outcome. The **Most acc. \underline{I}** and **Most acc. \bar{I}** columns report on the number of subjects on which a technique computes the most accurate lower bound, and upper bound, respectively. For the analyses that complete, we report the *Average* runtime in seconds. The final three columns report on different characteristics across runs. The *Complete* column lists the number of subjects for which the technique finishes the analysis without timing out or being depth limited. The *Timeout* column lists the number of subjects that the technique could not analyze within the 90-minute bound. The *Bounded* column lists the number of subjects for which the technique hit the depth limit—this only applies to analyses using PSE or SSE.

We note that a run of a technique may be both depth limited and timeout; for a given technique, the counts for complete, timeout, and depth limited need not sum to 136. If two techniques produce the same most-accurate bound (for either \bar{I} or \underline{I}), then both are counted as having produced the most-accurate bound; thus the columns reporting on accuracy in the table do not sum to 136.

RQ1 (Time/Accuracy). CQA improves the state-of-the-art in quantitative analysis by computing more accurate probability bounds than previous techniques, at a comparable cost. Across the study, both CQA_{pse} and CQA_{sse} produced more accurate lower and upper bounds than their unconditioned counterparts. All CQA variants produce a greater number of most-accurate lower bounds than the state-of-the-art. This is because the variety of analysis techniques used with *generate_intervals* can discover ψ -state reachability within state spaces in which unconditioned PSE/SSE find little to no ψ -state reachability.

With respect to the upper bound, the instantiation of *generate_intervals* used in this study computes upper bounds that, when quantified directly with a model counter, are too approximate to compete with the exhaustive techniques of PSE/SSE. Accordingly, CQA_# computes the most-accurate upper bound for the seven subjects in which the intervals given by *generate_intervals* coincide; the eighth subject is a degenerate case on which all techniques compute the same trivial \bar{I} . However, when *quantify_in_bounds* is instantiated with PSE and SSE, these approximate bounds allow CQA_{pse} and CQA_{sse} to produce 23 and 2 more most-accurate upper bounds than unconditioned PSE and SSE, respectively.

Both CQA_# and CQA_{sse} complete in less time, on average, than the state-of-the-art. The average runtime of CQA_{pse} is less time than SSE, but is nearly 14 minutes longer, on average, than PSE; the tradeoff for the longer runtime is an increase in the accuracy of bounds produced by CQA_{pse}.



Fig. 1. Linear diagram of overlap for most-accurate \underline{I} .

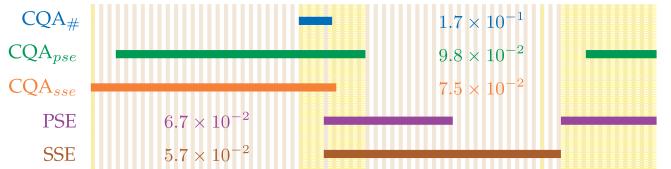


Fig. 2. Linear diagram of overlap for most-accurate \bar{I} . Sets of the most-accurate lower (Fig. 1) and upper (Fig. 2) bound for each technique are depicted as horizontal lines, and their intersection by overlapping vertical segments. Each subject is given by a vertical stripe; gold stripes are subjects on which an exhaustive technique completes. The numbers give a technique's average distance to the best bound.

Some of the increased accuracy in CQA_{pse} and CQA_{sse} is a result of the provided conditioning allowing these techniques to complete on more subjects than their unconditioned counterparts. The time spent in *generate_intervals* causes CQA_{pse} and CQA_{sse} to time out on 1 and 3 more subjects than PSE and SSE, respectively, but this time spent refining the conditioning allowed CQA_{pse} and CQA_{sse} to avoid being depth-bounded on 21 and 13 more subjects than PSE and SSE, respectively.

Though the CQA variants produce a greater number of most-accurate bounds than PSE and SSE, their strengths were found to be complementary across this study.

Figs. 1 and 2 use linear diagrams [78] to depict the overlap between techniques for achieving the most-accurate lower and upper bounds on given subjects, respectively. A linear diagram is an alternative to Venn diagrams in showing the intersection between sets, in which sets are depicted as horizontal lines across the diagram, and their intersection is given by overlapping vertical segments. Each subject is demarcated by a vertical stripe. For example, the lefthand side of Fig. 1's linear diagram tells us that CQA_{pse} alone computes the most-accurate lower bound for four subjects, then both CQA_{pse} and PSE compute the most-accurate lower bound for the next six subjects, and so on. Note that the i th stripe in Fig. 1 does not necessarily correspond to the same subject as the i th stripe in Fig. 2.

We highlight in gold those subjects on which some exhaustive technique (i.e., symbolic-execution-based) could complete. No technique always produces the most-accurate bound, so we give the average distance to that best bound as colored annotations laid over the linear diagram, e.g., whenever CQA_# does not produce the most-accurate lower bound, it is on average 1.7×10^{-11} away from that best bound.

The main takeaway from Figs. 1 and 2 is that, in the context of this evaluation, conditioned and unconditioned quantitative analyses can be seen as having complementary strengths in bounding the probability of reaching a ψ -state. This is especially apparent from Fig. 2, where the set of subjects for which CQA variants compute the most accurate upper bound are nearly disjoint from those on which PSE/SSE perform the best.

The differences in magnitude between the average distance to the most-accurate bounds between Figs. 1 and 2 are due to the fact that this study is done in the context of finding ψ -states that have a low probability of occurring. This means that even if a technique like PSE does not find any probability mass reaching a ψ -state, if the most-accurate lower bound is 10^{-27} , then PSE’s probability mass of 0 is still relatively close to the most-accurate lower bound. In contrast, if some technique times out or hits a depth bound, the vast majority of the probability mass involves paths that do *not* reach a ψ -state may remain unaccounted for—e.g., in the worst case for CQA techniques, all time is spent within *generate_intervals* and the upper bound is not lowered at all, as with *email_spec3_product29*. So if one technique does not complete for a subject on which another technique is particularly effective in collecting mass reaching $\neg\psi$ -states, the distance to the most-accurate upper bound can be many magnitudes more than to that of the most-accurate lower bound.

The disjointness of each interval returned by *generate_intervals* also offers the potential for parallelism through separate interval quantification. While it is possible to merge each separate interval into a single one and focus quantitative techniques within this larger interval—by quantifying within the space described by the disjunction of each upper bound and the negation of each of the lower bounds (see Section 3.3)—we observed a $2.9\times$ speedup within CQA_{pse} by quantifying each interval in parallel compared to doing so all at once.⁹

In terms of both runtime cost and effectiveness in computing accurate probability bounds, CQA is a clear improvement on the state-of-the-art. But the two need not be competitors. The complementary strengths of CQA variants and unconditioned PSE/SSE observed across this study recommends that both the conditioned *and* the unconditioned techniques should be applied, if possible.

RQ2 (Counter Calls). As pointed out in previous work on quantitative techniques [18], [20], in addition to the traditional cost of program analysis—e.g., constructing a program model, exploring its feasible branches, etc.—there is a significant component of the cost that goes into quantification, i.e., calls to a model counter. Being able to focus quantitative analyses on small subspaces of a program can significantly cut down on the number of calls made to a model counter. When an interval is exact, i.e., $\bar{I} \equiv \underline{I}$, this reduction can be drastic. (Note that, even if the probability mass within a noncoinciding interval is relatively small, this interval may still contain a large number of paths, necessitating a proportional amount of model count calls within *quantify_in_bounds*.)

We restrict this discussion to comparing techniques among common subjects that complete. When two techniques complete on a subject, they have produced the same probability bounds, so the overall work to compute the bounds is fixed, and we can compare quantification head-to-head, e.g., we can compare the exhaustive quantification done by PSE against that of CQA_{pse} , which divides its work

amongst ALPACA and PSE within intervals. Comparing on completing subjects allows us to evaluate the effect conditioning has on the cost of quantification. The possible improvement in quantification is related to how the probability mass outside of the conditioned intervals is distributed across paths. If the probability mass is concentrated in a single path outside of the intervals, then the improvement within CQA will be negligible (a single call saved); but if there are many such paths then the improvement can be substantial.

On the single subject with noncoinciding intervals for which both SSE and CQA_{sse} complete—*cdaudio_simpl11*—SSE spends 1,246 seconds issuing 48,758 counter queries, while CQA_{sse} spends 252 seconds issuing 18,360 queries. For 24 of the subjects with noncoinciding intervals on which both PSE and CQA_{pse} complete, CQA_{pse} reduces the number of counter queries and counter time by 15 percent—with CQA_{pse} issuing an average of 15,060 queries in 1,233 seconds, and PSE issuing an average of 17,412 queries in 1,424 seconds. The 25th subject with a noncoinciding interval on which both PSE and CQA_{pse} complete—*floppy_simpl13*—is an outlier in that CQA_{pse} issues 1,750 queries in 3973 seconds, while PSE issues 1,756 queries in only 368 seconds. The reason for the large difference in count times is a combination of the fact that this subject has a large number of symbolic variables and the conditioning includes a disjunctive formula, causing an exponential blowup in each query’s clauses and slowing down CQA_{pse} ’s overall quantification considerably. This is a scalability issue specific to the model counter we used for the experiments when handling disjunctive constraints in high-dimensional spaces. Different model counters may offer different scalability tradeoffs for different classes of constraints, e.g., [49].

When CQA provides intervals that *do* coincide, then a significant number of model counts can be avoided using CQA. This effect is observed on 7 of the 136 subjects in our study. On one of those subjects—*kbfiltr*—it is possible to characterize this reduction since $CQA_{\#}$, PSE, and SSE all complete; for the remaining 6, PSE and SSE do not complete. On the single subject for which CQA’s intervals coincide and both PSE and SSE complete—*kbfiltr*—, the CQA techniques spend under 1 second issuing 4 queries, while PSE spends 123 seconds issuing 600 queries, and SSE spends 211 seconds issuing 16,408 queries.

The relation between reduced calls to a model counter and reduced overall analysis time depends on the complexity of a program’s constraints. The subjects in this study have only linear integer constraints, meaning programs containing more complex constraints will only further highlight the benefit of fewer calls to a model counter.

RQ3 (Focusing). This research question will not compare against the state-of-the-art but will instead look at how the framework of CQA itself is used to progressively focus the accuracy of its computed logical intervals. We assume for this question that the computed intervals do not coincide, i.e., $\bar{I}_\psi \neq \underline{I}_\psi$, and that there is some uncertainty that can be progressively resolved. (If this were not the case, and the lower and upper bounds of I_ψ coincide, then the “focusing” is a few calls to a model counter, as discussed in RQ2.)

When two bounds of an interval do not coincide, this means there is some amount of probability mass implied by

9. Merging disjoint intervals into a single one causes a blowup of clauses-to-quantify due to the presence of disjunctions in the merged upper bound as well as disjunctions resulting from negating the conjuncted lower bounds, increasing the cost of constraint solving and model counting in our experiments.



Fig. 3. Signatures of conditioned PSE raising/reducing the lower/upper bound across different subjects.

the upper bound of which we are uncertain, i.e., we do not know if this mass leads to a ψ -state or not. CQA can focus further quantitative analyses within $\neg I_\psi \wedge \bar{I}_\psi$, as discussed in Section 3.3. Within this focused subspace of the subject, any probability mass proven to lead to a ψ -state effectively raises I_ψ (the lower bound), and mass proven to miss a ψ -state reduces \bar{I}_ψ (the upper). We will refer to the reduction of uncertainty as *tightening* an interval. How do further analyses tighten an interval over time, e.g., is the upper bound first reduced, followed by the lower?; is the rate of tightening linear? do intervals containing less probability mass get tightened in less time than those containing more mass?

Fig. 3 depicts how I_ψ is tightened from below and from above across a sample of four subjects¹⁰ whose bounds do not coincide; we will call each depiction a subject's signature. The x -axis of a signature gives the running time of a quantitative analysis, from 0 to 270 seconds. The y -axis of a signature ranges from 0 to 1, and the gray areas represent the probability mass whose uncertainty has been left unresolved at a given time. The y -axis is depicted on a log scale in order to visualize minuscule changes in probability mass, ranging from 2.3×10^{-17} up to 1. This means that an upper bound of 1×10^{-11} will look the same as an upper bound of 0.9, so there can be a dramatic tightening of bounds that appears as slight reductions on the logscale plot, e.g., the upper bound of signature d is eventually lowered to 4.7×10^{-10} , though no shift in the upper bound is apparent in logscale. In Fig. 3, the instantiated quantitative analysis is PSE. The visual steps are an artifact of PSE reporting its probability mass findings in 15-second intervals.

In the best case, a quantitative analysis will resolve all uncertainty, signified in a signature by the absence of gray after some time step. In the worst case, no uncertainty is resolved, and the gray area is not reduced at all.

The best case is visualized in both signatures b and c , though their respective uncertainties are resolved in different ways. In signature b , all probability mass collected accounts for inputs that do not lead to a ψ -state, and the upper bound is successively lowered. In signature c , probability mass accounting for both inputs that miss and inputs that lead to a ψ -state are collected in the first 15 second timestep, both raising the lower bound and lowering the upper bound; eventually just a sliver of uncertainty remains in a disjoint interval until PSE resolves this bit of probability mass.

Signatures a and d are examples of subjects whose uncertainty has not been resolved within a time bound. The uncertainty shown in signature a is tightened from above and below in distinct time steps, finally leaving a relatively small amount of probability mass unresolved. Signature d shows PSE raising the lower bound slightly, and, though

not apparent with the log scale, the upper bound is lowered to 4.7×10^{-10} , but the remaining probability mass is left unresolved.

The diversity of signatures indicates how, in relation to some property, the resolution of a state space's uncertainty can occur in quite unpredictable ways. Some of this unpredictability occurs because certain portions of the state space contain more paths to explore than other portions; but part of the unpredictability comes from the fact that probability mass is not distributed evenly across paths.

The amount of probability mass contained in an interval is not related to the number of paths explored in this interval. For instance, one interval contains 6 paths whose probabilities sum to 2.3×10^{-9} , while another interval contains 7,062 paths whose probabilities sum to the same amount. At the other end of the spectrum, one interval contains 16 paths whose probability mass covers most of the input space. This suggests that it is difficult to predict based on the amount of probability mass in unexplored intervals (given by: $(1 - \# \bar{I}) - \# L$, how long such an interval will take to explore).

4.4 Discussion

This subsection is a more anecdotal discussion of observations culled from the study.

We observed that PSE and SSE are cost-effective when there are paths of modest number and depth whose constraints are amenable to efficient quantification. This was the case for 33 of the subjects in the study. These range from 1082 to 2726 SLOC with between 878 and 9,032 paths—all of length less than 1,000. A representative example is `email_spec8_product15`, which took 1,996 seconds to analyze, of which 1,568 seconds was spent in quantification procedures, and computed an *exact* probability of reaching a violation of 4.7×10^{-38} .

When a subject contains significantly more paths, as in `email_spec1_productSimulator`, a 3236 SLOC subject with at least 19705 paths, PSE times out. PSE faces challenges with subjects like `Problem01_label115` where paths are deep and quantification is expensive. On this 580 line subject, PSE explored 29,562 complete paths, but was forced to abort the exploration of 42,098 after hitting the depth limit and spent 84 percent of its time quantifying path conditions.

Like PSE, SSE also performs well on the subjects with small state spaces—a few thousand paths of depth less than 1,000—but PSE always outperforms SSE on these cases (both explore the entire state space, but SSE has higher model counting overhead). The data reveal cases where SSE's ability to prioritize state-space exploration by probability mass has notable benefits. Both SSE and PSE timeout on `token_ring_08`. After analyzing 40,999 paths, PSE is able to reduce the upper bound to 2.3×10^{-9} , but it does not find any paths to a ψ -state, and its lower bound is not raised at all. In contrast SSE, only analyzes 1,275 paths before

¹⁰ We chose these four signatures as representatives of others with similar visual patterns. All signatures can be viewed at bitbucket.org/mgerrard/cqa_signatures.

timing out, but is able to reduce the upper bound to 4.7×10^{-10} and raise the upper bound to 2.2×10^{-19} .

Many benchmarks in the study mimic the structure of embedded control system components. They include a top-level REPL that reads an input at each iteration, then applies a cascade of filters to the inputs to determine how to update its internal state, and finally executes an action based on the input and state. The subjects vary in the size of their internal state, the number of filters they apply, the nature of their state updates, and the specific location of the property violation. They range in size from 580 to 9,464 SLOC and they have substantial state spaces, as evidenced by PSE’s exploration of more than 100k paths prior to timing out on Problem03_label143.

Our manual analysis of representatives from this group of subjects reveals a common structure to their violations. Properties are violated only when sequences of values satisfying specific constraints are read during iterations of the top-level REPL. For all of these subjects, there is an iteration bound on the REPL beyond which no violation will be exhibited. This give rise to an unbounded nonviolating state space.

It is no surprise then, that on all of these subjects PSE and SSE either timeout or reach a depth limit. We note that for subjects like these, a depth-limited symbolic execution to handle infinite loops changes the semantics of such long running subjects and may lead to unusable results (i.e., there may be violations beyond the depth bound that would not be detected and quantified). While the maximum probability of such deep violations must be smaller than the probability of the gray paths, because their execution has been truncated, in REPL control loops the total mass of gray paths may be significantly large, preventing PSE and SSE from obtaining tight violation probability bounds within a feasible search depth.

While the CQA variants also cannot produce exact bounds on this group of subjects, the conditioning provided allows symbolic execution to avoid many unbounded non-violating state spaces—the number of depth-limited paths is always less for CQA_{pse} and CQA_{sse} than their unconditioned counterparts. In many cases, the reduction of depth-limited paths is drastic: on Problem03_label126, PSE is depth-limited on 64,679 paths and yields an upper bound of 2.3×10^{-9} , while the conditioning given by *generate_intervals* allows CQA_{pse} to only hit 7176 depth-limited paths and yields an upper bound of 6.5×10^{-18} .

4.5 Limitations and Threats to Validity

Implementation. The main goal of this preliminary evaluation was to explore the capabilities of a proof-of-concept prototype of the mathematical framework behind CQA. Our implementation of PSE/SSE inherits all the limitations of the current version of cvl’s symbolic execution engine (e.g., strict conformance to C standards, limited support for non-integer domains, specific assumptions about the memory model) [79]. Our model counting interface delegates counting of linear integer constraints to Barvinok [36], after basic simplifications of the constraints via Z3 [77]; more advanced or specialized counting routines developed for established PSE/SSE analyzers may be faster.

Benchmark. Because there is no universally accepted specification format for properties and violation witnesses, to

maximize compatibility with the tools in ALPACA we used the SV-COMP benchmark. Filtering out subjects not analyzable by our prototype tool implementations and with high violation probability, left a corpus of 136 programs that were sufficient to highlight limitations of all the approaches we considered. We remark that despite the limitations of the artifacts studied in this work, they have been able to confirm both the limitations and the potential of CQA techniques that were expected from their mathematical formalization.

We caution the reader in making conclusions about the external validity of our findings. While this corpus of programs may be a starting point, clearly a broader set of programs, ideally accompanied with realistic input distribution specifications, will be needed to construct a comprehensive benchmark for assessing quantitative analysis tools.

Internal Validity. The ability to integrate available tools off-the-shelf¹¹ allowed us to develop prototype CQA implementations and assess their potential. However, the tools underlying ALPACA, PSE, and SSE are complex and highly-configurable. We use these tools in their default configuration and do not have control over their internals. We have not controlled for all of the factors that may influence their performance and this may impact the performance of CQA techniques. This is quite challenging and benchmarking the performance of static analysis tools and constraint solvers remains an open problem, e.g., [80], [81], [82], [83], [84]. Nevertheless, we took measures to cross check the probability intervals produced by all tools to confirm their consistency and we monitored for anomalies in underlying metrics reporting on the operation of the tools. After this check, we found no inconsistencies in reported bounds across the study.

4.6 A Benchmark for Analysis Techniques for High-Confidence Systems

The results of this study establish an absolute ground truth for 40 of the 136 subjects considered. These are the subjects on which some exhaustive technique (i.e., symbolic-execution-based) could complete. For those subjects on which none of the techniques could compute this ground truth, the probability mass of reaching a ψ -state is in general tightly bounded. There is a corpus of 135 examples on which the probability of reaching a ψ -state ranges from less than 4.7×10^{-10} down to 4.8×10^{-96} . (The 136th subject is a degenerate case in which all techniques compute the same trivial \bar{I} .)

Fig. 4 depicts the least upper bound computed by any technique across subjects via an impulse plot. The y -axis gives the upper bound on the probability mass reaching a ψ -state in logscale, and each “impulse” on the x -axis represents one of the 135 nondegenerate subjects. The impulses are sorted by descending heights of least upper bounds, where the probability mass above each least upper bound represents the probability mass that is guaranteed to be

11. Tools that participate in SV-COMP are able to interpret annotation primitives such as `__VERIFIER_error()`—whose reachability defines ψ -states in our evaluation, and `__VERIFIER_assume()`—which allows us to inject assumptions about program variables into the code. In this way, any analyzer competing in SV-COMP can be plugged into ALPACA off-the-shelf.

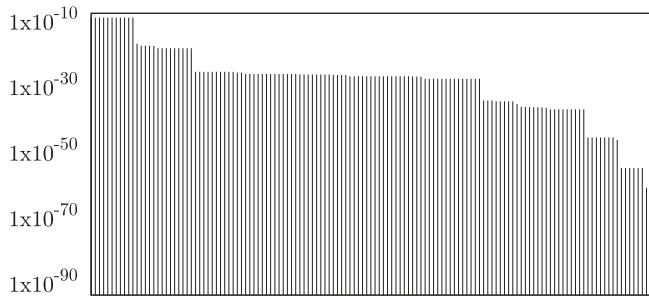


Fig. 4. Least upper bounds per subject across techniques.

safe. Fig. 4 demonstrates that the techniques in this study are able to compute highly accurate upper bounds on the probability of reaching a ψ -state for the given subjects, and we hope these bounds will soon be further lowered.

When analyzing the probability-of-failure in high-confidence systems, the upper bounds offer the most appealing guarantees, e.g., the assurance that your system will not fail outside of these bounds. For the state-of-the-art in quantitative analysis, the challenge and excitement lies in effectively reducing these upper bounds. While the techniques used within our study could not compute ground truths for all subjects, we hope this corpus will be used as a benchmark for other researchers to use in exploring analysis and testing for high-confidence systems.

5 RELATED RESEARCH

In this work we proposed a framework to use a combination of static analysis tools to synthesize conditions characterizing execution subspaces that *must* or *may not* reach a target state, and use this information to either bound the probability of reaching a target, or to condition subsequent path-sensitive quantitative analysis procedures for PSE or SSE.

Static program analysis of non-quantitative properties is a broad field, but we are applying the state-of-the-art rather than extending it. Our instantiation of *generate_intervals* using ACA (via ALPACA) provides us with ready access to a large and diverse set of analyzers [21], [22]. These include abstract interpretation based overapproximation tools, such as CPACHECKER, and SMT-based underapproximation tools, such as CBMC. The ACA framework builds upon research that combines may and must analyses [12], [85], [86], [87], [88], [89], [90]; applying these ideas is recommended in order to implement a nontrivial instantiation of *generate_intervals*. For CQA, the presence of overapproximating analyses is critical since they can summarize and classify entire sets of execution paths [91], which in turn can be quantified as a whole instead of requiring a more costly path-sensitive traversal of each path.

Conditioning is a method proposed in verification to combine the portions of state space confirmed as safe or failing using different techniques [47]. Researchers have proposed various methods to focus verification efforts on reduced portions of a program: passing state predicates between model checkers to restrict the considered state space [47], [92]; combining verification and systematic testing [93], [94]; transforming one program to another containing fewer execution paths while retaining the possible property violations of the original program [95], [96]. With

CQA, we aimed at providing a mathematical foundation linking logical conditioning to conditional probability theorems, thus enabling the instantiation of conditional verification in the area of quantitative analysis.

Quantitative analysis in software engineering has historically been focused on the analysis of probabilistic abstractions of architectural models via probabilistic model checking [97] or on the definition of probabilistic abstract interpretation methods [28], [98], [99]. The former can take advantage of efficient probabilistic model checkers [97], but requires a manual construction of the models, which are difficult to keep consistent with code implementations. The latter proves difficult to effectively generalize to the constructs of modern programming languages (no tools exist for industrial-strength languages, to the best of our knowledge). Probabilistic symbolic execution [18] is a recent technique exploiting symbolic execution to extract conditions on the input leading to target states. We presented techniques in this family in Section 3. Variations of PSE and SSE have also been used for exact/approximate reliability analysis [19], [100], performance analysis [101], and detection and automated exploitation of side-channel vulnerabilities in both regular and probabilistic programs [102], [103]. In this paper, we focused on non-probabilistic programs; investigating possible extensions of CQA to general probabilistic programs [104] and approximate computing [105], [106] is left as future work.

Testing [52], [53] and classic underapproximating analyses aim at producing actionable evidence to drive the debugging process and their use is complementary to verification and CQA for certification as they cannot prove the absence of errors or sound bounds on the probability of error [51]. As already discussed, statistical techniques can be coupled with uniformly random testing to obtain statistical bounds on error probability (e.g., [24], [25], [26]), however, the number of runs required to achieve high accuracy and confidence bounds may be prohibitive and rare paths (e.g., guarded by an equality condition like $x==42$) are unlikely to be explored. As a white-box analysis, CQA pays the scalability cost of static analyses which can limit its applicability to larger problems, for which weaker statistical guarantees are the only viable alternative [107]. Despite their limitations, CQA techniques have proven applicable to components of up 9,400 SLOC which is larger than component sizes suggested for safety critical systems [108].

6 CONCLUSION

We introduced *Conditional Quantitative Analysis*, which instantiates principles from conditional verification to allow logical evidence produced by non-quantitative static analyses to support more expensive quantitative analyses. CQA links logical conditioning used in verification to a conditional probability framework, enabling quantitative analyses to compute improved sound bounds on the probability of property violations. Our preliminary evaluation found evidence that CQA-enabled techniques extend the state-of-the-art in quantitative program analysis, highlighting the limitations and potential of several of them.

Our future work will seek to investigate further optimizations of the techniques enabling the CQA framework and

their interactions, including the direct use of quantitative information within the iterative process of ACA to greedily drive logical interval synthesis and refinement towards efficiently accumulating violation evidence with the largest probability mass.

More broadly, we believe it essential for the research community to focus on developing mathematically well-founded analysis techniques to support the certification of software in critical systems. We plan to work with researchers and practitioners developing such systems to build a broader and more representative benchmark that can drive future research advances.

ACKNOWLEDGMENTS

This material is based in part upon work supported by the U.S. National Science Foundation under Grants 1617916 and 1901769, by the U.S. Army Research Office under Grant W911NF-19-1-0054, and by the DARPA ARCOS program under Contract FA8750-20-C-0507.

REFERENCES

- [1] RTCA, "DO-178C: Software considerations in airborne systems and equipment certification," RTCA and EUROCAE, Washington DC, USA, 2011.
- [2] International Organization for Standardization, "ISO 10218: Robots and robotic devices safety requirements for industrial robots," Geneva, Switzerland: ISO, 2011.
- [3] International Organization for Standardization, "ISO 13482: Robots and robotic devices safety requirements for personal care robots," Geneva, Switzerland: ISO, 2014.
- [4] European Committee for Electrotechnical Standardization, "EN 50126: Railways applications the specification and demonstration of reliability, availability, maintainability and safety (RAMS)," Geneva, Switzerland: ISO, 2017.
- [5] International Organization for Standardization, "ISO 26262: Road vehicles functional safety," Geneva, Switzerland: ISO, 2011.
- [6] International Electro-technical Commission, "IEC 62304: Medical device software—software life cycle processes," Geneva, Switzerland: IEC, 2006.
- [7] M. P. E. Heimdahl, "Safety and software intensive systems: Challenges old and new," in *Proc. Future Softw. Eng.*, 2007, pp. 137–152.
- [8] S. Nair, J. L. De La Vara, M. Sabetzadeh, and L. Briand, "An extended systematic literature review on provision of evidence for safety certification," *Inf. Softw. Technol.*, vol. 56, no. 7, pp. 689–717, 2014.
- [9] International Electro-technical Commission, "IEC 61508: Functional safety of electrical/electronic/programmable safety-related systems," Geneva, Switzerland: IEC, 2010.
- [10] RTCA, "DO-333 : Formal methods supplement to DO-178C and DO-278A," Washington DC, USA, 2011.
- [11] D. D. Cofer and S. P. Miller, "DO-333 certification case studies," in *Proc. 6th Int. Symp. NASA Formal Methods*, 2014, pp. 1–15.
- [12] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, *Counterexample-Guided Abstraction Refinement*. Berlin, Germany: Springer, 2000, pp. 154–169.
- [13] P. Ladkin, "Some practical issues in statistically evaluating critical software," in *Proc. 10th IET Syst. Saf. Cyber-Secur. Conf.*, 2015, pp. 1–5.
- [14] P. B. Ladkin and B. Littlewood, "Practical statistical evaluation of critical software," 2016. [Online]. Available: <http://www.rvs.unibielefeld.de/publications/Papers/LadLitt20150301.pdf>
- [15] R. W. Butler and G. B. Finelli, "The infeasibility of quantifying the reliability of life-critical real-time software," *IEEE Trans. Softw. Eng.*, vol. 19, no. 1, pp. 3–12, Jan. 1993.
- [16] J. Nutaro and O. Ozmen, "Using simulation to quantify the reliability of control software," in *Proc. Winter Simul. Conf.*, 2019, pp. 3267–3276.
- [17] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, Jul. 1976.
- [18] J. Geldenhuys, M. B. Dwyer, and W. Visser, "Probabilistic symbolic execution," in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 166–176, doi: [10.1145/2338965.2336773](https://doi.org/10.1145/2338965.2336773).
- [19] A. Filieri, C. S. Păsăreanu, and W. Visser, "Reliability analysis in symbolic pathfinder," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 622–631.
- [20] A. Filieri, C. S. Păsăreanu, W. Visser, and J. Geldenhuys, "Statistical symbolic execution with informed sampling," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 437–448.
- [21] M. J. Gerrard and M. B. Dwyer, "Comprehensive failure characterization," in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 365–376. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3155562.3155611>
- [22] M. J. Gerrard and M. B. Dwyer, "ALPACA: A large portfolio-based alternating conditional analysis," in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 35–38.
- [23] T. Héault, R. Lassaigne, F. Magniette, and S. Peyronnet, "Approximate probabilistic model checking," in *Proc. Int. Workshop Verification Model Checking Abstract Interpretation*, 2004, pp. 73–84.
- [24] P. Zuliani, A. Platzer, and E. M. Clarke, "Bayesian statistical model checking with application to stateflow/simulink verification," *Formal Methods Syst. Des.*, vol. 43, no. 2, pp. 338–367, Oct. 2013.
- [25] A. Legay, B. Delahaye, and S. Bensalem, "Statistical model checking: An overview," in *Proc. Int. Conf. Runtime Verification*, 2010, pp. 122–135.
- [26] G. Agha and K. Palmskog, "A survey of statistical model checking," *ACM Trans. Model. Comput. Simul.*, vol. 28, no. 1, pp. 6:1–6:39, Jan. 2018.
- [27] G. Ramalingam, "Data flow frequency analysis," *ACM SIGPLAN Notices*, vol. 31, pp. 267–277, 1996.
- [28] D. Monniaux, "Abstract interpretation of probabilistic semantics," in *Proc. Int. Static Anal. Symp.*, 2000, pp. 322–339.
- [29] S. Khurshid, C. S. Păsăreanu, and W. Visser, "Generalized symbolic execution for model checking and testing," in *Proc. 9th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2003, pp. 553–568.
- [30] C. Cadar *et al.*, "KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *Proc. 8th USENIX Conf. Operating Syst. Des. Implementation*, 2008, vol. 8, pp. 209–224.
- [31] S. K. Cha, T. Avgerinos, A. Rebert, and D. Brumley, "Unleashing mayhem on binary code," in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 380–394.
- [32] M. Zheng, M. S. Rogers, Z. Luo, M. B. Dwyer, and S. F. Siegel, "CIVL: Formal verification of parallel programs," in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2015, pp. 830–835.
- [33] A. Barvinok and J. E. Pommersheim, "An algorithmic theory of lattice points," *New Perspectives Algebr. Combinatorics*, vol. 38, 1999, Art. no. 91.
- [34] V. Baldoni *et al.*, "A user's guide for latte integrale v1. 7.2," *Optimization*, vol. 22, no. 2, Oct. 2014.
- [35] L. Granvilliers and F. Benhamou, "Algorithm 852: RealPaver: An interval solver using constraint satisfaction techniques," *ACM Trans. Math. Softw.*, vol. 32, no. 1, pp. 138–156, Mar. 2006, doi: [10.1145/1132973.1132980](https://doi.org/10.1145/1132973.1132980).
- [36] S. Verdoolaege, R. Seghir, K. Beyls, V. Loeschner, and M. Bruynooghe, "Counting integer points in parametric polytopes using barvinok's rational functions," *Algorithmica*, vol. 48, no. 1, pp. 37–66, Mar. 2007, doi: [10.1007/s00453-006-1231-0](https://doi.org/10.1007/s00453-006-1231-0).
- [37] L. Luu, S. Shinde, P. Saxena, and B. Demsky, "A model counter for constraints over unbounded strings," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 565–576, doi: [10.1145/2594291.2594331](https://doi.org/10.1145/2594291.2594331).
- [38] A. Aydin, L. Bang, and T. Bultan, "Automata-based model counting for string constraints," in *Proc. 27th Int. Conf. Comput. Aided Verification*, 2015, pp. 255–272.
- [39] M. Borges, A. Filieri, M. d'Amorim, C. S. Păsăreanu, and W. Visser, "Compositional solution space quantification for probabilistic software analysis," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 123–132, doi: [10.1145/2594291.2594329](https://doi.org/10.1145/2594291.2594329).
- [40] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," in *Proc. 12th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.*, 1985, pp. 291–299. [Online]. Available: <http://doi.acm.org/10.1145/318593.318659>
- [41] M. J. Gerrard, "ALPACA: An instantiation of the alternating conditional analysis framework," 2019, Accessed: Jun. 15, 2019. [Online]. Available: <https://bitbucket.org/mgerrard/alpaca>
- [42] SV-COMP benchmarks, 2018, Accessed: Aug. 1, 2018. [Online]. Available: <https://github.com/sosy-lab/sv-benchmarks>

- [43] T. Reps, "Program analysis via graph reachability," *Inf. Softw. Technol.*, vol. 40, no. 11/12, pp. 701–726, 1998.
- [44] R. Malik, "Lecture notes in model checking," Sep. 2018. [Online]. Available: <https://www.cs.waikato.ac.nz/~robi/comp552-07b/comp552-lecture10.pdf>
- [45] P. Cousot and R. Cousot, "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proc. 4th ACM SIGACT-SIGPLAN Symp. Princ. Program. Lang.*, 1977, pp. 238–252.
- [46] E. M. Clarke Jr., O. Grumberg, and D. A. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
- [47] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler, "Conditional model checking: A technique to pass information between verifiers," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, Art. no. 57.
- [48] W. R. Pestman, *Mathematical Statistics: An Introduction*, vol. 1. Berlin, Germany: Walter de Gruyter, 1998.
- [49] M. Borges, Q.-S. Phan, A. Filieri, and C. S. Păsăreanu, "Model-counting approaches for nonlinear numerical constraints," in *Proc. NASA Formal Methods Symp.*, 2017, pp. 131–138.
- [50] A. Biere and H. van Maaren, *Handbook of Satisfiability*. Amsterdam, Netherlands: IOS Press, 2009.
- [51] A. Groce, G. Holzmann, and R. Joshi, "Randomized differential testing as a prelude to formal verification," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 621–631, doi: 10.1109/ICSE.2007.68.
- [52] P. Godefroid, M. Y. Levin, D. A. Molnar et al., "Automated white-box fuzz testing," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2008, vol. 8, pp. 151–166.
- [53] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," *IEEE Trans. Softw. Eng.*, vol. 45, no. 5, pp. 489–506, May 2019.
- [54] M. Borges, A. Filieri, M. d'Amorim, and C. S. Păsăreanu, "Iterative distribution-aware sampling for probabilistic symbolic execution," in *Proc. 10th Joint Meeting Eur. Softw. Eng. Conf. and the ACM SIGSOFT Symp. Found. Softw. Eng.*, 2015, pp. 866–877.
- [55] L. Valiant, "The complexity of computing the permanent," *Theor. Comput. Sci.*, vol. 8, no. 2, pp. 189–201, 1979.
- [56] B. Büeler, A. Enge, and K. Fukuda, *Exact Volume Computation for Polytopes: A Practical Study*. Basel, Switzerland: Birkhäuser, 2000, pp. 131–154.
- [57] J. Ninin, "Global optimization based on contractor programming: An overview of the ibex library," in *Proc. Int. Conf. Math. Aspects Comput. Inf. Sci.*, 2016, pp. 555–559.
- [58] M. Dyer, A. Frieze, and R. Kannan, "A random polynomial-time algorithm for approximating the volume of convex bodies," *J. ACM*, vol. 38, no. 1, pp. 1–17, Jan. 1991, doi: 10.1145/102782.102783.
- [59] S. Sankaranarayanan, A. Chakarov, and S. Gulwani, "Static analysis for probabilistic programs: Inferring whole program properties from finitely many paths," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2013, pp. 447–458, doi: 10.1145/2491956.2462179.
- [60] C. Robert and G. Casella, *Monte Carlo Statistical Methods*. Berlin, Germany: Springer, 2013.
- [61] M.-T. Trinh, D.-H. Chu, and J. Jaffar, "Model counting for recursively-defined strings," in *Proc. Int. Conf. Comput. Aided Verification*, 2017, pp. 399–418.
- [62] S. Chakraborty, K. S. Meel, and M. Y. Vardi, *A Scalable Approximate Model Counter*. Berlin, Germany: Springer, 2013, pp. 200–216.
- [63] C. P. Gomes, A. Sabharwal, and B. Selman, *Model Counting*. Amsterdam, The Netherlands: IOS Press, 2009, pp. 633–654.
- [64] D. Chistikov, R. Dimitrova, and R. Majumdar, "Approximate counting in SMT and value estimation for probabilistic programs," *Acta Informatica*, vol. 54, no. 8, pp. 729–764, Dec. 2017.
- [65] P. Hennig, M. A. Osborne, and M. Girolami, "Probabilistic numerics and uncertainty in computations," *Proc. Roy. Soc. A: Math. Phys. Eng. Sci.*, vol. 471, no. 2179, 2015, Art. no. 20150142.
- [66] S. Chakraborty, K. S. Meel, and M. Y. Vardi, "Algorithmic improvements in approximate counting for probabilistic inference: From linear to logarithmic sat calls," in *Proc. 25th Int. Joint Conf. Artif. Intell.*, 2016, pp. 3569–3576. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3061053.3061119>
- [67] S. F. Siegel et al., "CIVL: The concurrency intermediate verification language," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2015, pp. 1–12.
- [68] D. Kroening and M. Tautschnig, "CBMC-C bounded model checker," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2014, pp. 389–391.
- [69] P. Andrianov, K. Friedberger, M. Mandrykin, V. Mutilin, and A. Volkov, "CPA-BAM-BnB: Block-abstraction memoization and region-based memory models for predicate abstractions," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2017, pp. 355–359.
- [70] M. Dangl, S. Löwe, and P. Wendler, "Cpachecker with support for recursive programs and floating-point arithmetic," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2015, pp. 423–425.
- [71] J. Morse, M. Ramalho, L. Cordeiro, D. Nicole, and B. Fischer, "ESBMC 1.22," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2014, pp. 405–407.
- [72] C. Richter and H. Wehrheim, "PeSCo: Predicting sequential combinations of verifiers," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2019, pp. 229–233.
- [73] J. Slaby, J. Strejček, and M. Trtík, "Symbiotic: Synergy of instrumentation, slicing, and symbolic execution," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2013, pp. 630–632.
- [74] M. Heizmann et al., "Ultimate automizer and the search for perfect interpolants," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2018, pp. 447–451.
- [75] M. Greitschus et al., "Ultimate taipan: Trace abstraction and abstract interpretation," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2017, pp. 399–403.
- [76] P. Darke et al., "VeriAbs: Verification by abstraction and test generation," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2018, pp. 457–462.
- [77] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proc. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.
- [78] P. Rodgers, G. Stapleton, and P. Chapman, "Visualizing sets with linear diagrams," *ACM Trans. Comput.-Hum. Interaction*, vol. 22, no. 6, pp. 1–39, 2015.
- [79] M. B. Dwyer et al., "CIVL: The concurrency intermediate verification language reference manual, v1.19," Feb. 2019. [Online]. Available: <https://vsl.cis.udel.edu/civl>
- [80] D. Beyer, "Automatic verification of C and Java programs: SV-COMP 2019," in *Proc. Tools Algorithms Construction Anal. Syst.*, 2019, pp. 133–155.
- [81] X. Wang, J. Sun, Z. Chen, P. Zhang, J. Wang, and Y. Lin, "Towards optimal concolic testing," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 291–302.
- [82] H. Palikareva and C. Cadar, "Multi-solver support in symbolic execution," in *Proc. Int. Conf. Comput. Aided Verification*, 2013, pp. 53–68.
- [83] M. B. Dwyer, S. Person, and S. G. Elbaum, "Controlling factors in evaluating path-sensitive error detection techniques," in *Proc. 14th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2006, pp. 92–104, doi: 10.1145/1181775.1181787.
- [84] E. F. Rizzi, S. Elbaum, and M. B. Dwyer, "On the techniques we create, the tools we build, and their misalignments: A study of KLEE," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 132–143.
- [85] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Software verification with blast," in *Proc. Int. SPIN Workshop Model Checking Softw.*, 2003, pp. 235–239.
- [86] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani, "SYNERGY: A new algorithm for property checking," in *Proc. 14th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2006, pp. 117–127.
- [87] N. E. Beckman, A. V. Nori, S. K. Rajamani, and R. J. Simmons, "Proofs from tests," in *Proc. Int. Symp. Softw. Testing Anal.*, 2008, pp. 3–14, doi: 10.1145/1390630.1390634.
- [88] P. Godefroid, A. V. Nori, S. K. Rajamani, and S. D. Tetali, "Compositional may-must program analysis: Unleashing the power of alternation," in *Proc. 37th Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 2010, pp. 43–56. [Online]. Available: <http://doi.acm.org/10.1145/1706299.1706307>
- [89] A. Albargouthi, A. Gurfinkel, and M. Chechik, "From under-approximations to over-approximations and back," in *Proc. 18th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2012, pp. 157–172, doi: 10.1007/978-3-642-28756-5_12.
- [90] A. Albargouthi, Y. Li, A. Gurfinkel, and M. Chechik, "Ufo: A framework for abstraction-and interpolation-based software verification," in *Proc. Int. Conf. Comput. Aided Verification*, 2012, pp. 672–678.
- [91] F. Nielson, H. R. Nielson, and C. Hankin, *Principles of Program Analysis*. Berlin, Germany: Springer, 2005.
- [92] D. Beyer and P. Wendler, "Reuse of verification results," in *Proc. Int. SPIN Workshop Model Checking Softw.*, 2013, pp. 1–17.
- [93] M. Czech, M.-C. Jakobs, and H. Wehrheim, "Just test what you cannot verify!" in *Proc. Int. Conf. Fundam. Approaches Softw. Eng.*, 2015, pp. 100–114.

- [94] M. Christakis, P. Müller, and V. Wüstholtz, "Guiding dynamic symbolic execution toward unverified program executions," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 144–155.
- [95] K. Ferles, V. Wüstholtz, M. Christakis, and I. Dillig, "Failure-directed program trimming," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 174–185.
- [96] D. Beyer, M.-C. Jakobs, T. Lemberger, and H. Wehrheim, "Reducer-based construction of conditional verifiers," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 1182–1193.
- [97] J.-P. Katoen, "The probabilistic model checking landscape," in *Proc. 31st Annu. ACM/IEEE Symp. Logic Comput. Sci.*, 2016, pp. 31–45. [Online]. Available: <http://doi.acm.org/10.1145/2933575.2934574>
- [98] D. Kozen, "Semantics of probabilistic programs," *J. Comput. Syst. Sci.*, vol. 22, no. 3, pp. 328–350, 1981.
- [99] P. Cousot and M. Monerau, "Probabilistic abstract interpretation," in *Proc. Eur. Symp. Program.*, 2012, pp. 169–193.
- [100] K. Luckow, C. S. Păsăreanu, M. B. Dwyer, A. Filieri, and W. Visser, "Exact and approximate probabilistic symbolic execution for non-deterministic programs," in *Proc. 29th ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2014, pp. 575–586.
- [101] B. Chen, Y. Liu, and W. Le, "Generating performance distributions via probabilistic symbolic execution," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 49–60. [Online]. Available: <http://doi.acm.org/10.1145/2884781.2884794>
- [102] T. Brennan, S. Saha, T. Bultan, and C. S. Păsăreanu, "Symbolic path cost analysis for side-channel detection," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 27–37. [Online]. Available: <http://doi.acm.org/10.1145/3213846.3213867>
- [103] P. Malacaria, M. Khouzani, C. S. Pasareanu, Q. Phan, and K. Luckow, "Symbolic side-channel analysis for probabilistic programs," in *Proc. IEEE 31st Comput. Secur. Found. Symp.*, 2018, pp. 313–327.
- [104] A. D. Gordon, T. A. Henzinger, A. V. Nori, and S. K. Rajamani, "Probabilistic programming," in *Proc. Future Softw. Eng.*, 2014, pp. 167–181. [Online]. Available: <http://doi.acm.org/10.1145/2593882.2593900>
- [105] A. Sampson, P. Panchekha, T. Mytkowicz, K. S. McKinley, D. Grossman, and L. Ceze, "Expressing and verifying probabilistic assertions," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 112–122, 2014.
- [106] J. Bornholt, T. Mytkowicz, and K. S. McKinley, "Uncertain: A first-order type for uncertain data," in *Proc. 19th Int. Conf. Architectural Support Program. Lang. Operating Syst.*, 2014, pp. 51–66. [Online]. Available: <http://doi.acm.org/10.1145/2541940.2541958>
- [107] M. Böhme and S. Paul, "On the efficiency of automated testing," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 632–642.
- [108] A. Mayr, R. Plösch, and M. Saft, "Towards an operational safety standard for software: Modelling IEC 61508 Part 3," in *Proc. 18th IEEE Int. Conf. Workshops Eng. Comput.-Based Syst.*, 2011, pp. 97–104.



Mitchell Gerrard is currently working toward the PhD degree at the University of Virginia, Charlottesville, Virginia. His research interests include software analysis and verification, particularly in facilitating conditional verification between a diversity of analysis tools. His nonresearch interests include literate programming, functional languages, and Shandeism.



Mateus Borges is currently working toward the PhD degree at the Department of Computing, Imperial College London, United Kingdom. His research interests include software engineering and program analysis methods to improve the accuracy, scalability and effectiveness of testing, debugging, and quantitative verification techniques.



Matthew B. Dwyer (Fellow, IEEE) is a professor and the John C. Knight faculty fellow with the Department of Computer Science, University of Virginia, Charlottesville, Virginia. His research interests include software analysis, verification and testing and his work in these areas has been recognized over the years with several test-of-time and distinguished paper awards. He is a fellow of the ACM.



Antonio Filieri is an assistant professor at Imperial College London, United Kingdom. His main research interests are in the application of mathematical methods for Software Engineering, in particular, Probability, Statistics, Logic, and Control theory. His recent work includes exact and approximate methods for probabilistic symbolic execution, incremental verification, quantitative software modeling and verification at runtime, and control-theoretical software adaptation. For more information, please visit <https://antonio.filieri.name>.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csl.

DPWord2Vec: Better Representation of Design Patterns in Semantics

Dong Liu, He Jiang^{ID}, Member, IEEE, Xiaochen Li^{ID}, Zhilei Ren,
Lei Qiao^{ID}, and Zuohua Ding^{ID}, Member, IEEE

Abstract—With the plain text descriptions of design patterns, developers could better learn and understand the definitions and usage scenarios of design patterns. To facilitate the automatic usage of these descriptions, e.g., recommending design patterns by free-text queries, design patterns and natural languages should be adequately associated. Existing studies usually use texts in design pattern books as the representations of design patterns to calculate similarities with the queries. However, this way is problematic. Lots of information of design patterns may be absent from design pattern books and many words would be out of vocabulary due to the content limitation of these books. To overcome these issues, a more comprehensive method should be constructed to estimate the relatedness between design patterns and natural language words. Motivated by Word2Vec, in this study, we propose DPWord2Vec that embeds design patterns and natural language words into vectors simultaneously. We first build a corpus containing more than 400 thousand documents extracted from design pattern books, Wikipedia, and Stack Overflow. Next, we redefine the concept of context window to associate design patterns with words. Then, the design pattern and word vector representations are learnt by leveraging an advanced word embedding method. The learnt design pattern and word vectors can be universally used in textual description based design pattern tasks. An evaluation shows that DPWord2Vec outperforms the baseline algorithms by 24.2–120.9 percent in measuring the similarities between design patterns and words in terms of Spearman’s rank correlation coefficient. Moreover, we adopt DPWord2Vec on two typical design pattern tasks. In the design pattern tag recommendation task, the DPWord2Vec-based method outperforms two state-of-the-art algorithms by 6.6 and 32.7 percent respectively when considering *Recall@10*. In the design pattern selection task, DPWord2Vec improves the existing methods by 6.5–70.7 percent in terms of MRR.

Index Terms—Design pattern, word embedding, Word2Vec, semantic similarity, tag recommendation, design pattern selection

1 INTRODUCTION

SOFTWARE design patterns derive from the notion of design pattern in the area of architecture [1], aiming to document reusable experience for recurring software design problems [2]. In recent years, many studies about design patterns have been conducted [3], [4], [5]. As to the literature, there are roughly two ways to describe design patterns: the formal way and the informal way.

The formal way specifies design patterns with formally defined pattern languages. For example, the Gang-of-Four (GOF) book respectively uses Unified Modeling Language (UML) class diagram and sequence diagram to illustrate the

structure and collaborations of each design pattern [2]. A number of studies are based on the formal descriptions of design patterns [4], [6], as formal specifications enhance the capabilities of machine processing [7]. However, there are some weaknesses of the formal way. First, it is inconvenient to precisely specify the intent and applicability of design patterns. Second, building the meta-model of each design pattern is usually costly [8]. Third, the formal way may lose human readability, which is critically important to the utility of design patterns [7].

Conversely, the informal way depicts design patterns with free text. Comparing with the formal way, it is more understandable and convenient to describe design pattern relevant artifacts in words. Thus, the informal way is a profitable supplement to the formal way. To provide tool supports for design pattern relevant tasks based on informal descriptions, the key point is to establish the semantic relationships between design patterns and natural languages, so that the retrieval or identification of design patterns can be practically realized. However, to associate design patterns with natural languages is no easy job. A design pattern name is usually a phrase, such as “factory method”. An experienced developer may capture the semantics of the design pattern via the name as he/she understands the relevant background. But for the automatic tools, it is difficult to comprehend the connotations from only these several words. More information about design patterns should be provided for them to “learn” the background knowledge.

- Dong Liu is with the School of Software, Dalian University of Technology, Dalian 116620, China. E-mail: dongliu@mail.dlut.edu.cn.
- He Jiang and Zhilei Ren are with the Key Laboratory for Ubiquitous Network and Service Software of Liaoning Province, School of Software, Dalian University of Technology, Dalian 116620, China. E-mail: {fjianghe, zren}@dlut.edu.cn.
- Xiaochen Li is with the School of Software, Dalian University of Technology, Dalian 116620, China, and also with the Software Verification and Validation Lab, University of Luxembourg, L-1855 Luxembourg, Luxembourg. E-mail: xiaochen.li@uni.lu.
- Lei Qiao is with the Beijing Institute of Control Engineering, Beijing 100190, China. E-mail: fly2moon@aliyun.com.
- Zuohua Ding is with the School of Information Sciences, Zhejiang Sci-Tech University, Hangzhou 310018, China. E-mail: zouhuad@hotmail.com.

Manuscript received 28 Nov. 2019; revised 17 July 2020; accepted 29 July 2020. Date of publication 18 Aug. 2020; date of current version 18 Apr. 2022. (Corresponding author: He Jiang.) Recommended for acceptance by N. Bencomo. Digital Object Identifier no. 10.1109/TSE.2020.3017336

To obtain exact semantic information of design patterns, the existing studies usually take the descriptions in design pattern books as standard definitions of design patterns [8], [9], [10]. If a snippet of text is similar to the standard definition of a design pattern, then it is likely to be related to the design pattern. Hence, the relatedness between design patterns and natural languages can be estimated. However, this kind of methods is still problematic. On one hand, much information about design patterns is absent from these books. Design pattern books usually depict the mechanisms, scenarios, and specifications of design patterns [2]. As time goes by, many applications beyond the original design pattern books have been developed. For example, the Active Record design pattern is related to the Ruby on Rails framework as Active Record provides the data model of the framework.¹ The AngularJS framework implements the Dependency Injection design pattern itself and usually accompanies by this design pattern.² These relationships cannot be mined from design pattern books. On the other hand, the vocabulary extracted from design pattern books is usually too small. The lengths of descriptions in design pattern books are limited and many natural language words may be out of the scope. It is difficult to handle the texts containing many out-of-vocabulary words. Therefore, the wide usage of this kind of methods is restricted.

In this study, we aim to overcome these issues by constructing a general method to estimate the relatedness between design patterns and natural language words, in order that it can be universally used in the tasks based on informal descriptions of design patterns. The “words” here refer to as both plain natural language words, such as “factory” and “method”, and software specific terms, such as “angularjs”. Inspired by the word embedding method [11], we propose DPWord2Vec that maps both design patterns and natural language words into one vector space. With the design pattern and word vectors, the similarity between a design pattern and a word or a document can be calculated. In this way, the relationship between natural languages and design patterns can be built. However, there are two challenges to be addressed. First, how to find a relatively large corpus about design patterns? Second, how to associate a design pattern with its relevant natural language words for vectors training?

To handle the first challenge, we build a general corpus containing 491,555 documents. The general corpus consists of two parts: the description corpus and the crowdsourced corpus. The description corpus contains relatively formal design pattern descriptions that are extracted from design pattern books and Wikipedia. The crowdsourced corpus is constructed based on a set of design pattern relevant Stack Overflow posts obtained from our previous work [12]. Then we extend the concept of context window in Word2Vec to our general corpus and define the context windows for each design pattern and each word respectively. In this way, the linkages between design patterns and words are established, that is, the design pattern context windows contain words and design patterns appear in word context windows. Hence, the second challenge can be properly addressed. Finally, the

design pattern and word vector representations are learnt by leveraging an advanced word embedding method, namely GloVe [13], based on these context windows.

To clarify the quality of the learnt design pattern and word vectors, we deploy an evaluation with a dp-word (design pattern - word) similarity task. Experimental results on 2,000 manually labelled dp-word pairs show that the learnt vectors by DPWord2Vec are more effective than some widely used semantic relatedness estimation algorithms, i.e., outperform these algorithms by 24.2-120.9 percent in terms of Spearman’s rank correlation coefficient. To show the practicability, we depict two applications of DPWord2Vec to solve two typical design pattern tasks, i.e., design pattern tag recommendation and design pattern selection. In the first application, when recommending the top 10 design pattern tags for the posts in a software information site, the DPWord2Vec-based method outperforms two state-of-the-art tag recommendation methods by 6.6 and 32.7 percent respectively in terms of *Recall@10*. In the second application, the method refined by DPWord2Vec outperforms the two existing design pattern selection methods by 6.5 and 70.7 percent respectively when considering the mean values of Mean Reciprocal Rank (MRR) over three design pattern collections.

In this paper, we make the following contributions:

- 1) We propose DPWord2Vec that maps both design patterns and natural language words into vectors to support design pattern relevant tasks. To the best of our knowledge, this is the first work that establishes the universal relationship between design patterns and natural languages.
- 2) We evaluate DPWord2Vec on a manually labelled dp-word pair dataset to show its effectiveness in semantic relatedness estimation.
- 3) DPWord2Vec is applied to two design pattern relevant tasks, namely design pattern tag recommendation and design pattern selection. DPWord2Vec outperforms the state-of-the-art methods.

The rest of this paper is organized as follows. Section 2 shows the background of the study. Section 3 presents the framework of DPWord2Vec. The settings and results for evaluating DPWord2Vec are depicted in Sections 4 and 5, respectively. Sections 6 and 7 introduce two applications of DPWord2Vec. Section 8 discusses potential threats to validity. Some studies related to our work are outlined in Section 9. We conclude the paper in Section 10.

2 PRELIMINARIES

Before the depiction of DPWord2Vec, we demonstrate the concept of design pattern in this study and briefly introduce the word embedding technique.

2.1 Concept of Design Pattern

Generally speaking, design patterns are proven solutions to recurring software design problems [2]. However, to the best of our knowledge, there are no formal definitions nor standard lists of design patterns. There exist numbers of design pattern collections that are published with multiple channels, such as design pattern books, academic papers, or

1. https://guides.rubyonrails.org/active_record_basics.html
 2. <https://angular.io/guide/dependency-injection>

online libraries [7]. Design patterns in different collections may be depicted in different ways, e.g., in flat text format or using UML. In this paper, we focus on the design patterns with rich textual descriptions and collect design patterns from various sources.

Similar to “design pattern”, “architecture pattern” is also a means for software design. Strictly, they are not a same concept, but the boundary between them may not be unified for different design pattern collections. For example, Model View Controller is an example of architectural pattern in Wikipedia³ but marked as a design pattern in MSDN.⁴ Therefore, instead of creating a standard subjectively, we choose not to distinguish them in our study. Once an entity is identified as a design pattern in some design pattern collections, we regard it as a design pattern.

2.2 Word Embedding

Word embedding is a set of techniques that maps words or phrases in the vocabulary to vectors of real numbers. The core part of DPWord2Vec is also word embedding, but it handles both words and design patterns. Word embedding methods focus on mapping words into a continuous vector space with a much lower dimension than the size of vocabulary and the vector representation of each word is determined by supervised learning based on the corpus [11].

To facilitate the demonstration, we explain how word embedding works with an example. Assuming there is a corpus that contains a sentence: “software design patterns encapsulate proven solutions that address recurring problems”. To mine the relationships between words, the sliding context window strategy is usually used [11]. A context window contains a central word and several surrounding words which are at a distance of no more than c positions from the central word. For example, the context window with centre “patterns” and $c = 2$ contains the surrounding words “software”, “design”, “encapsulate”, and “proven”. Multiple local context windows are constructed as the central word slides from the beginning (“software”) to the end (“problems”) of the corpus.

Then the word vectors are learnt based on these local context windows. The intuition is that if two words appear frequently in the same context window then their vector representations are highly associated. For example, the objective of the Skip-gram model is to learn word vector representations that are good at predicting each surrounding word by the vector of the central word [11]. Conversely, the Continuous Bag-of-Words (CBOW) model aims to predict the central word by the concatenation or average of the vectors of the surrounding words [11]. Different from them, the GloVe model counts the number of the total co-occurrences of each pair of words through all the local context windows and predicts the co-occurrence number by the vectors of the words in the pair [13].

3 THE DPWORD2VEC FRAMEWORK

DPWord2Vec aims to embed natural language words and design patterns into one vector space. This process can be

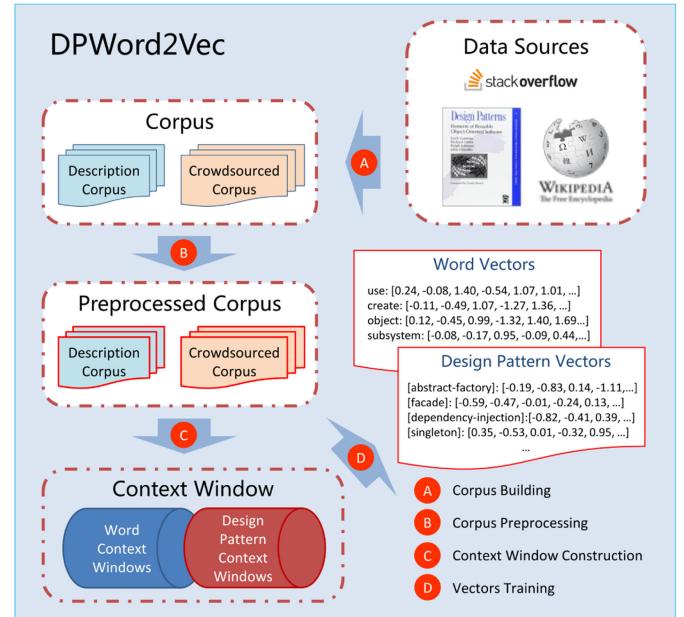


Fig. 1. The framework of DPWord2Vec.

divided into four phases (as shown in Fig. 1). At first, the corpus related to design patterns are acquired from multiple sources. Next, the documents in the corpus are preprocessed. Then, we propose a context window-based strategy to strengthen the tie between words and design patterns. At last, the word and design pattern vectors are trained based on the corpus and the context windows.

3.1 Corpus Building

To train the vectors of words and design patterns, a corpus relevant to design patterns should be built at first. Formally, we construct a general corpus C , which contains multiple documents. For each document doc in C , doc has two components: the token component $doc.Tokens$, a sequence of natural language words that describes some design patterns, and the design pattern component $doc.DPs$, a set of design patterns described by $doc.Tokens$. The general corpus C can be further categorized into two groups according to their sources.

Description Corpus. Documents in this corpus are extracted from design pattern books and Wikipedia. Some design pattern books catalog their own lists of design patterns. For example, GoF presents 23 design patterns with the problem definitions and design specifications [2]. A design pattern is usually described by a chapter or a section in a design pattern book. Similarly, a number of design patterns are specified by Wikipedia as entries with one page for each design pattern.⁵ A chapter or section of a design pattern book, or a Wikipedia page of a design pattern forms a document doc . In this corpus, $doc.Tokens$ denotes the whole text in the chapter, section, or page, but excluding the code snippets. Meanwhile, $doc.DPs$ contains only one element, i.e., the described design pattern.

Totally, the description corpus contains 431 documents, which are associated with 13 design pattern books and 125 Wikipedia pages. Amongst the design pattern components, 349 unique design patterns are involved.

3. https://en.wikipedia.org/wiki/Architectural_pattern

4. <https://msdn.microsoft.com/en-us/library/ms978748.aspx>

5. https://en.wikipedia.org/wiki/Category:Software_design_patterns

Crowdsourced Corpus. Documents in this corpus are constructed by referring to the programming forum, i.e., Stack Overflow.⁶ In the previous study [12], 187,493 design pattern relevant question posts spanning from August 2008 to December 2017 are detected in Stack Overflow.

A design pattern relevant post indicates the design pattern name(s) appears at least one time in the post. However, it is not a trivial string matching task to detect the design pattern occurrences in Stack Overflow posts, as the discussions on Stack Overflow are usually informal [14], [15] and the name of a design pattern may not be mentioned in a unique form. It is also referred to as the morphological form issue [14]. The previous study has attempted to address this issue in two aspects. On the one hand, the standard design pattern names as well as other common names are collected simultaneously from the existing design pattern collections, e.g., design pattern books, in which the other well-known names of each containing design pattern are usually presented explicitly, e.g., marked as “also known as”. These names include aliases, e.g., “open implementation” is an alias for “reflection”, and acronyms, e.g., “mvc” is an acronym for “model view controller”. On the other hand, regular expressions are leveraged to allow some variants when searching a design pattern name in the text of the Stack Overflow posts. For example, the regular expression for “model view controller” is “model[à-z]?view [à-z]?controller”, where “[à-z]?” denotes a non-alphabetic character that matches zero or one time, so that the variants such as “model-view-controller”, “model_view_controller”, and “modelviewcontroller” can be involved. A manual validation on the sampled posts shows that the detection is acceptably accurate, i.e., achieves Precision value of 97.3 percent and Recall value of 87.8 percent. More details can be obtained by referring to [12].

We use these question posts to construct the crowdsourced corpus. Moreover, it is enriched by all the answer posts to these design pattern relevant question posts. A question post and each of its answer post are assigned to different documents. The relevant design pattern(s) to an answer post is as same as its question post. For a document doc in this corpus, $doc.Tokens$ denotes a content merging the title and body part of a question or answer post with code snippets discarded, and $doc.DPs$ is the set of the relevant design pattern(s) to the post.

Finally, there are 491,124 documents in this corpus and 210 unique design patterns are involved.

By merging the two corpora, we obtain a general corpus C , which contains 491,555 documents.⁷ The involved design patterns are indexed and form a design pattern vocabulary, namely V_{DP} , with 372 design patterns. Although the documents in the description corpus are far less than those in the crowdsourced corpus, the description corpus is indispensable for building the design pattern vectors. On one hand, the description corpus makes it possible to build vectors for the design patterns that are rarely discussed in Stack Overflow. On the other hand, this corpus tends to provide more formal and precise depictions of design patterns than the

crowdsourced corpus. We will show its significance in Section 5.1.

3.2 Corpus Preprocessing

Comparing to the general natural language documents, the amount of design pattern relevant documents tends to be quite small. Therefore, our built corpus is relatively smaller than those for training the common word vectors [11], [13]. Based on this actuality, we perform preprocessing on the token component of each document aiming to filter out the insignificant and redundant information and build a compact vocabulary.

At first, code-like tokens (e.g., function names) in a natural language sentence are split according to its camel style to ensure the semantic integrity of the sentence. With this step, on the one hand, these code-like tokens can be converted into more understandable identifiers [16] to better reflect the semantic meanings. On the other hand, the volume of the vocabulary can be reduced. Next, we tokenize and lowercase the token component of each document. Then, the less informative tokens, including English-language stop words, special tags (HTML tags in Stack Overflow posts, and reference markers in design pattern books and Wikipedia pages), and non-alphabetic characters (e.g., numbers) are removed from the text, as they are not very useful to reflect the semantic relationship between the natural language and design patterns. Moreover, each token is stemmed to its root form, e.g., “developer”, “developed”, and “developing” to “develop”. As the words with a same root usually have similar meaning [17] and the vector representations of them are also similar in some word embedding methods [18], [19], we can simply regard them as a same word without losing much semantic information. At last, we discard the words that occur no more than five times in the corpus when constructing the vocabulary but retain them in the corpus. These words are likely to be noisy terms [20] and it is not significant to train the vectors of them.

Some of the above steps, such as camel case splitting, stop words removing, and word stemming, may be not common in word embedding methods. With abundant training corpora, vector representation of each distinct identifier in the text can be learnt. However, due to the scale of the design pattern corpus, it is reasonable to conduct these preprocessing steps to reduce the vocabulary size, i.e., the number of vectors to be learnt, to adapt to the corpus. Furthermore, the focus of this study is to build the semantic relationship between natural languages and design patterns, it is not a core concern to represent all the identifiers precisely. As a common concept in the word embedding methods, the word context will not be significantly affected by the preprocessing, since the eliminated tokens contain little semantic information and the meanings of the changed tokens are mainly retained. It is adequate to apply the word embedding methods to the preprocessed corpus.

After the preprocessing, we obtain a word vocabulary V_{Word} that contains 27,770 words.

3.3 Context Window Construction

As to the corpus we build, each document contains two parts: the natural language words and the design patterns. To train the vectors of words and design patterns together,

6. <https://stackoverflow.com/>

7. The detailed description corpus and crowdsourced corpus, as well as the number of relevant documents to each design pattern are available via <https://github.com/WoodenHeadoo/dpword2vec>.

Dependency Injection is a practice where objects are designed in a manner where they receive instances of the objects from other pieces of code, instead of constructing them internally. This means that any object implementing the interface which is required by the object can be substituted in without changing the code, which simplifies testing, and improves decoupling.

Fig. 2. A paragraph that describes the Dependency Injection design pattern. The design pattern name is in red bold font and the words in the context window (of size five) of the name are in blue italic font.

we should combine the two parts. In standard word embedding models, words are usually associated by leveraging the sliding context window-based strategy [11]. For example, in the Skip-gram model, the vector representation of the central word is learned for predicting the other words in a context window. Similarly, the CBOW model uses the composition of the vectors of the surrounding words in a context window to predict the central word. Hence, a reasonable method for associating natural language words and design patterns is to locate them in a context window.

To this end, an intuitive way is to regard design pattern names appearing in natural language text as special “words”. Concretely, given a document doc in the corpus C , for the design patterns in $doc.DPs$, we detect all the occurrences of design pattern names (including aliases) in $doc.Tokens$ and replace them with predefined tokens. These predefined tokens are the “words” of design patterns and mixed with the natural language words. Then design patterns can be handled together with natural language words by the sliding context window-based strategies. However, there is a main issue for this way: design pattern names tend to appear infrequently in the text. For instance, Fig. 2 presents a paragraph in a post (#131766) of Stack Overflow. This paragraph indeed describes the Dependency Injection design pattern, but the design pattern name only appears one time at the beginning of the paragraph. When applying the sliding context window-based strategies to this paragraph, the design pattern Dependency Injection can be only associated with some words in the front but the rest are ignored.

To resolve this issue, we redefine the concept of context window by considering both natural language words and design patterns. In the new definition, the context window size is not fixed, but there is also a parameter of context window size for words as the standard models. For clarity, we name it as c .

There are two types of context windows:

Context Window for Word. For a word in a document, the context window for this word contains other words around

the word with radius c and all the design patterns the document describes. Formally, for a document doc in C , let $doc.Tokens(i)$ denote the i th word of the text and $doc.Tokens.len$ denote the length of the text. The Context Window of $doc.Tokens(i)$ is defined as

$$\begin{aligned} & \text{Context}_{\text{doc}}^{\text{Word}}(i, doc.Tokens(i)) \\ &= \{doc.Tokens(j) | \max\{1, i - c\} \leq j \leq \\ & \quad \min\{doc.Tokens.len, i + c\}, j \neq i\} \cup doc.DPs. \end{aligned} \quad (1)$$

Take the document in Table 1 as an example. Assuming $c = 2$, the Context Window for the sixth word “interface” contains the two words ahead of it (i.e., “facade” and “provide”), the two words behind it (i.e., “create” and “subsystem”), as well as the two design patterns mentioned in the document (i.e., “[abstract-factory]” and “[facade]”).

Context Window for Design Pattern. Given a design pattern described by a document, the context window for the design pattern consists of all the words in the text and the other described design patterns. Formally, for a document doc and a design pattern $dp \in doc.DPs$, the Context Window of dp is

$$\begin{aligned} & \text{Context}_{\text{doc}}^{DP}(dp) \\ &= \{doc.Tokens(j) | 1 \leq j \leq doc.Tokens.len\} \\ & \quad \cup (doc.DPs - \{dp\}). \end{aligned} \quad (2)$$

For example, in Table 1, the Context Window for the design pattern “[abstract-factory]” contains all the words (i.e., “abstract”, “factory”, ..., “class”) and the other design pattern “[facade]”.

According to the definitions of the two context windows, a design pattern can be associated with each word in the document that describes the design pattern. The tie between words and design patterns is strengthened. To show the effectiveness of the new definitions, we use the performance of the method that leverages design pattern name occurrences (mentioned above) for comparison in Section 5.3.

With the definitions, for any document doc in C , the context window of each word in $doc.Tokens$ and the context window of each design pattern in $doc.DPs$ are constructed.

3.4 Vectors Training

Once the context windows are clarified, the word and design pattern vectors can be generated by any sliding context window-based models. In DPWord2Vec, we choose GloVe [13] for vector generation, due to the following reasons:

TABLE 1
An Example for Two Types of Context Windows ($c = 2$)

$doc.Tokens^a$	$doc.DPs$
¹ Abstract ² factory can be ³ used(use) ⁴ facade to ⁵ provide an ⁶ interface for ⁷ creating(create) ⁸ subsystem ⁹ objects(object) in a ¹⁰ subsystem ¹¹ independent ¹² way. ¹³ Abstract ¹⁴ factory can ¹⁵ also be ¹⁶ used(use) as an ¹⁷ alternative to ¹⁸ facade to ¹⁹ hide ²⁰ platform ²¹ specific ²² classes(class).	[abstract-factory], [facade]
$\text{Context}_{\text{doc}}^{\text{Word}}(6, \text{"interface"}) = \{\text{"facade"}, \text{"provide"}, \text{"create"}, \text{"subsystem"}, \text{"[abstract-factory]"}, \text{"[facade]"}\}$ $\text{Context}_{\text{doc}}^{DP}(\text{"[abstract-factory]"}) = \{\text{"abstract"}, \text{"factory"}, \dots, \text{"class"}, \text{"[facade]"}\}$	

^a As declared above, the stop words are eliminated from the text of the document (in strikeout fonts) and the rest of the words are stemmed to their root forms.

- 1) GloVe is a state-of-the-art model that outperforms Skip-gram and CBOW on several natural language processing tasks with higher efficiency [13].
- 2) GloVe benefits from both global co-occurrences and local context windows. Global co-occurrences suit to present the dp-word relationships and design pattern - design pattern relationships. Meanwhile, the word - word relationships could be well handled by local context windows. Therefore, the GloVe model is suitable for this scenario.

To train the vectors with GloVe, the input of GloVe should be specified. Generally, the input of GloVe is the entries co-occurrence counts matrix X , whose element X_{ij} represents the number of times entry j occurs in the context window of entry i . In DPWord2Vec, entry j and entry i can be any word in the word vocabulary V_{Word} or any design pattern in the design pattern vocabulary V_{DP} . Therefore, in DPWord2Vec, X_{ij} is calculated respectively when entry i is a word and when entry i is a design pattern according to the two definitions of context window. Note that $X_{ij} = X_{ji}$ for any j and i according to our context window definitions, hence only half of the entries co-occurrence counts should be calculated.

Moreover, the dp-word co-occurrences are weighted. According to [21], the frequencies of words follow Zipf's law in natural language corpora. Similarly, the number of relevant posts in Stack Overflow to each design pattern exhibits a long tail behavior [12]. That means, the distribution of words or design patterns is highly skewed. Moreover, according to the definitions, the context window of a design pattern contains all the words in the document and the design pattern is also contained in the context window of each of the words. As a result, some design patterns may appear commonly in the context windows of many words, i.e., potentially relate to many words, and vice versa. When dealing with the tasks which request to associate design patterns with words, e.g., to retrieve design patterns by keywords, we should ensure these very common design patterns not to be over weighted. Likewise, the words that are contained in the context windows of many design patterns should also be well handled. Hence, a weighting strategy is applied to diminish the effects of these common terms. Formally, if entry j is a word and entry i is a design pattern, X_{ij} is tuned by the weights of j and i . The weights are calculated just like the inverse document frequency value

$$w_j = \log\left(\frac{\#V_{DP}}{Occur_{DP}(j)}\right), w_i = \log\left(\frac{\#V_{Word}}{Occur_{Word}(i)}\right), \quad (3)$$

where $Occur_{DP}(j)$ denotes the number of unique design patterns in V_{DP} that ever occur in the context window of word j and $Occur_{Word}(i)$ denotes the number of unique words in V_{Word} that ever occur in the context window of design pattern i . The weights are normalized by the average values

$$\widetilde{w}_j = \frac{w_j}{\text{avg}\{w_{j'} | j' \in V_{Word}\}}, \widetilde{w}_i = \frac{w_i}{\text{avg}\{w_{i'} | i' \in V_{DP}\}}. \quad (4)$$

Finally, X_{ij} is recalculated as

$$\widetilde{X}_{ij} = \text{ceil}(X_{ij} \cdot \widetilde{w}_i \cdot \widetilde{w}_j), \quad (5)$$

where $\text{ceil}(\cdot)$ is a function that converts a floating number to the nearest integer.

Given the vector dimension, the vectors of words in V_{Word} and design patterns in V_{DP} are generated by GloVe⁸ based on the entries co-occurrence counts matrix X . For training GloVe, we use the settings in [13], i.e., the number of iterations is 100, the initial learning rate is 0.05, and the model parameters $x_{max} = 100$ and $\alpha = 0.75$. Finally, the word and design pattern vectors are calculated as the sum of the "input" and "output" vectors generated by GloVe.⁹

4 EVALUATION SETTINGS

In this section, we present the experimental settings for evaluating the DPWord2Vec model, including evaluation protocols, baseline algorithms, evaluation metrics, and parameter settings of DPWord2Vec.

4.1 Evaluation Protocols

In this subsection, we demonstrate the strategy and dataset for evaluating DPWord2Vec.

Word similarity tasks are usually leveraged to evaluate the quality of word vectors in word embedding models [13], [18], [22], [23]. Generally speaking, two semantically relevant words should indicate that their vector representations are similar [22]. In DPWord2Vec, "word" means natural language word or design pattern. As we focus on the relationship between natural languages and design patterns, only the dp-word similarity is considered. This similarity can be estimated by calculating the cosine similarity of the word vector and the design pattern vector. To the best of our knowledge, there exist no publicly available datasets for dp-word similarity evaluation. Therefore, we build a new dataset of dp-word pairs with relatedness labels to address this issue.¹⁰

Design Pattern Selection. At first, a list of design patterns is selected. To obtain a diverse list of design patterns, we select design patterns based on their frequencies, like the methods for word similarity datasets construction [18], [23]. The frequency of a design pattern means the number of documents in C that describe the design pattern. The 372 design patterns in V_{DP} can be grouped into five classes according to five frequency intervals: (0,10], (10,50], (50,400], (400,1500], and (1500, +∞). Except the first class which contains a relatively large number of infrequent design patterns, the other four classes have similar sizes, i.e., there are 34, 33, 33, and 34 design patterns in these classes respectively. We randomly sample ten design patterns from each class and get a list of 50 design patterns.

Pair Construction. Next, for each design pattern, we select a list of words to form pairs. Given a design pattern, if a word is randomly selected from V_{Word} , it is unlikely to be related to the design pattern. In other studies, word pairs are constructed by using WordNet synonym sets [18], [23]. However, there are no similar databases specified for design patterns as to our knowledge. Hence, we employ the frequency

8. <https://nlp.stanford.edu/projects/glove/>

9. The source code and the learnt word and design pattern vectors can be accessed on <https://github.com/WoodenHeadoo/dpword2vec>.

10. We provide the dataset on <https://github.com/WoodenHeadoo/dpword2vec>.

of co-occurrence to select words. The intuition is if a design pattern and a word appear in the same document frequently, they are more likely to be relevant, then the word is more likely to be chosen. Concretely, given a design pattern, 40 non-duplicated words are randomly chosen based on a distribution, in which the probability of choosing a word is proportional to the number of documents containing both the word and the design pattern. Then we obtain $50 \times 40 = 2,000$ dp-word pairs and the number is comparable to those in [18] and [23].

Human Judgment. According to the last step of word similarity datasets construction [18], [23], [24], the relatedness between the design pattern and the word in each pair is manually labelled. To reduce the influence of personal biases, we recruit three graduate students to label the pairs. These participants all have bachelor's degrees majoring in computer science or software engineering and have been trained in object-oriented programming including design pattern relevant skills. They are also experienced with annotating software artifacts, such as evaluating the quality of the enriched API specifications and scoring the results of the code search algorithms.

Before labelling these pairs, all the participants go over the materials of the involving design patterns as a retrospect. When labelling, each dp-word pair is sent to each participant and he/she attempts to construct a context that the word is mentioned and associated with the design pattern. In this procedure, the participants are allowed to search for the texts that contain the design pattern and the word on the Internet to help them. If one still doubts whether such a context exists, the documents in C , in which the design pattern and the word co-occur, can serve as references. For each participant, a pair is labelled as "related" if the design pattern and the word can be associated in some certain contexts, and labelled as "unrelated" if they are hard to be linked or the meaning of the word is so general that the link seems to be too weak. The final label of a pair is "related" or "unrelated" if the participants can reach an agreement, i.e., they all label it as "related" or "unrelated". Otherwise, its final label is "somewhat related". That means, there exists some uncertainty but the relatedness is between "related" and "unrelated".

From the labelling process, we get some observations. Some pairs are consistently labelled as "related" since the word can describe the use scenario of the design pattern directly and the relationship between them can be easily imagined. For example, Publish/Subscribe is a messaging design pattern that provides instant notifications for distributed applications. The related words include "event" (the notifications are events), "channel" (notifications are broadcasted via the channel), and "endpoint" (the notification publishers and subscribers are all endpoints). Some pairs are related when considering the background of the entity that the word represents. For example, the word "wpf" refers to a programming framework. It is supposed to be related to the Model View ViewModel (MVVM) design pattern as it is a typical application of MVVM. For the pairs with the consistent label "unrelated", the association between the word and the design pattern is usually too weak to make sense. They may just be mentioned in a same document occasionally, for instance, Sharding - "excel", Iterator - "message",

and Decorator - "plugin". The words whose meanings tend to be very general, such as "idea", "make", and "sometime", are also labelled as "unrelated" to any design patterns as it is difficult to specify a scenario that they can be related. Except for the consistently labelled ones, some pairs are controversial. For example, "dismiss" can represent a specific operation in the ViewController design pattern. However, it is also somewhat a general meaning word. Two participants judge it to be related to ViewController but the other one labels "unrelated". Hence, the final label is "somewhat related". To measure the degree of agreement among the participants, we calculate the Fleiss' Kappa. The value is 0.6421, which means a substantial agreement. Therefore, the labelling results are relatively reliable.

After the labelling process, 369 pairs (18.45 percent) are labelled as "related", 457 pairs (22.85 percent) are labelled as "somewhat related", and 1,174 pairs (58.7 percent) are with the label "unrelated".

4.2 Baseline Algorithms

There exist several similarity methods to estimate semantic relatedness between natural language words. We take three categories of intensively used methods as baselines. This categorization can cover that adopted in [25].

4.2.1 Latent Semantics Based Methods

In this category of methods, the words and design patterns are represented by latent variable vectors. Then the relatedness between a word and a design pattern can be measured by the cosine similarity.¹¹ This category includes Latent Semantic Indexing (LSI) and Latent Dirichlet Allocation (LDA).

LSI (also known as Latent Semantic Analysis, LSA) is an unsupervised algorithm of analyzing the relationships between documents and terms by producing a set of latent semantic concepts [26]. It has been used in estimating semantic relatedness in source code [25]. In the evaluation, the input of LSI is the $term \times document$ matrix, in which an element represents the frequency of a term (word or design pattern) appearing in a document. Then the words and the design patterns are represented in a low-dimensional (latent) space by applying singular value decomposition. The dimension of the latent space is initially set as 10 and then gradually increased. During this process, the performance of LSI is evaluated. The value which achieves the best performance is retained and recorded. Finally, the dimension is set as 400.¹²

LDA is a topic modeling technique that has been used for analysing software-specific data in several studies [20], [27], [28], [29]. To use LDA in the evaluation, each document in the corpus C is represented as a bag of words and design patterns without order. With the Gibbs sampling based implementation of LDA [30], each word or design pattern in a document is assigned to a topic. By considering the whole corpus, the words and the design patterns can be represented as probability distributions over topics. The topic

11. https://en.wikipedia.org/wiki/Cosine_similarity

12. In fact, the performance of LSI in terms of $NDCG@40$ and Spearman's ρ does not change much when the dimension is larger than 250. The details are shown on <https://github.com/WoodenHeadoo/dpword2vec/blob/master/baselines/LSI.md>.

number is set to 40 as it has been shown to be appropriate for the Stack Overflow dataset [28].

4.2.2 Co-occurrence Based Methods

Co-occurrence based methods calculate the similarity (or distance) between a word and a design pattern directly based on their co-occurrences, including Pointwise Mutual Information (PMI) and Normalized Google Distance (NGD).

PMI is an intuitive and computationally efficient relatedness method for massive corpora of textual data [31]. NGD is a semantic distance measure between words or phrases based on information distance and Kolmogorov complexity [32]. It has been verified to be effective in quantifying semantic relatedness between individual code terms (named Normalized Software Distance, NSD) [25]. Since NGD is a distance measure, the similarity can be obtained by negating the value of NGD. Both PMI and NGD take the frequency of a word (i.e., the number of documents containing the word), the frequency of a design pattern (i.e., the number of documents containing the design pattern), and the frequency of the co-occurrence (i.e., the number of documents containing both the word and the design pattern) in the corpus C as input, but calculate the measures in different ways.

4.2.3 Vector Space Model Based Method

Another baseline is the Vector Space Model (VSM). Specifically, we use the Term Frequency - Inverse Document Frequency (TF-IDF) [33] schema to model the text. By multiplying each row of the $term \times document$ matrix (which is also the input of LSI) by the IDF value of the corresponding term, we obtain a matrix of TF-IDF values. Each row of the TF-IDF matrix can be regarded as the vector of the corresponding term (word or design pattern), which indicates the TF-IDF value of the term in each document. With these term vectors, the dp-word similarity can be also obtained by calculating the cosine similarity. Actually, the calculation of the IDF values is redundant in this case. Since the IDF weighting is operated on each entire term vector, the multiplied IDF values are eliminated automatically when calculating the cosine similarities. Therefore, this model is equivalent to represent a term with a row of the $term \times document$ matrix.

4.2.4 Software-Specific Method

In the evaluation, we consider a domain-specific method, WordSimSE, which aims to build WordNet like resources for software [24]. WordSimSE is a composite method that measures the similarity between terms by combining weighting strategy and co-occurrences. We use the WordSimSE method to calculate the dp-word similarities based on the corpus C . Moreover, there are three parameters to be clarified. According to the definition in [24], a word or a design pattern can be classified into one of the three groups: popular software tag, if it is a top 10 percent most frequent Stack Overflow tag; non-popular software tag, if it is a Stack Overflow tag but not in the top 10 percent; and ordinary term, otherwise. The three groups are weighted with three different parameters, namely 2.8, 2.0, and 1.4, which are also used in [24].

4.3 Evaluation Metrics

In our built dataset, each design pattern is paired with 40 words, which are labelled as “related”, “somewhat related”, or “unrelated” to the design pattern. We want to investigate whether the similarity scores given by the similarity methods could correspond with the labelled ones. To this end, we use two metrics for evaluation, namely NDCG and Spearman’s rank correlation coefficient.

Normalized Discounted Cumulative Gain (NDCG) is a measure of ranking quality in information retrieval and employed in several software engineering tasks [34], [35], [36]. For each design pattern, a similarity method ranks the 40 words in descending order according to their similarity scores. The measure $NDCG@k$ is calculated as

$$NDCG@k = \frac{DCG@k}{IDCG@k}, DCG@k = \sum_{i=1}^k \frac{r_i}{\log_2(i+1)}, \quad (6)$$

where r_i denotes the degree of relevancy of the i th ranked word and its permissible values are 3 (“related”), 2 (“somewhat related”), and 1 (“unrelated”). $IDCG@k$ is the ideal value of $DCG@k$ that normalizes the measure into [0,1].

Spearman’s rank correlation coefficient (Spearman’s ρ) is a non-parametric measure of rank correlation which is usually used in the evaluations of word similarity tasks [13], [18], [23]. It represents the correlation between the ranks of the 40 words based on the similarity scores of a similarity method and the ranks based on the labelled relevance scores. However, there are only three unique labelled relevance scores in our dataset. Following [37], words with a same score are assigned with a same average fractional rank. Specifically, after ranking the 40 words according to the three labels, we assume that the first m_1 words are “related”, the middle m_2 words are “somewhat related”, and the last m_3 words are “unrelated”. The rank of the “related” words is $\frac{1}{m_1} \cdot \frac{(1+m_1)m_1}{2} = \frac{m_1+1}{2}$, the rank of the “somewhat related” words is $m_1 + \frac{m_2+1}{2}$, and the rank of the “unrelated” words is $m_1 + m_2 + \frac{m_3+1}{2}$. Then the coefficient is calculated as

$$\rho = 1 - \frac{6 \sum_{i=1}^N d_i^2}{N(N^2 - 1)}, \quad (7)$$

where $N = 40$, denotes the length of the rank list, and d_i is the difference between the two ranks of the i th word.

5 EVALUATION RESULTS

In this section, we investigate the following four research questions (RQs) to evaluate different aspects of DPWord2Vec.

5.1 RQ1: How Do the Settings of the Parameters Affect the Performance of DPWord2Vec?

5.1.1 Motivation

The performance of DPWord2Vec may vary when using different settings. In this RQ, we investigate how DPWord2Vec performs under different values of the parameters, i.e., the dimension of the vectors, the size of context window for words, and the ratio of the weights of the two corpora.

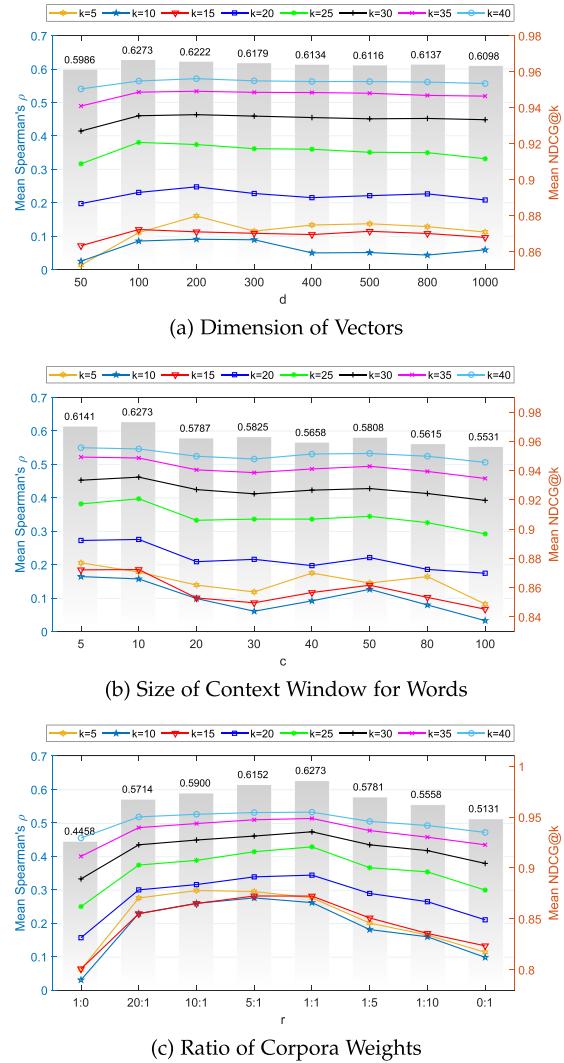


Fig. 3. Mean NDCG and Spearman's ρ of DPWord2Vec under different parameter settings on the 50 design patterns.

5.1.2 Approach

Each of the three parameters is investigated independently. Specifically, we adjust the value of one parameter and analyse how the performance of DPWord2Vec changes. Meanwhile, the other two parameters keep fixed.

We change the value of the vector dimension (d) from 50 to 1,000, including 50, 100, 200, 300, 400, 500, 800, and 1,000. The value of the context window size for words (c) varies from 5 to 100, including 5, 10, 20, 30, 40, 50, 80, and 100. Moreover, we explore the importance of the description corpus and the crowdsourced corpus under different ratios of weights (r). The ratio $m : n$ indicates that each document in the description corpus and each document in the crowdsourced corpus are added into the final corpus for m and n times, respectively.

5.1.3 Results

The results for the three parameters are presented respectively at follows.

Dimension of Vectors. The fold lines in Fig. 3a plot how the mean values of $NDCG@k$ change with different vector dimensions. For simplicity, we only show the results for $k =$

5, 10, ..., 40. The bars in Fig. 3a show the mean values of Spearman's ρ on different vector dimensions. The settings of the other two parameters are $c = 10$ and $r = 1:1$. In Fig. 3a, we notice that all the fold lines have similar trends. The values of NDCG rise slightly when the vector dimension varies from 50 to 200 and then keep stable as the vector dimension increases further. Meanwhile, by referring to the bars, a similar trend can also be found on Spearman's ρ . In general, the performance of DPWord2Vec is not very sensitive to the vector dimension in terms of NDCG and Spearman's ρ .

The dimension of the vector controls over the granularity of the representation of a word or a design pattern. A larger vector dimension tends to produce more fine-grained and detailed vector representations. However, the performance cannot further improve when the vector dimension is larger than 200. It may imply that the representations of words and design patterns reach the saturations at this vector dimension based on the current model and corpus.

Size of Context Window for Words. The values of NDCG and Spearman's ρ under different settings are presented in Fig. 3b as line chart and bar chart, respectively. The other two parameters are fixed at $d = 100$ and $r = 1:1$. As shown in the figures, both NDCG and Spearman's ρ all have an approximately descending trend as the context window size increases, especially from $c = 10$ to $c = 20$. The performance at $c = 5$ is comparable to that at $c = 10$. For example, $NDCG@40$ is 0.9556 at $c = 5$ and 0.9548 at $c = 10$, the former is slightly better; Spearman's ρ is 0.6141 at $c = 5$ and 0.6273 at $c = 10$, the later is slightly better. Generally, the descending trends are not very significant.

The context window size in DPWord2Vec only affects the context windows for words, it determines the number of surrounding words that a word is associated with. Too large context window size results in too many surrounding words that would diminish the syntactic information. It may lead to low-quality vector representations of words and design patterns, and then impairs the performance.

Ratio of Corpora Weights. The results are shown in Fig. 3c. The other two parameters are set as $d = 100$ and $c = 10$. From the figures, we notice that the values of both NDCG and Spearman's ρ reach their peaks at $r = 1:1$, i.e., when the two corpora are directly mixed. The performance at $r = 5:1$ is the most similar one to that at $r = 1:1$. When changing the ratio, the performance drops and reaches the worst in the two directions at $r = 1:0$ and $r = 0:1$. That means, we will get bad results when using only one of the two corpora.¹³

From the results, we can conclude that both the description corpus and the crowdsourced corpus are all indispensable for good performance. Although the description corpus is much smaller than the crowdsourced corpus, its effects cannot be neglected. The description corpus may stand for "quality" which supplies precise descriptions of design patterns, and the crowdsourced corpus stands for "quantity" which provides rich textual data relevant to design patterns.

13. Some words or design patterns may be out of the vocabulary when using only one corpus. In this case, the vectors are represented as random initial values. It may be a reason for the bad results. Nevertheless, it also implies that neither of the corpora is negligible.

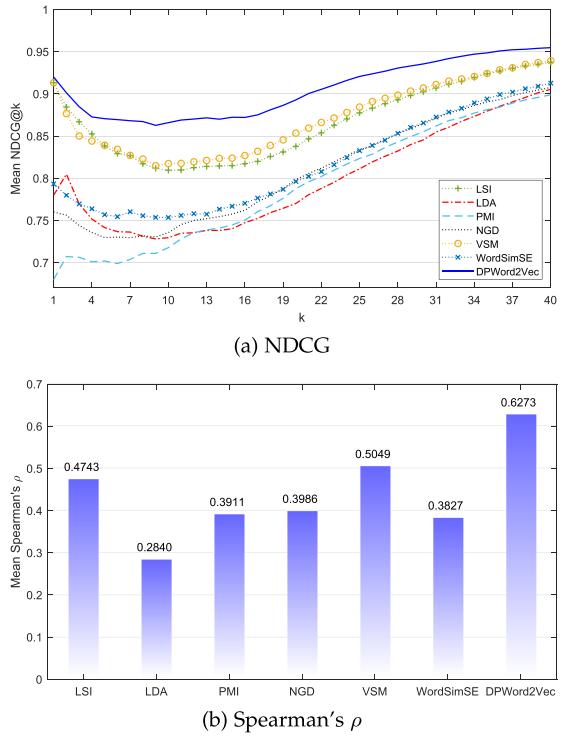


Fig. 4. Mean NDCG and Spearman's ρ of each baseline algorithm and DPWord2Vec on the 50 design patterns.

5.1.4 Conclusion

Generally, the performance of DPWord2Vec is not very sensitive to the dimension of vectors, but the settings of the context window size and the corpora weights affect the performance. To get a good performance, the context window size for words should not be too large, and the description corpus and the crowdsourced corpus should be balanced. The following experiments are all based on the settings that $d = 100$, $c = 10$, and $r = 1:1$.

5.2 RQ2: Does DPWord2Vec Outperform the Baseline Algorithms in the dp-Word Similarity Task?

5.2.1 Motivation

In this RQ, we explore whether DPWord2Vec can be superior to the baseline algorithms in dp-word similarity estimation.

5.2.2 Approach

We compare DPWord2Vec against the six baseline algorithms, namely LSI, LDA, PMI, NGD, VSM, and WordSimSE, on our dp-word pair dataset. The two metrics, i.e., NDCG and Spearman's ρ , are applied for evaluation.

5.2.3 Results

Fig. 4a shows the mean values of $NDCG@k$ of the five algorithms and DPWord2Vec over the 50 design patterns on various k . Fig. 4b presents the averaged value of Spearman's ρ of these algorithms. As shown in Fig. 4a, DPWord2Vec almost outperforms all the baseline algorithms for all values of k . For example, $NDCG@40$ of DPWord2Vec is 0.9548,

which outperforms those of LSI, LDA, PMI, NGD, VSM, and WordSimSE by 0.0173, 0.0494, 0.0559, 0.0472, 0.0155, and 0.0421, respectively. In Fig. 4b, DPWord2Vec outperforms LSI, LDA, PMI, NGD, VSM, and WordSimSE by 32.3, 120.9, 60.4, 57.4, 24.2, and 63.9 percent respectively in terms of Spearman's ρ . As the metrics are only shown in mean values, we use Wilcoxon signed rank test [38] to investigate whether there are significant differences between the performance of DPWord2Vec and the baseline algorithms over the 50 design patterns. For $NDCG@40$, the p-values when comparing DPWord2Vec against the baseline algorithms are all less than 3e-6. For Spearman's ρ , the corresponding p-values are all less than 1e-7. That means, DPWord2Vec significantly outperforms the baseline algorithms in terms of NDCG and Spearman's rank correlation coefficient.

Among the baseline algorithms, LSI and VSM achieve better performance and the other four have somewhat comparable performance when considering NDCG and Spearman's ρ . We note that LSI and VSM are all based on the *term × document* matrix. It means that this way of text representation is relatively suitable for this task. The software specific method, WordSimSE, does not perform quite well in the evaluation. A possible reason is that there are differences between the software domain and the design pattern domain, as design patterns are universal solutions to recurring design problems and tend to be independent of specific software entities.

To gain more intuitions of how the algorithms perform, we give an example of ranked lists of these algorithms. Table 2 shows the top ten most related words to the design pattern Record Set [39] ranked by each algorithm. For DPWord2Vec, the ten words are all labelled as "related" or "somewhat related" to the design pattern Record Set. The top ten lists of the other algorithms all contain "unrelated" words, which are shown in boldface. For example, for LDA, PMI, and NGD, the top ten lists are contaminated by the noise word "jone". The word "jone" is a person name and usually used as an example of username when discussing database records in Stack Overflow (e.g., post #10050790). However, "jone" is not semantically related to Record Set. The top ten lists of LSI, VSM, and WordSimSE contain words with too general or vague meanings, e.g., "try", "get", and "use".

5.2.4 Conclusion

DPWord2Vec significantly outperforms the baseline algorithms on the dp-word similarity task in terms of NDCG and Spearman's ρ .

5.3 RQ3: Does the Usage of the New Context Windows Contribute to the Performance of DPWord2Vec?

5.3.1 Motivation

In DPWord2Vec, we define new context windows for design patterns and words respectively (Section 3.3). In this RQ, we explore whether the usage of these context windows is an advisable choice to associate design patterns with words.

5.3.2 Approach

To investigate the effects of the new context windows, we replace them with the traditional fixed context windows

TABLE 2
The Top 10 Most Related Words to Record Set
Design Pattern of Each Algorithm

LSI	LDA	PMI	NGD	VSM	WordSimSE	DPWord2Vec
recordset	querydef	recordset	recordset	recordset	recordset	recordset
record	recordset	tado	tado	record	row	row
querydef	clause	querydef	querydef	row	record	record
tado	statement	jone	row	try	value	clause
clause	row	pivot	pivot	tado	string	value
row	id	statement	record	use	execute	string
statement	jone	row	clone	value	try	array
exit	index	record	jone	get	id	pivot
try	record	clone	clause	need	server	index
modify	find	clause	statement	array	get	querydef

used in Word2Vec [11] and repeat the experiments on the dp-word pair dataset. As the words and the design patterns are independent in the corpus C , we use two strategies to integrate words and design patterns into sequences, namely the *occurrence strategy* and the *shuffling strategy*, so that they can be handled by the traditional context windows.

The design pattern name occurrences strategy is to detect the occurrences of design pattern names in the text as design pattern tokens. This strategy is discussed in Section 3.3. The shuffling strategy is leveraged in a recent study to align words and APIs into a fixed context window [40]. Following [40], for a document doc , the words in $doc.Tokens$ and the design patterns in $doc.DPs$ are merged and randomly shuffled for ten times to produce ten token sequences (containing both words and design patterns).

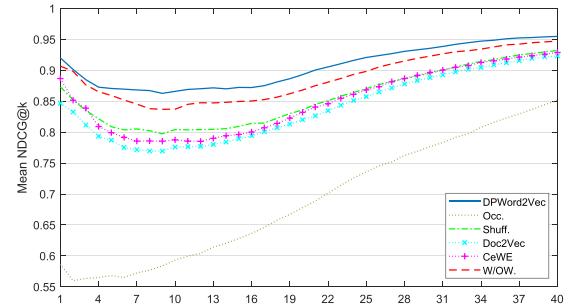
Moreover, we also consider two other strategies which represent design patterns in higher levels rather than token level. They are from Doc2Vec [41] and Category enhanced Word Embedding (CeWE) [42], respectively. The original Doc2Vec aims to embed words and paragraphs or documents into vector spaces. Based on this model, we regard a design pattern as a document-level term to learn its vector representation. Specifically, the vector of each document in Doc2Vec is substituted with the vector of the design pattern which is contained in the document. Each design pattern in V_{DP} always keeps a unique vector even if it appears in different documents. However, a document may contain multiple design patterns. In this case, its word tokens ($doc.Tokens$) are duplicated multiple times so that each duplicate can be combined with a design pattern. Recently, Nguyen *et al.* have used the same approach to produce the vector representations of APIs and words [43].

Likewise, CeWE can learn the vector representations of words as well as categories. A category indicates a label or a classification of documents. A document may belong to multiple categories. In this study, we regard each design pattern as a category. In this way, design patterns are also associated with words in document level and their vectors can be obtained accordingly.

For all the strategies above, the parameters, including the dimension of the vectors, the size of context window, the initial learning rate, and the number of iterations, are the same as in Section 3.4. As introduced in [42], the parameter λ of CeWE is set to be $1/(2 \cdot c + 1)$, where c is the size of context window.

5.3.3 Results

The results are shown in Figs. 5a and 5b in terms of NDCG and Spearman's ρ , respectively. As shown in the figures, we



(a) NDCG

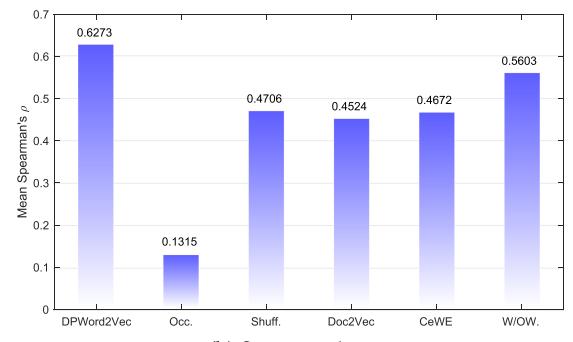
(b) Spearman's ρ

Fig. 5. Mean NDCG and Spearman's ρ of the variants of DPWord2Vec on the 50 design patterns. Occ. = DPWord2Vec with the occurrence strategy, Shuff. = DPWord2Vec with the shuffling strategy, Doc2Vec = DPWord2Vec with the strategy of Doc2Vec, CeWE = DPWord2Vec with the strategy of CeWE, W/O/W. = DPWord2Vec without the weighting strategy.

notice that the performance of DPWord2Vec with the *occurrence strategy* (Occ.) is poor. For example, the Spearman's ρ is 0.1315, even worse than all the baseline algorithms in Section 5.2. DPWord2Vec with the *shuffling strategy* (Shuff.), and the strategies of Doc2Vec and CeWE, achieve comparable performance. Among them, the *shuffling strategy* tends to be slightly better than the other two, but still surpassed by DPWord2Vec with the new context windows. According to Wilcoxon signed-rank test, the differences between the performance of the default DPWord2Vec and that with the other strategies on $NDCG@40$ and Spearman's ρ are statistically significant (p-values are all less than $1e-5$).

The drawback of the *occurrence strategy* is obvious. As the design pattern names tend to be sparse in the text, it is hard to mine the relationships between words and design patterns adequately by leveraging the context windows. With regard to the *shuffling strategy*, it may break the structure of the natural language sentences and do harm to the capture of semantic relationships. Moreover, the shuffling process will significantly increase the size of the corpus (almost ten times the original one) which results in extra computation complexity.

The two document-level strategies, i.e., that from Doc2Vec and CeWE, have similar mechanisms. The core is that, in each document, the vectors of the design patterns are integrated with the vectors of the surrounding words in a context window to predict the central one. Hence, design patterns can be deemed to be contained in the context of words in some way. However, there exists no similar context for design patterns and the design patterns in a document are not predicted by the vectors of the involving

words. Compare to these strategies, the new context windows can build stronger ties between design patterns and words.

5.3.4 Conclusion

DPWord2Vec with the new context windows can achieve better results than the variants with the two serializing strategies and the two document-level strategies. Thus, the usage of the new context windows does contribute to the performance of DPWord2Vec.

5.4 RQ4: Does the Weighting Strategy Contribute to the Performance of DPWord2Vec?

5.4.1 Motivation

A weighting strategy is applied in the training phase of DPWord2Vec (Section 3.4). To verify whether this strategy is redundant, we set up this RQ.

5.4.2 Approach

We construct a variant of DPWord2Vec by removing the weighting strategy. Then the performance of the variant is compared against that of the default DPWord2Vec.

5.4.3 Results

The results are also presented in Figs. 5a and 5b (W/OW.). In Fig. 5a, we observe that there are minor effects on $NDCG@k$ with small k after removing the weighting strategy. The differences are obvious when k is larger than five. When considering all the 40 words for each design pattern, the values of $NDCG@40$ and Spearman's ρ after removing are respectively 0.9471 and 0.5603, which are all worse than the original ones, i.e., 0.9548 and 0.6273. As the mean values seem to be close, we perform Wilcoxon signed-rank test on $NDCG@40$ and Spearman's ρ . The p-values are respectively 2.72e-3 and 1.05e-5, which indicates the differences are significant according to the $p < 0.05$ standard. Moreover, we quantify the magnitude of the difference of performance by analysing the effect size. Specifically, Cohen's d [44] is calculated to measure the differences between the means of the metrics with and without the weighting strategy. The results for $NDCG@40$ and Spearman's ρ are 0.2959 and 0.6087, which indicate a small-medium effect size and a medium-large effect size [45], respectively. That means, the effect of the weighting strategy on the performance is not negligible.

Based on the results, we note that DPWord2Vec achieves better performance with the weighting strategy, especially in terms of $NDCG@k$ with $k > 5$. Without the weighting strategy, the irrelevant but frequent words, such as "get" and "case", may be included in the top k list with a relatively large k and ranked ahead of the ones which are labelled as "related". The weighting strategy could effectively weaken the relationships between the design patterns and these words, thus improves the performance.

5.4.4 Conclusion

DPWord2Vec can benefit from the weighting strategy for measuring dp-word similarity.

6 APPLICATION I: DESIGN PATTERN TAG RECOMMENDATION

Many software information sites allow developers to label their posts with tags, such as Stack Overflow, Ask Ubuntu, and Freecode. Tags are short descriptions within a few words long that are provided as metadata to classify, identify, and search software objects in these sites [46]. To improve the quality of tags in software information sites, a series of automatic tag recommendation methods have been proposed to recommend appropriate tags for new posts based on existing tag candidates [47], [48], [49], [50], [51]. In this application, we consider a design pattern specific tag recommendation task that recommends design pattern tags for design pattern relevant posts. That is, each recommended tag is a design pattern. By the recommendations, the synonymous design pattern tags could be better avoided, which results in better information organization and retrieval for design pattern relevant posts.

6.1 Common Methods for Tag Recommendation

Actually, the design pattern tag recommendation task can also be accomplished by general tag recommendation methods. We briefly introduce the methods for tag recommendation.

The main intuition of the existing tag recommendation methods is to use the historical information of tag assignments to recommend tags for new posts. Concretely, the tag recommendation methods analyse the existing posts and their tags in a software information site, and then infer the relationship between a tag and a word or a whole post. When a new post is coming, the same analysis process is deployed on this post with the inferred results and each tag is given a likelihood score. The top few tags with the highest likelihood scores will be recommended. By restricting the tags to design pattern tags, i.e., each tag represents a design pattern, these methods are directly applied in the design pattern tag recommendation task.

6.2 Design Pattern Tag Recommendation Based on DPWord2Vec

In this part, we explain how to recommend design pattern tags by leveraging DPWord2Vec.

With DPWord2Vec, design patterns and natural languages are associated. We can use these associations for design pattern tag recommendation. As the content of a post is a typical document that contains multiple words, to recommend design pattern tags for a post, the relationship between a design pattern and a document should be built based on the word and design pattern vectors. Therefore, we adopt the text semantic similarity [52] to measure the relatedness between a design pattern and a set of words

$$\begin{aligned} Sim(Words, dp) = \frac{1}{2} & \left[\frac{\sum_{w \in Words} IDF(w) \cdot Sim(w, dp)}{\sum_{w \in Words} IDF(w)} \right. \\ & \left. + \max_{w \in Words} Sim(w, dp) \right], \end{aligned} \quad (8)$$

where $IDF(w)$ is the inverse document frequency¹⁴ value of the word w in the corpus C and $Sim(w, dp)$ is the vector cosine similarity between w and the design pattern dp .

14. <https://en.wikipedia.org/wiki/Tf-idf>

Generally, given a new design pattern relevant post, there are three steps for design pattern tag recommendation:

- 1) Preprocess and tokenize the textual description of the post following the procedures in Section 3.2.
- 2) For each design pattern tag in the tag candidate set, calculate the similarity between the design pattern and the post as Formula (8).
- 3) Rank the design pattern tags in descending order according to their similarities and recommend the top k design pattern tags.

6.3 Evaluation on Design Pattern Tag Recommendation

6.3.1 Motivation

In the evaluation, we try to explore whether the DPWord2Vec-based method performs better than the common tag recommendation methods on the design pattern tag recommendation task.

6.3.2 Approach

To evaluate the effectiveness of the DPWord2Vec-based method, we compare it against the state-of-the-art tag recommendation algorithms on a real-world dataset. We detail the strategies for evaluation, the constructed datasets, the state-of-the-art tag recommendation algorithms, and the leveraged metrics as follows.

Strategies. As to our knowledge, there are two software information sites in which design patterns are broadly discussed: Stack Overflow and Software Engineering.¹⁵ However, on one hand, the posts in Stack Overflow have been leveraged by DPWord2Vec, it is inappropriate to use them to evaluate DPWord2Vec again. On the other hand, the amount of design pattern relevant posts in Software Engineering is relatively small (less than 3,000, a dataset of tag recommendation usually contains more than 13,000 posts [47], [48], [49], [50], [51]), it may be detrimental for the other tag recommendation algorithms to train proper models based on these posts. Therefore, the main strategy for evaluation is to use the Software Engineering posts for testing, and use the Stack Overflow posts to train tag recommendation models.

Datasets. We download the Stack Overflow posts (from August 2008 to December 2017) and the Software Engineering posts (from September 2010 to March 2019) to construct the datasets. Before that, the design pattern tags should be detected. At first, we construct the regular expressions for the names of each design pattern in V_{DP} . Specifically, each design pattern name is split into word(s), i.e., $word_1, word_2, \dots, word_n$, and the regular expression is written as “ $word_1?word_2\dots?word_n(-pattern)?$ ” (as words can only be separated by hyphens in tags). In this way, the tags like “active-record”, “activerecord”, and “active-record-pattern” can all be matched with the design pattern name “active record”. Next, all the tags of these posts are extracted and a tag is mapped to a design pattern if it matches with a name of the design pattern via the corresponding regular expression. Then, we manually review each mapped tag if it has a description in the corresponding software information site

to filter out false-positive tags that do not denote the design patterns. At last, multiple tags are merged into one tag if they are mapped to the same design pattern. Finally, 94 and 36 design pattern tags are detected in Stack Overflow and Software Engineering, respectively. In this way, the design pattern tags of the two sites are unified and these tags have a one to one correspondence with the design patterns.

With the design pattern tags, we construct two datasets: a dataset for training the common tag recommendation models and a dataset for testing the common models and the DPWord2Vec-based model. To build the training set, we extract the Stack Overflow posts that contain the design pattern tags but discard the tags appearing in less than 50 posts as they are less interesting and less useful to serve as representative tags [47]. For the test set, we extract the Software Engineering posts that contain the design pattern tags but discard the tags not appearing in the training set as they cannot be recommended by the common tag recommendation algorithms. Finally, the training set contains 176,427 Stack Overflow posts and 74 design pattern tags which are used as candidates, the test set contains 2,986 Software Engineering posts and 35 design pattern tags.¹⁶ Like the training set here, the crowdsourced corpus, which is for training the design pattern and word vectors, is also constructed based on the Stack Overflow posts. It should be noted that they are distinct. The crowdsourced corpus consists of the posts with at least one design pattern name appearing in the titles, bodies, or tags. It involves 210 design patterns in total. In contrast, the training set only cares about the posts containing design pattern tag(s). The latter can be roughly covered by the former.

According to the settings above, the common tag recommendation models are trained on the Stack Overflow posts containing the design pattern tags. Meanwhile, our DPWord2Vec-based model relies on the design pattern and word vectors learnt from the corpus C . In other words, these models do not have a consistent training set. To achieve unbiased comparisons, we conduct another part of evaluation in which the corpus C is also used for training the common tag recommendation models. Specifically, each document in C is regarded as a post and each design pattern in a document is regarded as a design pattern tag. Then, all the 372 design patterns in V_{DP} serve as candidates.

State of the Arts. To the best of our knowledge, there are three common tag recommendation algorithms, TagMulRec [49], EnTagRec++ [50], and FastTagRec [51], shown to be the state-of-the-art on software information sites. Similar to word embedding models, FastTagRec represents words as vectors and recommends tags using neural network-based classification. Given a new post, TagMulRec first locates the posts that are semantically similar to it, and then exploits multi-classification to produce a ranked tag list. EnTagRec++ integrates the historical tag assignments and the information of users for tag recommendation. However, EnTagRec++ cannot be applied here as the training set and the test set are from different sites which do not share the same group of users. Therefore, we only take TagMulRec and FastTagRec for comparisons.

¹⁶ The training and test sets, as well as the original tag - design pattern mappings are available on <https://github.com/WoodenHeadoo/dpword2vec>.

15. <https://softwareengineering.stackexchange.com/>

In addition, with the concern that the design pattern names may appear in the posts explicitly, we deploy a baseline method which leverages the occurrences of design patterns. Specifically, the design pattern names of each design pattern in the tag candidate set are searched in the Software Engineering posts (the test set) by using the regular expressions (as discussed in Section 3.1). A post is supposed to contain a design pattern tag if one of the design pattern names appears in the title or body of the post. Since the common tag recommendation methods only provide likelihood scores for ranking the candidate tags, for this baseline method, the design pattern tags are also sorted according to the numbers of design pattern occurrences for comparability. If there are no or not enough design pattern occurrences found in the post, the design pattern tags are sorted in alphabetical order.

Metrics. The recommending strategy of all the algorithms above is to provide a rank list of candidate design pattern tags and recommend the top k ones. To evaluate the recommendations, we exploit three metrics, $Recall@k$, $Precision@k$, and $F1 - score@k$, which are usually used to evaluate tag recommendation systems on software information sites [49], [51]. In particular, the sample-wise metrics are calculated as

$$Recall@k_i = \frac{|RankList_i^k \cap Tag_i|}{|Tag_i|}, \quad (9)$$

and

$$Precision@k_i = \frac{|RankList_i^k \cap Tag_i|}{k}, \quad (10)$$

where Tag_i and $RankList_i^k$ are the set of real design pattern tags and the set of top k recommended design pattern tags for the i th posts in the test set, respectively. By combining $Recall@k_i$ and $Precision@k_i$

$$F1 - score@k_i = \frac{2 \cdot Recall@k_i \cdot Precision@k_i}{Recall@k_i + Precision@k_i}. \quad (11)$$

Then the set-wise metrics $Recall@k$, $Precision@k$, and $F1 - score@k$ are respectively the average values of the sample-wise metrics in Formulas (9), (10), and (11) over all the posts in the test set. According to the literature [47], [48], [49], [50], [51], k is set to 5 and 10.

6.3.3 Results

As introduced before, the evaluation contains two parts. In the first part, the Stack Overflow posts with the design pattern tags are used for training the TagMulRec model and the FastTagRec model, the Software Engineering posts are used for testing all the models. The tag candidate set for recommendation includes the 74 design pattern tags appearing in these Stack Overflow posts. The results are shown in Table 3. The best result on each metric is shown in boldface. As shown in the table, the DPWord2Vec-based method achieves much better performance than TagMulRec, i.e., over 30 percent improvements on all metrics. When comparing against FastTagRec, the improvements are not so apparent, i.e., all within 10 percent. We perform Wilcoxon signed-rank test on sample-wise metrics of all the 2,986 posts and the p-values on the six metrics are all less than 0.0025 when comparing DPWord2Vec

TABLE 3
The Results on the Design Pattern Tag Recommendation Task
(Stack Overflow Posts for Training TagMulRec and FastTagRec,
the 74 Design Pattern Tags in Stack Overflow as Candidates)

	Baseline	TagMulRec	FastTagRec	DPWord2Vec
<i>Recall@5</i>	0.7369	0.5279	0.8167	0.8399
<i>Precision@5</i>	0.1618	0.1123	0.1786	0.1837
<i>F1 – score@5</i>	0.2625	0.1838	0.2901	0.2984
<i>Recall@10</i>	0.7369	0.6954	0.8658	0.9230
<i>Precision@10</i>	0.0809	0.0749	0.0952	0.1017
<i>F1 – score@10</i>	0.1448	0.1345	0.1704	0.1820

against FastTagRec. That means, the DPWord2Vec-based method significantly outperforms FastTagRec in statistics.

In the second part, we train the TagMulRec model and the FastTagRec model using the corpus C and test all the models with the Software Engineering posts. The candidates are changed to all the 372 design patterns in V_{DP} . Table 4 presents the evaluation results. From the table, we notice that the performance of TagMulRec and FastTagRec improves on all the metrics contrast to the previous ones, but is still not as good as that of the DPWord2Vec-based method. The DPWord2Vec-based method is relatively stable as the results are almost unchange when involving more design pattern tag candidates. According to the results of Wilcoxon signed-rank test, the differences on $Recall@5$, $Precision@5$, and $F1 - score@5$ are not significant, i.e., the p-values are 0.22, 0.44, and 0.19, respectively. However, the DPWord2Vec-based method still significantly outperforms FastTagRec when recommending ten design pattern tags, i.e., p-values on $Recall@10$, $Precision@10$, and $F1 - score@10$ are all less than 1e-6. It implies that the DPWord2Vec-based method benefits from not only a comprehensive corpus but also an appropriate algorithmic model.

As shown in Tables 3 and 4, it is surprising that the performance of the baseline method is better than that of TagMulRec on all metrics, although surpassed by that of FastTagRec and the DPWord2Vec-based method. That means, to detect the design pattern occurrences is also effective for design pattern tag recommendation to some degree. From the perspective of Recall, the names of a part of the design patterns serving as tags appear in the text of the posts as well. But it does not achieve a quite ideal coverage. From the perspective of Precision, an occurrence of a design pattern name in a post does not necessarily mean that it is also a tag of the post, as the design pattern may be not the main focus or the mentioned design pattern name is ambiguous. Comparing Table 4 against Table 3, we notice that the baseline method has minor changes in performance when enlarging the tag candidate set. The reason is that the newly involved design patterns appear rarely in the posts.

Generally, the performance of the DPWord2Vec-based method is relatively close to that of FastTagRec. Nevertheless, there are some advantages of our method. On the one hand, the DPWord2Vec-based method is more efficient than FastTagRec. As DPWord2Vec is based on the GloVe model, the time complexity for calculating and updating the gradients is usually $O(d(|C|^{1/\alpha} + |DPTags|^{1/\beta}))$ for some $\alpha, \beta > 1$ [13], where d denotes the dimension of the vectors, $|C|$ denotes the total number of word tokens, and $|DPTags|$ denotes the total

TABLE 4
The Results on the Design Pattern Tag Recommendation Task
(Corpus C for Training TagMulRec and FastTagRec, All the 372 Design Patterns as Candidates)

	Baseline	TagMulRec	FastTagRec	DPWord2Vec
Recall@5	0.7358	0.5559	0.8322	0.8399
Precision@5	0.1615	0.1183	0.1826	0.1837
F1-score@5	0.2620	0.1936	0.2963	0.2984
Recall@10	0.7369	0.7040	0.8895	0.9224
Precision@10	0.0809	0.0758	0.0978	0.1017
F1-score@10	0.1448	0.1361	0.1750	0.1819

number of design pattern tag occurrences in the training set. As $|DPTags|$ ought to be much smaller than $|C|$, the time complexity can be written as $O(d \cdot |C|^{1/\alpha})$. For FastTagRec, the time complexity is $O(d \cdot |C| \cdot \log(|DPCands|))$ [51], where $|DPCands|$ denotes the size of the design pattern tag candidate set. Hence, the DPWord2Vec-based method is more scalable when involving more posts for training ($|C|$ gets larger). Moreover, enlarging the tag candidate set will make the model of FastTagRec more complex, but not explicitly increase the model complexity of the DPWord2Vec-based method. On the other hand, the DPWord2Vec-based method is more understandable. FastTagRec is essentially a classification model. It regards each design pattern tag candidate as a class and recommends tags by training the classifier. However, the classifier is somewhat a black-box for the users. In contrast, DPWord2Vec represents the elements of the natural language and the tag candidates as vectors, and ranks the tags according to the similarities between them and the post. It tends to be more intuitive and acceptant for humans. Moreover, by exploring the sentences or phrases with high similarities to the tags, the users could understand the motivation of the recommendation better.

6.3.4 Conclusion

In the design pattern tag recommendation task, the DPWord2Vec-based method performs better than TagMulRec and FastTagRec in terms of Recall, Precision, and F1-score, even when they are provided with the same data for training. This shows that the learned word and design pattern vectors could better express the relationships between a post and a design pattern.

7 APPLICATION II: DESIGN PATTERN SELECTION

When developing a software (sub)system, the developer(s) may be willing to leverage design patterns to facilitate the development process. This is called a design problem. However, there exist a large number of design patterns [7] and determining the applicability of these design patterns heavily depends on the experience of a developer [53]. It is usually difficult to find the right design pattern(s) for a given design problem especially for novice developers [8]. To resolve this problem, several studies focus on selecting appropriate design pattern(s) automatically based on the textual description of the design problem [8], [54]. The textual description is a short text that may depict the main features, requirements of the (sub)system, or how it works.

In this application, we attempt to solve this design pattern selection problem by leveraging the learnt word and

design pattern vectors. Comparing to the previous task, i.e., design pattern tag recommendation, design pattern selection is usually a more challengeable task. In the previous task, a post may involve explicit characteristics of design patterns, e.g., design pattern names. However, in this task, the description of the design problem cannot contain such information as the suitable design pattern(s) is assumed to be unknown. The semantic meaning of the description should be explored and it should match the application scenarios of the selected design pattern(s).

7.1 General Method of Design Pattern Selection

In this part, we introduce the general framework of design pattern selection in the existing studies.

The existing design pattern selection approaches usually use the problem definition of a design pattern as the oracle for design pattern selection [8], [54]. The problem definition describes what problems the design pattern solves and where the design pattern can be applied. For example, in the GoF book, the problem definition contains the intent, motivation, and applicability Sections [8]. Given a design problem description and a collection of design patterns, the design pattern selection procedure can be detailed in the following three phases [8], [54].

Vectorizing the Documents. The documents, i.e., the design problem description and the problem definitions of design patterns, are preprocessed and vectorized by leveraging the vector space model, in which each document is presented as a feature vector and each feature indicates the weight of a word in the document.

Determining the Design Pattern Class. This phase aims to preliminarily find a set of design patterns that are likely to be right for the design problem. It is motivated by the expert classification of design patterns. For example, the 23 design patterns in GoF are divided into three classes, i.e., Creational Patterns, Structural Patterns, and Behavioral Patterns [2], and each class focuses on one type of design problems. Therefore, the goal is to determine the most suitable design pattern class for the design problem. With this phase, the design pattern selection process can leverage the expert classification information besides the similarity between the design problem and the oracle of a design pattern. Hence, this phase is a reinforcement for the similarity-based selection and the accuracy is expected to be improved.

To determine the design pattern class, text categorization methods are applied to these vectorized text documents. For example, [8] leverages supervised learning methods to build a classifier for textual descriptions based on the expert classes of design patterns. Then the design problem description is classified into a class by the classifier and the design patterns in this class are delivered to the next phase. Similarly, [54] uses clustering methods to group the problem definitions of design patterns and the design problem description into multiple clusters. This partition may be not consistent with the expert classification, but the numbers of classes (or clusters) in the two partitions are equal. The design patterns whose problem definitions are in the same cluster with the design problem description are retained for further selection.

Suggesting the Design Pattern(s). With the determined class of design patterns, the appropriate design pattern(s) is further suggested based on the similarities between the

design problem description and the problem definitions of design patterns. Concretely, the i th design pattern in the determined class is suggested if

$$\begin{cases} |S_i| > \theta_1 \\ |S_i - S_{max}| \leq \theta_2 \end{cases}, \quad (12)$$

where S_i is the similarity between the problem definition of the i th design pattern and the design problem description, S_{max} is the maximum among the similarity S_j corresponding to each design pattern in the determined class, and θ_1 and θ_2 are thresholds that should be specified manually. We note that more than one design patterns may be selected finally. The result relies on the values of the thresholds.

7.2 Refined Design Pattern Selection Method Based on DPWord2Vec

With the learnt design pattern and word vectors, we show how to refine the existing design pattern selection method.

As to the depictions above, the design pattern selection method depends on the expert classification of design patterns. However, this classification may involve inconsistencies and anomalies [8]. In other words, the classification may not be fully reflected by the problem definitions of the design patterns. As a result, the determined class may be unreliable. Therefore, we modify the second phase, i.e., Determining the Design Pattern Class, by leveraging the learned word and design pattern vectors to refine the design pattern selection method.

There are three steps for the modified phase:

- 1) Preprocess and tokenize the design problem description following the procedures in Section 3.2.
- 2) For each design pattern candidate, calculate the similarity between the design pattern and the design problem description as Formula (8).
- 3) Perform k-means clustering [55] on the design pattern candidates to group them into the “relevant” class and “irrelevant” class based on their similarities with the design problem description. The initial centroids of the two clusters are the maximum and minimum of the similarities, respectively. The “relevant” class is considered as the candidate design pattern class for the design problem.

This new phase doesn't use any information of the expert classification but leverages the relatedness between the design problem and design patterns inferred from the word and design pattern vectors. The design patterns with very weak relatedness to the design problem are unlikely to be the appropriate ones and eliminated, the rests are retained for further selection. Except for the second phase, the first and third phases of the method keep unchanged.

7.3 Evaluation of the DPWord2Vec-Based Method

7.3.1 Motivation

To investigate whether the refined method based on DPWord2Vec is effective, we set up this evaluation.

7.3.2 Approach

We compare the refined method based on DPWord2Vec against the existing ones on design pattern selection

benchmarks. In the following parts, we depict the benchmarks, the methods for comparison, the evaluation metrics, and the settings of all the methods, respectively.

Benchmarks. The benchmarks we use are the same as those used in [54], which involve 80 design problems and three design pattern collections, namely GoF [2], Security [56], and Douglass [57]. The GoF collection includes 23 object-oriented design patterns which are divided into three classes. The Security collection includes 46 design patterns used in integrating security systems and presented in eight classes. There are 34 real-time system relevant design patterns in the Douglass collection and they have been divided into five classes. The numbers of design problems corresponding to the three collections are 30, 30, and 20, respectively. For each design problem, only one design pattern in the collection is regarded as correct.¹⁷

Following [54] and [8], for each collection, the evaluation is deployed independently. Only the design patterns in this collection are considered as the original candidates for selection.

State of the Arts. As to our knowledge, there are two studies, [54] and [8], that propose completely automatic design pattern selection methods based on publicly available textual descriptions of design patterns. The methods in these two studies all follow the three-phases framework mentioned above. In this evaluation, we take them for comparison.

Metrics. Following [54] and [8], the design pattern selection methods are evaluated by the Ratio of Correct Detection of Design Pattern (RCDDP) metric, which is calculated as

$$RCDDP = \frac{1}{N} \sum_{i=1}^N \frac{|SDP_i \cap CDP_i|}{|SDP_i|}, \quad (13)$$

where N is the number of design problems for the design pattern collection, CDP_i is the set of correct design pattern (s) to solve the i th design problem (contains only one design pattern in the dataset), and SDP_i is the set of suggested design pattern(s) by the design pattern selection method.

As to the definition above, the RCDDP metric depends on the values of the thresholds θ_1 and θ_2 , as they will determine which design pattern(s) is finally suggested, i.e., SDP_i . It may make the comparisons complicated, since the appropriate values of the thresholds for different design pattern selection methods may be not unified. Actually, our refined method only modifies the phase of determining the design pattern class, but does not deal with the settings of the thresholds. Without losing the reasonability, we leverage another metric for evaluation, namely Mean Reciprocal Rank (MRR) [58], which is not affected by the thresholds. MRR is a standard evaluation metric in information retrieval and used in several software engineering related studies [59]. Specifically,

$$MRR = \frac{1}{N} \sum_{i=1}^N \frac{1}{rank_c^i}, \quad (14)$$

¹⁷ The 80 design problems and the corresponding correct design patterns can be found on <https://github.com/WoodenHeadoo/dpword2vec>.

where $rank_c^i$ denotes the position of the correct design pattern to the i th design problem in the rank based on the similarities in the third phase. The expression $1/rank_c^i$ is called as reciprocal rank. If the correct design pattern is eliminated in the second phase, then the reciprocal rank is 0. As to the definition, the value of MRR is low if most of the correct design patterns are omitted; and high if the irrelevant design patterns ranked before the correct ones are eliminated. Therefore, MRR is able to evaluate the candidate design pattern class produced in the second phase.

Settings of the Methods. The methods in [54] and [8] are more like frameworks rather than concrete algorithms. That means, the concrete algorithms for each step should be specified according to the realities. Therefore, we unify the settings for all methods and leverage the moderate ones that perform best in the most cases according to the results in [54] and [8].

Specifically, the TF-IDF technique is used for the vectorization of the documents in the first phase. In the second phase, the improved global feature selection scheme [60] is used to reduce the dimension of the document vectors. The support vector machine [55] classification algorithm and fuzzy c-means clustering [61] algorithm are leveraged to determine the candidate design pattern class for the method in [8] and the method in [54], respectively. The number of classes (clusters) is consistent with that of the expert classification in each design pattern collection. In the third phase, the cosine similarity is applied to measure the correlation between the vectorized problem definitions of design patterns and design problem descriptions.

For the refined method, the TF-IDF technique and cosine similarity are also used in the first and third phases, respectively. But the second phase is replaced by the modified one.

According to [54] and [8], the effective values of the thresholds and the number of features (dimension of the document vectors after feature selection) rely on the design pattern collections. Hence, we attempt to find the most suitable settings for each collection and report the optimal results. For the methods in [54] and [8], we try various feature numbers from 50 to the vocabulary size at an interval of 50 and the best one in terms of MRR is recorded. Then, for each method, we find the highest value of RCDDP by traversing all the combinations of θ_1 and θ_2 from the range $\{0, 0.1, 0.2, \dots, 1.0\}$ and the range $\{0, 0.01, 0.02, \dots, 0.10\}$ [8], respectively.

7.3.3 Results

The metric values and the corresponding parameter settings are displayed in Table 5. As shown in the table, the refined method achieves the best performance on all three collections. Averaging across the three collections, the refined method outperforms the method in [54] (M1) by 6.3 and 6.5 percent in terms of RCDDP and MRR, respectively. The performance of the method in [8] (M2) is overall unsatisfactory. For example, the refined method improves M2 by over 70 percent in terms of the mean value of MRR.

The possible reason for the bad results of M2 is that too many correct design patterns are eliminated when determining the design pattern class. To show this observation, for each method on each collection, we calculate the ratio of

TABLE 5
The Results and Parameter Settings on the Design Pattern Selection Task

Gof					
Algorithm	RCDDP	MRR	(θ_1, θ_2)	# Features	EER
M1	0.5333	0.6368	(0, 0)	950	16.67%
M2	0.3333	0.3417	(0, 0)	950	63.33%
Refined	0.5667	0.6806	(0, 0)	-	13.33%
Security					
Algorithm	RCDDP	MRR	(θ_1, θ_2)	# Features	EER
M1	0.8000	0.8278	(0, 0)	900	13.33%
M2	0.4333	0.4500	(0, 0)	200	53.33%
Refined	0.8667	0.9111	(0, 0)	-	3.33%
Douglass					
Algorithm	RCDDP	MRR	(θ_1, θ_2)	# Features	EER
M1	0.6000	0.6917	(0, 0)	600	15.00%
M2	0.4583	0.5542	(0, 0.08)	650	30.00%
Refined	0.6225	0.7058	(0, 0.1)	-	5.00%

M1 = the method in [54], *M2* = the method in [8], *Refined* = the refined method based on DPWord2Vec

cases in which the correct design pattern is eliminated after the second phase. The results are also shown in Table 5, namely Erroneously Eliminating Ratio (EER). From the results, we notice that the EERs of M2 are very high for all the collections. For example, the EER of M2 is 63.33 percent for the Gof collection. That means, the correct design patterns of the 19 among the 30 design problems are mistakenly eliminated. Meanwhile, the EERs of the refined method are the lowest among all the methods on the three collections.

Notably, the performance of the refined method is not much better than that of M1 on the Douglass collection. The MRR values are respectively 0.7058 and 0.6917, which are quite similar. Generally, the quality of the learnt design pattern vectors relies on the design pattern relevant documents. However, in the corpus C , the number of documents relevant to each design pattern in the Douglass collection seems to be too few. Counting all the 20 design patterns mentioned in the benchmarks, nine of them relate to less than 10 documents each, eight design patterns occupy 10 to 49 documents each, and each of the other three ones involves 50 to 73 documents. It could be the reason for the nonsignificant improvement in the Douglass collection.

The main difference among M1, M2, and the refined method is the way of determining the candidate design pattern class. M2 chooses one class of the expert classification as the candidate class but this way does not work well. M1 does not completely follow the expert classification but use it during the feature selection. The performance of M1 is much better than that of M2, but not as good as that of the refined method. It implies that the way by leveraging the learnt word and design pattern vectors is more appropriate to find a candidate set of design patterns than the way by using the expert classification.

7.3.4 Conclusion

The refined method based on DPWord2Vec is superior to the methods in [54] and [8] on the benchmarks. Therefore, DPWord2Vec contributes to accomplish the task of design pattern selection.

8 THREATS TO VALIDITY

8.1 Internal Validity

There are several threats to internal validity of our work.

First, the size of the corpus may restrict the effectiveness of DPWord2Vec. The corpus in this paper is relatively small comparing with those used in other word embedding methods [11], [13]. This may influence the quality of the learnt word and design pattern vectors. However, we believe this problem would be alleviated as more design pattern relevant documents could be extracted in the coming future due to the popularity of programming forums. Second, only the default values of the parameters are used to build the word and design pattern vectors. However, the empirical study shows that the performance of DPWord2Vec is not very sensitive to the settings of the main parameters, i.e., the context window size for words and the dimension of vectors. Third, the human judgment process of the dp-word pairs may contain uncertainties, since it may be not easy to judge whether a design pattern and a word is related sometimes. However, such procedures are common practice in similarity tasks of various domains [18], [23], [24], [25]. We try to mitigate the uncertainties by involving a new label, i.e., “somewhat related”. Moreover, the Fleiss’ Kappa measure shows that the annotators reach a substantial agreement. Finally, the way of determining design pattern relevant posts for constructing the crowdsourced corpus is not completely precise. This factor is in the scope of our previous study. We have performed a validation to ensure the reliability of the results [12].

8.2 External Validity

The threats to external validity relate to the generalization of DPWord2Vec. We sample 2,000 dp-word pairs to evaluate DPWord2Vec in terms of dp-word similarity and employ two applications to evaluate DPWord2Vec in terms of design pattern - words (document) similarity. It is unclear whether DPWord2Vec still works well on other tasks. More datasets or applications will be investigated to reduce this threat in the future.

9 RELATED WORK

9.1 Word Embedding for Software Artifacts

Similar to our work, numbers of studies leverage word embedding methods on software artifacts to aid in software engineering relevant tasks.

Some studies focus on mapping APIs into vector space. Nguyen *et al.* propose API2Vec that learns API vectors based on API usage sequences extracted from code corpora [62]. Similarly, Li *et al.* embed natural language words and APIs at the same time by leveraging both API sequences and the method comments [40]. To establish API mappings between third-party libraries, Chen *et al.* present an unsupervised deep learning-based approach to map both API usage semantics and API description semantics into vectors [63].

Meanwhile, some studies aim to learn the representations of programs. Alon *et al.* produce general representations of programs based on the paths in abstract syntax trees [64]. Henkel *et al.* represent programs as abstractions of traces obtained from symbolic execution and learn the vectors of the abstractions using word embedding [22].

Piech *et al.* introduce a neural network method to learn the feature embedding of a whole program and give automatic feedback based on the representation [65].

Moreover, some studies directly use word embedding methods on software-related documents to support some other tasks. Ye *et al.* train the word embeddings on API relevant documents and aggregate them to estimate semantic similarities between documents [59]. Calefato *et al.* exploit word embedding on Stack Overflow posts to help to analyse the sentiments of developers [66]. Guo *et al.* attempt to generate trace links among software artifacts by utilizing word embedding and recurrent neural network trained on clean text from related domain documents [67].

Different from these studies, our work concentrates on associating natural language words and design patterns by embedding them into one vector space. To the best of our knowledge, no previous studies have ever considered the general relatedness between words and design patterns.

9.2 Tag Recommendation in Software Information Sites

In the first application, we apply DPWord2Vec to the design pattern tag recommendation task. There exist a series of tag recommendation methods specified for software information sites.

To automatically recommend tags in software information sites, Xia *et al.* propose TagCombine which ranks each tag candidate by integrating three ranking component [47]. After that, EnTagRec is proposed and outperforms TagCombine on four software information sites in terms of Recall [48]. To adopt tag recommendation methods in large-scale software information sites, Zhou *et al.* propose a more scalable approach called TagMulRec [49]. TagMulRec outperforms EnTagRec in terms of Precision and F1-score on four software information sites. Then Wang *et al.* enhance EnTagRec to a new version, namely EnTagRec++, by leveraging the information of users of software information sites [50]. EnTagRec++ improves TagCombine by over 10 percent on five software information sites in terms of Recall. Recently, Liu *et al.* propose FastTagRec which recommends tags using neural network-based classification [51]. An evaluation on ten software information sites shows FastTagRec is more accurate than TagMulRec.

Most of these methods can also be used in the design pattern tag recommendation task. In the evaluation, the DPWord2Vec-based design pattern tag recommendation method is compared against the state-of-the-art ones, i.e., FastTagRec and TagMulRec, to show its effectiveness.

9.3 Design Pattern Selection Based on Text

The related work for the second application is about design pattern selection. We focus on the methods leveraging textual descriptions here. These works can be roughly categorized into two types.

The first type is based on design pattern use cases and recommend design patterns by exploring the most similar use cases to the current design problem. Gomes *et al.* propose a case-based reasoning approach for design pattern selection and index cases by using WordNet [68]. Similarly, Muangon *et al.* present a design pattern searching model by

combining case-based reasoning and formal concept analysis techniques [10]. Bouassida *et al.* integrate case search and questionnaire strategy to create an interactive design pattern selection method [69]. These approaches are based on the assumption that there exists a case library. However, few such libraries are publicly available.

The second type is based on general textual descriptions of design patterns. Palma *et al.* provide an expert system for design pattern recommendation and parses design pattern descriptions to formulate questionnaires for designers [9]. In [70], Pavlič *et al.* document the knowledge of design patterns by building an ontology for design pattern advisement. The studies [54] and [8] automate the process and only utilize the original descriptions in design pattern books for design pattern selection.

In this application, we follow the automatic design pattern selection framework in [54] and [8] but refine the design pattern class determining phase by DPWord2Vec. The refined method outperforms the methods in [54] and [8] on the benchmarks.

10 CONCLUSION

In this work, we propose DPWord2Vec, a framework that maps both natural language words and design patterns into one vector space. With the word and design pattern vectors, each design pattern is associated with English natural language. DPWord2Vec leverages the word embedding method to learn the word and design pattern vector representations based on two built corpora with our redefined context windows. An evaluation on a dp-word pair dataset shows that DPWord2Vec is more effective than the baseline methods in measuring the dp-word similarity. Moreover, two design pattern relevant applications are leveraged to investigate the usefulness of DPWord2Vec. The experimental results indicate that DPWord2Vec can outperform the state-of-the-art algorithms on the specific tasks.

In the future, on one hand, we will extract more design pattern relevant documents from other sources to enrich the corpora; on the other hand, we will attempt to apply DPWord2Vec to more design pattern relevant tasks. Moreover, it is also worth investigating the effectiveness of DPWord2Vec on the corpora of non-English languages.

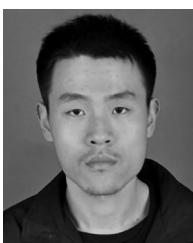
ACKNOWLEDGMENT

This work was supported by the National Key Research and Development Program of China under Grant No. 2018YFB1003903, and the National Natural Science Foundation of China under Grant Nos. 61722202, 61772107, and 61751210.

REFERENCES

- [1] C. Alexander, *A Pattern Language: Towns, Buildings, Construction*. New York, NY, USA: Oxford Univ. Press, 1977.
- [2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, USA: Addison-Wesley, 1995.
- [3] C. Zhang and D. Budgen, "What do we know about the effectiveness of software design patterns?" *IEEE Trans. Softw. Eng.*, vol. 38, no. 5, pp. 1213–1231, Sep./Oct. 2012.
- [4] H. Zhu and I. Bayley, "On the composability of design patterns," *IEEE Trans. Softw. Eng.*, vol. 41, no. 11, pp. 1138–1152, Nov. 2015.
- [5] A. D. Lucia, V. Deufemia, C. Gravino, and M. Risi, "Detecting the behavior of design patterns through model checking and dynamic analysis," *ACM Trans. Softw. Eng. Methodol.*, vol. 26, no. 4, pp. 1–41, Feb. 2018.
- [6] D. Faitelson and S. Tyszberowicz, "UML diagram refinement (focusing on class and use case diagrams)," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 735–745.
- [7] S. Henninger and V. Corrêa, "Software pattern communities: Current practices and challenges," in *Proc. 14th Conf. Pattern Lang. Programs*, 2007, pp. 14:1–14:19.
- [8] S. M. H. Hasheminejad and S. Jalili, "Design patterns selection: An automatic two-phase method," *J. Syst. Softw.*, vol. 85, no. 2, pp. 408–424, 2012.
- [9] F. Palma, H. Farzin, Y.-G. Guéhéneuc, and N. Moha, "Recommendation system for design patterns in software development: An DPR overview," in *Proc. 3rd Int. Workshop Recommendation Syst. Softw. Eng.*, 2012, pp. 1–5.
- [10] W. Muangon and S. Intakosum, "Case-based reasoning for design patterns searching system," *Int. J. Comput. Appl.*, vol. 70, no. 26, pp. 16–24, May 2013.
- [11] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Proc. 26th Int. Conf. Neural Inf. Process. Syst.*, 2013, pp. 3111–3119.
- [12] H. Jiang, D. Liu, X. Chen, H. Liu, and H. Mei, "How are design patterns concerned by developers?" in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 232–233.
- [13] J. Pennington, R. Socher, and C. D. Manning, "GloVe: Global vectors for word representation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2014, pp. 1532–1543.
- [14] C. Chen, Z. Xing, and X. Wang, "Unsupervised software-specific morphological forms inference from informal discussions," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 450–461.
- [15] D. Ye, Z. Xing, C. Y. Foo, J. Li, and N. Kapre, "Learning to extract API mentions from informal natural language discussions," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2016, pp. 389–399.
- [16] B. Dit, L. Guerrrouj, D. Poshyvanyk, and G. Antoniol, "Can better identifier splitting techniques help feature location?" in *Proc. IEEE 19th Int. Conf. Program Comprehension*, 2011, pp. 11–20.
- [17] M. Porter, "An algorithm for suffix stripping," *Program*, vol. 14, no. 3, pp. 130–137, 1980.
- [18] M.-T. Luong, R. Socher, and C. D. Manning, "Better word representations with recursive neural networks for morphology," in *Proc. 7th Conf. Comput. Natural Lang. Learn.*, 2013, pp. 104–113.
- [19] R. Soricut and F. Och, "Unsupervised morphology induction using word embeddings," in *Proc. Conf. North Amer. Chapter Assoc. Comput. Linguistics: Hum. Lang. Technol.*, 2015, pp. 1627–1637.
- [20] X. L. Yang, D. Lo, X. Xia, Z. Y. Wan, and J. L. Sun, "What security questions do developers ask? A large-scale study of stack overflow posts," *J. Comput. Sci. Technol.*, vol. 31, no. 5, pp. 910–924, Sep. 2016.
- [21] D. M. W. Powers, "Applications and explanations of zipf's law," in *Proc. Joint Conf. New Methods Lang. Process. Comput. Natural Lang. Learn.*, 1998, pp. 151–160.
- [22] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps, "Code vectors: Understanding programs through embedded abstracted symbolic traces," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 163–174.
- [23] E. H. Huang, R. Socher, C. D. Manning, and A. Y. Ng, "Improving word representations via global context and multiple word prototypes," in *Proc. 50th Annu. Meeting Assoc. Comput. Linguistics*, 2012, pp. 873–882.
- [24] Y. Tian, D. Lo, and J. Lawall, "Automated construction of a software-specific word similarity database," in *Proc. IEEE Conf. Softw. Maintenance Reeng. Reverse Eng.*, 2014, pp. 44–53.
- [25] A. Mahmoud and G. Bradshaw, "Estimating semantic relatedness in source code," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, pp. 10:1–10:35, Dec. 2015.
- [26] S. Deerwester, S. T. Dumais, G. W. Furnas, T. K. Landauer, and R. Harshman, "Indexing by latent semantic analysis," *J. Amer. Soc. Inf. Sci.*, vol. 41, no. 6, pp. 391–407, 1990.
- [27] C. Rosen and E. Shihab, "What are mobile developers asking about? A large scale study using stack overflow," *Empir. Softw. Eng.*, vol. 21, no. 3, pp. 1192–1223, Jun. 2016.

- [28] A. Barua, S. W. Thomas, and A. E. Hassan, "What are developers talking about? An analysis of topics and trends in stack overflow," *Empir. Softw. Eng.*, vol. 19, no. 3, pp. 619–654, Jun. 2014.
- [29] A. Panichella, B. Dit, R. Oliveto, M. Di Penta, D. Poshyvanyk, and A. De Lucia, "How to effectively use topic models for software engineering tasks? An approach based on genetic algorithms," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 522–531.
- [30] T. L. Griffiths and M. Steyvers, "Finding scientific topics," *Proc. Nat. Acad. Sci. USA*, vol. 101, no. suppl 1, pp. 5228–5235, 2004.
- [31] K. W. Church and P. Hanks, "Word association norms, mutual information, and lexicography," *Comput. Linguistics*, vol. 16, no. 1, pp. 22–29, Mar. 1990.
- [32] R. L. Cilibrasi and P. M. B. Vitanyi, "The Google similarity distance," *IEEE Trans. Knowl. Data Eng.*, vol. 19, no. 3, pp. 370–383, Mar. 2007.
- [33] A. Hotho, A. Nürnberger, and G. Paass, "A brief survey of text mining," *GLDV J. Comput. Linguistics Lang. Technol.*, vol. 20, pp. 19–62, Jan. 2005.
- [34] L. Nie, H. Jiang, Z. Ren, Z. Sun, and X. Li, "Query expansion based on crowd knowledge for code search," *IEEE Trans. Services Comput.*, vol. 9, no. 5, pp. 771–783, Sep./Oct. 2016.
- [35] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 664–675.
- [36] C. Mcmillan, D. Poshyvanyk, M. Grechanik, Q. Xie, and C. Fu, "Portfolio: Searching for relevant functions and their usages in millions of lines of code," *ACM Trans. Softw. Eng. Methodol.*, vol. 22, no. 4, pp. 37:1–37:30, Oct. 2013.
- [37] Y. Dodge, *The Concise Encyclopedia of Statistics*. New York, NY, USA: Springer, 2010.
- [38] S. Siegel, *Non-Parametric Statistics for the Behavioral Sciences*. New York, NY, USA: McGraw-Hill, 1956.
- [39] M. Fowler, *Patterns of Enterprise Application Architecture*. Reading, MA, USA: Addison-Wesley, 2002.
- [40] X. Li, H. Jiang, Y. Kamei, and X. Chen, "Bridging semantic gaps between natural languages and APIs with word embedding," *IEEE Trans. Softw. Eng.*, to be published, doi: [10.1109/TSE.2018.2876006](https://doi.org/10.1109/TSE.2018.2876006).
- [41] Q. Le and T. Mikolov, "Distributed representations of sentences and documents," in *Proc. 31st Int. Conf. Mach. Learn.*, 2014, pp. 1188–1196.
- [42] C. Zhou, C. Sun, Z. Liu, and F. C. M. Lau, "Category enhanced word embedding," 2015, *arXiv:1511.08629*.
- [43] T. Nguyen et al., "Complementing global and local contexts in representing API descriptions to improve API retrieval tasks," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 551–562.
- [44] J. Cohen, *Statistical Power Analysis for the Behavioral Sciences*, 2nd ed. Mahwah, NJ, USA: Lawrence Erlbaum Associates, 1988.
- [45] S. S. Sawilowsky, "New effect size rules of thumb," *J. Modern Appl. Stat. Methods*, vol. 8, no. 2, pp. 597–599, 2009.
- [46] J. M. Al-Kofahi, A. Tamrawi, T. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "Fuzzy set approach for automatic tagging in evolving software," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2010, pp. 1–10.
- [47] X. Xia, D. Lo, X. Wang, and B. Zhou, "Tag recommendation in software information sites," in *Proc. 10th IEEE Work. Conf. Mining Softw. Repositories*, 2013, pp. 287–296.
- [48] S. Wang, D. Lo, B. Vasilescu, and A. Serebrenik, "EnTagRec: An enhanced tag recommendation system for software information sites," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 291–300.
- [49] P. Zhou, J. Liu, Z. Yang, and G. Zhou, "Scalable tag recommendation for software information sites," in *Proc. 24th Int. Conf. Softw. Anal. Evol. Reeng.*, 2017, pp. 272–282.
- [50] S. Wang, D. Lo, B. Vasilescu, and A. Serebrenik, "EnTagRec++: An enhanced tag recommendation system for software information sites," *Empir. Softw. Eng.*, vol. 23, no. 2, pp. 800–832, Apr. 2018.
- [51] J. Liu, P. Zhou, Z. Yang, X. Liu, and J. Grundy, "FastTagRec: Fast tag recommendation for software information sites," *Automated Softw. Eng.*, vol. 25, no. 4, pp. 675–701, Dec. 2018.
- [52] R. Mihalcea, C. Corley, and C. Strapparava, "Corpus-based and knowledge-based measures of text semantic similarity," in *Proc. 21st Nat. Conf. Artif. Intell.*, 2006, pp. 775–780.
- [53] D.-K. Kim and C. E. Khawand, "An approach to precisely specifying the problem domain of design patterns," *J. Vis. Lang. Comput.*, vol. 18, no. 6, pp. 560–591, 2007.
- [54] S. Hussain, J. Keung, and A. A. Khan, "Software design patterns classification and selection using text categorization approach," *Appl. Soft Comput.*, vol. 58, pp. 225–244, 2017.
- [55] J. Friedman, T. Hastie, and R. Tibshirani, *The Elements of Statistical Learning*. New York, NY, USA: Springer, 2001.
- [56] M. Schumacher, E. Fernandez-Buglioni, D. Hybertson, F. Buschmann, and P. Sommerlad, *Security Patterns: Integrating Security and Systems Engineering*. Hoboken, NJ, USA: Wiley, 2006.
- [57] B. P. Douglass, *Real-Time Design Patterns: Robust Scalable Architecture for Real-Time Systems*. Reading, MA, USA: Addison-Wesley Professional, 2003.
- [58] E. M. Voorhees and D. M. Tice, "The TREC-8 question answering track report," in *Proc. 8th Text Retrieval Conf.*, 1999, pp. 77–82.
- [59] X. Ye, H. Shen, X. Ma, R. Bunescu, and C. Liu, "From word embeddings to document similarities for improved information retrieval in software engineering," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 404–415.
- [60] A. K. Uysal, "An improved global feature selection scheme for text classification," *Expert Syst. Appl.*, vol. 43, pp. 82–92, 2016.
- [61] J. C. Bezdek, *Pattern Recognition With Fuzzy Objective Function Algorithms*. Berlin, Germany: Springer, 2013.
- [62] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring API embedding for API usages and applications," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 438–449.
- [63] C. Chen, Z. Xing, Y. Liu, and K. L. X. Ong, "Mining likely analogical APIs across third-party libraries via large-scale unsupervised API semantics embedding," *IEEE Trans. Softw. Eng.*, to be published, doi: [10.1109/TSE.2019.2896123](https://doi.org/10.1109/TSE.2019.2896123).
- [64] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "A general path-based representation for predicting program properties," in *Proc. 39th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2018, pp. 404–419.
- [65] C. Piech, J. Huang, A. Nguyen, M. Phulsuksombati, M. Sahami, and L. Guibas, "Learning program embeddings to propagate feedback on student code," in *Proc. 32nd Int. Conf. Mach. Learn.*, 2015, pp. 1093–1102.
- [66] F. Calefato, F. Lanobile, F. Maiorano, and N. Novielli, "Sentiment polarity detection for software development," *Empir. Softw. Eng.*, vol. 23, no. 3, pp. 1352–1382, Jun. 2018.
- [67] J. Guo, J. Cheng, and J. Cleland-Huang, "Semantically enhanced software traceability using deep learning techniques," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 3–14.
- [68] P. Gomes et al., "Using CBR for automation of software design patterns," in *Proc. Eur. Conf. Case-Based Reasoning*, 2002, pp. 534–548.
- [69] N. Bouassida, S. Jamoussi, A. Msaed, and H. Ben-Abdallah, "An interactive design pattern selection method," *J. Universal Comput. Sci.*, vol. 21, no. 13, pp. 1746–1766, Jan. 2015.
- [70] L. Pavlič, V. Podgorelec, and M. Hericko, "A question-based design pattern advisement approach," *Comput. Sci. Inf. Syst.*, vol. 11, no. 2, pp. 645–664, Jun. 2014.



Dong Liu received the MS degree in computer science and technology from the Hebei University of Technology, Tianjin, China, in 2016. He is currently working toward the PhD degree at the Dalian University of Technology, Dalian, China. His current research interests include mining software repositories and data-driven methods in software engineering.



He Jiang (Member, IEEE) received the PhD degree in computer science from the University of Science and Technology of China, Hefei, China, in 2004. He is currently a professor with the Dalian University of Technology, and an adjunct professor with the Beijing Institute of Technology. His current research interests include intelligent software engineering, mining software repositories, and compilers.



Xiaochen Li received the PhD degree in software engineering from the Dalian University of Technology, Dalian, China, in 2019. He is currently a research associate with the Software Verification and Validation Lab, University of Luxembourg. His current research interests include mining software repositories, open-source software engineering, and software semantic analysis.



Lei Qiao received the PhD degree in computer science from the University of Science and Technology of China, Hefei, China, in 2007. He is currently a senior researcher with the Beijing Institute of Control Engineering. His current research interests include embedded operating system design and formal verification.



Zhilei Ren received the BS degree in software engineering and the PhD degree in computational mathematics from the Dalian University of Technology, Dalian, China, in 2007 and 2013, respectively. He is currently an associate professor with the Dalian University of Technology. His current research interests include evolutionary computation, automatic algorithm configuration, and mining software repositories.



Zuohua Ding (Member, IEEE) received the MS degree in computer science and the PhD degree in mathematics from the University of South Florida, Tampa, Florida, in 1996 and 1998, respectively. He is currently a professor and the director with the Laboratory of Intelligent Computing and Software Engineering, Zhejiang Sci-Tech University. His current research interests include system modeling, software reliability prediction, intelligent software systems, and service robots.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

Heuristic and Neural Network Based Prediction of Project-Specific API Member Access

Lin Jiang^{ID}, Hui Liu^{ID}, He Jiang^{ID}, Member, IEEE, Lu Zhang^{ID}, and Hong Mei^{ID}, Fellow, IEEE

Abstract—Code completion is to predict the rest of a statement a developer is typing. Although advanced code completion approaches have greatly improved the accuracy of code completion in modern IDEs, it remains challenging to predict project-specific API method invocations or field accesses because little knowledge about such elements could be learned in advance. To this end, in this paper we propose an accurate approach called HeeNAMA to suggesting the next project-specific API member access. HeeNAMA focuses on a specific but common case of code completion: suggesting the following member access whenever a project-specific API instance is followed by a dot on the right hand side of an assignment. By focusing on such a specific case, HeeNAMA can take full advantages of the context of the code completion, including the type of the left hand side expression of the assignment, the identifier on the left hand side, the type of the base instance, and similar assignments typed in before. All such information together enables highly accurate code completion. Given an incomplete assignment, HeeNAMA generates the initial candidate set according to the type of the base instance, and excludes those candidates that are not type compatible with the left hand side of the assignment. If the enclosing project contains assignments highly similar to the incomplete assignment, it makes suggestions based on such assignments. Otherwise, it selects the one from the initial candidate set that has the greatest lexical similarity with the left hand side of the assignment. Finally, it employs a neural network to filter out risky predictions, which guarantees high precision. Evaluation results on open-source applications suggest that compared to the state-of-the-art approaches and the state-of-the-practice tools HeeNAMA improves precision and recall by 70.68 and 25.23 percent, respectively.

Index Terms—Code completion, non-API, deep learning, heuristic, LSTM

1 INTRODUCTION

THE purpose of code completion is to predict the rest (or a part of the rest) of a statement a developer is typing. Code completion feature provided by modern Integrated Development Environments (IDEs) plays an important role in software development process [1], [2]. The usage data collected from 41 Java software developers suggests that code completion is one of the most commonly used commands [3]. It is executed as frequently as the common editing commands, e.g., *delete*, *save*, *paste* and *copy*.

Code completion is widely and frequently employed for several reasons. First, code completion lightens the amount of memory work required of developers [4]. Second, powerful and accurate code completion tools encourage developers to choose longer and more descriptive identifier names because with code completion tools developers do not have to type in all characters of such names [5]. Third, it helps to reduce the number of characters that should be typed in manually [6]. The benefit of the reduction in manually typed characters is twofold. On one side, it speeds up coding. On

the other side, it reduces misspelling. When developers type in source code, especially long identifiers, it is likely that typos are introduced. Because code completion tools greatly reduce the number of characters that should be typed in manually, the likelihood of introducing typos is reduced significantly as well.

In this paper, we focus on a common type of code completion: method invocation and field access completion (other forms of code completion include word completion [7], expression completion [8], method argument completion [9] and statement completion [10]). To improve the accuracy of code completion, a number of powerful code completion approaches have been proposed. The first category of such approaches is based on usage pattern mining algorithms [11], e.g., frequent item mining [2], [12], [13], frequent subsequence mining [14] and frequent subgraph mining [15]. These approaches discover code patterns from source code repositories and make code suggestions by matching the given source code against such patterns.

The second category of code completion approaches is based on statistical language models [16]. Such approaches take the assumption that programming languages are somewhat similar to natural languages, and thus the widely used natural language models could be applied to programming languages as well [7]. The most commonly employed language models include N-gram models [17], [18] and deep neural network based language models [19], [20]. An advantage of such language-model based approaches is that they are generic, and thus they can predict all kinds of tokens (e.g., the next character, identifier, member access or statement).

- Lin Jiang, Hui Liu, He Jiang, and Hong Mei are with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China. E-mail: {fjianglin17, lihuhi08, meihong}@bit.edu.cn, hejiang@ieee.org.
- Lu Zhang is with the Key Laboratory of High Confidence Software Technologies, Ministry of Education, Beijing 100871, China. E-mail: zhanglu@sei.pku.edu.cn.

Manuscript received 11 Dec. 2019; revised 8 Aug. 2020; accepted 11 Aug. 2020. Date of publication 19 Aug. 2020; date of current version 18 Apr. 2022.

(Corresponding author: Hui Liu.)

Recommended for acceptance by M. Nagappan.

Digital Object Identifier no. 10.1109/TSE.2020.3017794

TABLE 1
Subject Applications

Applications	Domain	Version	LOC
Ant	Software Build	1.10.1	270,028
Batik	SVG Toolkit	1.9	361,429
Cassandra	Database Management	3.11.1	592,595
Log4J	Log Management	2.10.0	236,825
Lucene-solr	Search Engine	7.2.0	1,591,582
Maven2	Software Build	2.2.1	91,760
Maven3	Software Build	3.5.2	169,988
Xalan-J	XSLT Processing	2.7.2	352,787
Xerces	XML parser	2.11.0	216,907

Although such advanced code completion approaches have greatly improved the accuracy of code completion in modern IDEs [12], [21], it remains challenging to predict project-specific API method invocations and project-specific API field accesses [22]. In this paper, we call methods and fields defined within the project under development as project-specific API methods and project-specific API fields, respectively. We also call method invocations and field accesses as member accesses for short in the rest of this paper. According to our empirical study on nine well-known open-source Java applications (as introduced in Table 1), public API member accesses account for less than half ($39\% = 339,866 / 861,618$) of the member accesses in source code, and the majority (more than 60 percent) is project-specific API member accesses. An empirical study conducted recently [22], however, suggests that existing approaches are often significantly less accurate in predicting project-specific API method accesses (what they call intra-project API completions) than public API member accesses. Our evaluation in Section 4 also confirms their conclusion: the accuracy of such approaches in predicting project-specific API member accesses deserves significant improvement.

To this end, in this paper we propose HeeNAMA, a heuristic and neural network based approach to predict project-specific API member accesses. It is challenging to predict project-specific API member accesses in general because little knowledge about such elements could be learned in advance. Consequently, in this paper we focus on a specific but common case of code completion: suggesting the following method call or field access whenever a project-specific API instance is followed by a dot (.) on the right hand side of an assignment (we call them member access on RHS for short). For example, once the developer types in “*String name = person.*”, HeeNAMA would suggest “*getName()*” as the next token. We reuse nine open-source Java applications from previous code completion research [7], [18], [23], [24] to conduct our empirical study. They cover various domains such as software build, database management, and search engine. The size (LOC) of subject applications varies from 91,760 to 1,591,582. According to the empirical study, such cases of code completion (i.e., project-specific API member access on RHS) are common, and on average a single project contains 11,522 such cases. By focusing on such a specific case, HeeNAMA can take full advantages of the context of the code completion, including the type of the left hand side expression of the assignment, the identifier on the left hand side, the type of the base instance, and similar assignments

typed in before. All such information together enables highly accurate code completion.

Given an incomplete assignment, HeeNAMA works as follows to predict the next member access. First, it generates the initial candidate set according to the type of the base instance. Mandelin *et al.* [10] and Gvero *et al.* [8] have proved that such type information is helpful in code completion. Second, it looks for highly similar assignments within the project under development. If successful, it would make suggestions based on the retrieved samples. Third, it filters out candidates that are type incompatible with the left hand side expression of the assignment. After that, it ranks candidates in descending order according to their lexical similarity with the identifier on the left hand side of the assignment. Finally, it leverages a deep neural network to decide whether the top one on the candidate list should be recommended. The evaluation results on nine well-known open-source Java applications suggest that HeeNAMA is more accurate than the state-of-the-art approach as well as the state-of-the-practice tool.

The paper makes the following contributions:

- First, we propose an approach called HeeNAMA to recommending project-specific API member accesses on RHS. HeeNAMA takes full advantages of the context, i.e., the type of the left hand side expression of the assignment, the identifier on the left hand side, the type of the base instance, and similar assignments typed in before. It also leverages a neural network based filter to exclude risky predictions, which significantly improves the precision of HeeNAMA. The combination of heuristics and neural network makes for a neat way of learning to avoid precisely the kinds of mistakes that heuristics make. To the best of our knowledge, HeeNAMA is the first one that is specially designed to predict project-specific API member accesses on RHS.
- Second, we implement HeeNAMA, and evaluate it on nine open-source Java applications. The evaluation results show that HeeNAMA is accurate.

The remainder of this paper is structured as follows. Section 2 presents a short overview of related research. Section 3 proposes our code completion approach. Section 4 presents an evaluation of the proposed approach on nine open-source applications. Section 5 provides conclusions and potential future work.

2 RELATED WORK

2.1 Language Model Based Code Completion

N-gram models are well known in the natural language processing community. They were applied to source code for the first time by Hindle *et al.* [7] when they find the repetitiveness and predictability of source code. Based on N-gram, they estimate the occurrence probabilities for code sequences (at the granularity of token) in code corpus, and predict the next token according to the corresponding occurrence probabilities. Allamanis *et al.* [17] build a giga-token corpus of Java source code from a wide variety of domains to train a n-gram model. The resulting model can successfully deal with token prediction across different project domains. They also find

that employing a large corpus in model training can increase the predictive capability of models. SLAMC [18] strengthens n-grams with semantic information to present token sequences. Such semantic information includes the token roles, data types, scopes, and structural and data dependencies. It also combines the local context with the global technical concerns/functionality into a n-gram based topic model.

Tu *et al.* [23] find that source code has high repetitiveness not only in the global scope but also in the local scope. Based on this finding, they propose a cache language model by enhancing the conventional n-gram model with an efficient caching mechanism that captures the local repetitiveness of source code. They compute the probability of a sequence of tokens based on a global n-gram model (trained with public corpus) and a local n-gram model (trained with source files in the enclosing folder). Based on this cache language model [23], Franks *et al.* develop an Eclipse plug-in CACHECA [21]. It combines the native suggestions made by Eclipse IDE with suggestions made by the cache language model. The evaluation results suggest that the combination leads to higher accuracy.

The latest advance in N-gram based code completion was achieved by Hellendoorn *et al.* [20]. Based on cached language models, they proposed a nested and cached N-gram model to capture the local repetition within a given scope, and to apply it to the nested sub-scopes. The evaluation results suggest that their approach significantly outperforms existing approaches (both statistical language model based approaches and deep learning based approaches).

Advanced neural networks, e.g., RNN [25] and LSTM [26], have been successfully employed to model source code as well. Raychev *et al.* [19] employ RNN for code completion. They first extract sequences of method calls from large code bases, and learn their probabilities with statistical language models, i.e., RNN, N-gram, or a combination of them. Once given a program with holes, they leverage learned probabilities to synthesize sequences of calls for holes. White *et al.* [27] also apply the RNN language model to source code and show its high effectiveness in predicting sequential software tokens.

To address the enormous vocabulary problem in modeling source code with deep neural networks, Karampatsis and Sutton [28] present a new open-vocabulary neural language model for code that is not limited to a fixed vocabulary of identifier names. They employ a segmentation into subword units, i.e., subsequences of tokens chosen based on a compression criterion. Including all single characters as subword units will allow the model to predict all possible tokens, so there is no need for special out-of-vocabulary handling.

Graph-based statistical language models are successfully employed in code completion as well. Nguyen *et al.* [29] introduce GraLan, a graph-based statistical language model to statistically learn API usage (sub)graphs [30] from a source code corpus. Given an observed (sub)graphs that representing the context of code completion, GraLan recommends the next API by computing the appearance probabilities of new usage graphs. SALAD [30], [31] also employs the graph-based model to represent API usage patterns. Given bytecode and source code, SALAD generates a graph-based model for extracting API sequences from such

model. Such API sequences are in turn employed to train a Hidden Markov Model [32] (called HAPI). According to their evaluation, the resulting HAPI is accurate in predicting the next method call.

2.2 Pattern Mining Based Code Completion

It is quite often that a group of related API methods are invoked in some order to accomplish a specific task. By mining code repositories, we may discover such patterns, i.e., the API methods in order [14], [33]. Such patterns, in turn, are employed to recommend the next API method invocation whenever the preceding API method invocations are typed in.

Bruch *et al.* [2] propose three similar intelligent code completion systems that learn API patterns from existing code repositories in different ways. The first system, called FreqCCS, counts API method invocations in code repositories, and recommends the most commonly invoked method as the next API method invocation. The second one, called ArCCS, mines association rules among API method invocations. An example of association rule is “If a new instance of *Text* is created, recommend *setText()*”. ArCCS makes code completions based on such association rules. The last and most advanced system, called BMNCCS, adapts the K-Nearest-Neighbor (KNN) [34] machine learning algorithm to manage API patterns. First, it extracts and encodes the context information (including methods invoked on the same base instance) for each API method invocation in the repository as a binary feature vector. With these feature vectors, it computes the distances between the current context and the API example contexts based on Hamming distance. For API methods associated with the resulting nearest contexts, the approach sorts them according to their frequency in repositories, and the most frequently used one is recommended.

CSCC [12], [35] is another powerful pattern mining based code completion system. The major difference between BMNCCS and CSCC is that the latter takes more context information into usage patterns, i.e., all method calls, Java keywords and type names that appear within the four lines prior to the completion location. To speed up the search for patterns, CSCC employs two distance measures to compute the similarities between the current context and the usage contexts mined from repositories. PBN [13] further extends BMNCCS to tackle the issue of significantly increased model sizes. Unlike BMNCCS that uses a table of binary values to represent usages of different framework types, PBN encodes the same information as a Bayesian network. A key consequence is that PBN allows to merge different patterns and to denote probabilities (instead of boolean existence) for all context information.

MAPO [14] combines frequent sequence mining with clustering to summarize API usage patterns from source files. It mines API usage patterns from open source repositories automatically, and recommends the mined patterns and their associated samples on programmer’s requests. MAPO also provides a recommender that integrates with Eclipse IDE.

GraPacc [15] extends the mining of API usage patterns successfully to higher-order patterns. It represents each pattern as a graph-based model [30] that captures the usage of multiple variables, method calls, control structures, and their data/control dependencies. The context features of

API methods from code repositories are extracted and used to search for the best-matched pattern concerning the current context.

As a conclusion, such pattern mining based approaches are highly accurate in recommending public API member accesses. However, they rely heavily on the rich invocation histories of the method to be recommended. Consequently, such approaches are often confined to popular public APIs only because there are rich invocation examples of such public API methods in open-source applications whereas it is challenging to collect large numbers of invocation examples of project-specific API methods.

2.3 Type Based Code Completion

Except for pattern mining and statistical language model based approaches, there are also type based approaches that complete code by searching for valid expressions of given data types.

Mandelin *et al.* [10] propose PROSPECTOR to synthesize jungloid code fragments automatically in response to user queries. Jungloids are the chain of method calls that receives a given input object and returns a desired output object. They first construct a jungloid graph from method signatures with every expression corresponding to a path in the graph. Examples of downcasts are then extracted from program corpus as jungloids, converted to paths and added to the graph. Finally, PROSPECTOR searches for the shortest path from the given type to the desired type in the graph and synthesize a complete code fragment with the path. HeeNAMA differs from PROSPECTOR in that PROSPECTOR constructs a code fragment that might consist of multiple statements whereas HeeNAMA recommends a member access only. Another difference is that they leverage different information for code completion: PROSPECTOR leverages type information and examples of downcasts to synthesize code fragments whereas HeeNAMA leverages type information, examples of member accesses, and lexical similarity.

Gvero *et al.* [8] presents a general code completion approach inspired by complete implementation of type inhabitation for typed lambda calculus. Their approach constructs an expression and inserts it at the given location so that the whole program type checks. They introduce a succinct representation for type judgments that merges types into equivalence classes to reduce the search space. They rank potential solutions by preferring closer declarations to the program point and more frequently occurring declarations from a corpus of code. The approach is complete completion [8] because each synthesized expression is complete in that method calls have all of their arguments synthesized. HeeNAMA differs from their approach in that their approach makes the program type check by inserting a complete expression whereas HeeNAMA recommends a single member access.

2.4 Lexical Similarity Between Identifiers

Identifier names chosen by developers convey rich information, and thus they play an important role in program comprehension and source code analysis [36], [37], [38]. As suggested by Lawrie *et al.* [39], there are two main sources of domain information: identifier names and comments.

However, many developers do not write comments, so identifier names are critical for program comprehension.

A number of approaches have been proposed to exploit lexical similarity between semantically similar software entities. Liu *et al.* [9] present an empirical study of the lexical similarity between arguments and parameters of methods, and find that many arguments are more similar to the corresponding parameter than any alternative argument. Pradel and Gross [40], [41] exploit the lexical similarity between arguments and parameters to identify incorrect arguments. HeeNAMA also exploits the lexical similarity between semantically similar software entities. It differs from existing approaches in that it exploits the lexical similarity in code completion whereas exiting approaches [40], [41] exploits it in bug detection.

Cohen *et al.* [42] compare different string metrics, i.e., edit distance (also called Levenshtein distance) and cosine similarity, for matching names and records. The edit distance is used in our approach because it is simple and efficient.

3 APPROACH

3.1 Overview

In this section, we propose a heuristic and neural network based approach (HeeNAMA) for code completion. As stated in Section 1, HeeNAMA is confined to project-specific API member accesses that are defined as follows.

Definition 1 (Project-Specific API Member Access). A project-specific API member access is a method call or a field access via a base instance whose class type is declared and implemented within the project where the member access appears.

The base instance for a member access is the instance (object) whose member is accessed. For example, in the member access `a.b.c.d`, the base instance is `c`. For member access `a.b.c`, however, the base instance is `b`. For an incomplete assignment like "`x=a.b.`", HeeNAMA makes prediction only if the type of the base instance (`b` for the example) is declared and implemented within the project where the incomplete assignment is typed in. In the rest of this paper, *member access*, if not especially specified, refers specifically to *project-specific API member access*.

An overview of HeeNAMA is presented in Fig. 1. HeeNAMA is composed of two parts: a sequence of heuristics (notated as H_1 , H_2 , H_3 , respectively) and a neural network based filter. The first part predicts the next member access based on a sequence of heuristics. Whereas the second part decides whether the prediction is accurate enough to be presented to developers.

Given an incomplete assignment (e.g., "`String name = person.`"), HeeNAMA works as follows to predict the next member access:

- 1) First, it parses the incomplete assignment, and decides whether the base instance (`person` for the illustrating example) is a project-specific API instance. If yes (i.e., the declaration and implementation of the data type of the based instance are found within the enclosing project), it goes to the next step for code completion. Otherwise, HeeNAMA suggests invoking API specific code completion algorithms.

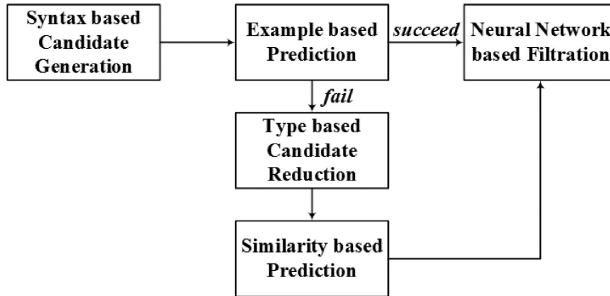


Fig. 1. Overview of HeeNAMA.

- 2) Second, it generates the initial candidates according to the type of the base instance as well as the location of the incomplete assignment. For the example “*String name = person.*”, the initial candidates include all members of the base instance *person* that are accessible on the location where the assignment is typed in.
- 3) Third, it looks for highly similar assignments within the project under development. If successful, it would predict the next member access based on the retrieved examples and the initial candidates. The prediction is forward to the neural network based filtering (Step 6). If failed, however, it would go to the next step to make prediction with other heuristics.
- 4) It removes the candidates that are type incompatible with the left hand side expression of the assignment according to our *type compatibility assumption*, i.e., the next member access should be type compatible with the left-hand side expression. For the given example of “*String name = person.*”, candidates that are not type compatible with *String* are removed from the candidate set.
- 5) It ranks the resulting candidates in descending order according to their lexical similarity with the identifier on the left hand side of the assignment (“*name*” for the illustrating example). The one on the top is taken as the most-likely member access.
- 6) Finally, it leverages a neural network to decide whether the most-likely member access should be recommended. Notably, the most-likely member access is potentially type incompatible with the left-hand side expression if it is predicted by Step 3.

According to the base instance on the right hand side of the incomplete assignment, HeeNAMA decides whether the member completion request is a project-specific API member access. Intuitively, HeeNAMA can make also such decisions according to the type of the left hand side expression as well: If the data type of the left hand side is defined within the project, the right hand side member completion request is a project-specific API member access. However, the decisions made in such a way could be inaccurate. Take “*String name = person.getName()*” as an illustrating example. The type of the right hand side base instance (i.e., *Person*) is project-specific, and thus the member completion request for “*String name = person.*” is project-specific and thus falls in the scope of HeeNAMA. However, the type of the left hand side expression (i.e., *String*) is not project-specific.

Details of the key steps are presented in the following sections.

3.2 Syntax Based Candidate Generation

First of all, HeeNAMA generates initial candidates based on Java syntax. Given an incomplete assignment, we extract its sketch that presents the key information our approach exploits for code completion

$$\text{sketch} = \langle lType, lName, baseIns, lct \rangle,$$

where *lType* is the type of left hand side expression, *lName* is the identifier name of left hand side, *baseIns* is the base instance, and *lct* is the location of the assignment.

For the incomplete assignment “*String name = person.*”, we have

$$\begin{aligned} lType &= "String" \\ lName &= "name" \\ baseIns &= "person". \end{aligned}$$

If the assignment is outside the package where the type of *person* (i.e., *Person*) is defined, the sketch of the assignment is

$$\text{sketch} = \langle String, name, person, outside \rangle.$$

lct indicates the relative location of the incomplete assignment with respect to the type of *baseIns*, i.e., *nested* (nested in the type of *baseIns*), *inherited* (inherited from the type of *baseIns*), *inside* (inside the package of the type of *baseIns*) or *outside* (outside the package of the type of *baseIns*). Consequently, *lct* can decide what kind of members of the base instance are available at the location. Based on the sketch, we generate the initial candidates *cdtSet* in two steps. First, we collect all members of the base instance *baseIns*. Second, we remove those members that are not accessible at the location (*lct*) of the assignment.

For the given example, if “*String name = person.*” is outside the package of class *Person*, the initial candidates are the public members of *Person*. However, if the assignment is within class *Person*, private and protected members of *Person* are taken as initial candidates as well.

3.3 Heuristic 1: Example Based Prediction

Repetitiveness is an important property of source code [7], [43]. Consequently, it is likely that we can predict the next member access based on highly similar member accesses. Algorithm 1 illustrates how HeeNAMA predicts the next member access based on sample assignments within the enclosing project.

First, given an incomplete assignment, HeeNAMA extracts its sketch (noted as *sketch*) that includes the type of its left hand side expression, the identifier name of its left hand side, the base instance, and its location (Line 2). Second, HeeNAMA retrieves sample assignments from the project under development (Line 4). The retrieving process is presented in Algorithm 2. We employ the Java parser provided by Java Development Tools (JDT) to parse source code of the project into Abstract Syntax Trees (ASTs). From such ASTs, we retrieve all AST nodes that represent assignments (Line 3 in Algorithm 2). For an assignment in the application, there may exist multiple scenarios where HeeNAMA can make predictions. For example, for assignment “*String name = this.person.getName()*”, HeeNAMA may make

prediction when incomplete assignment “*String name = this.*” or “*String name = this.person.*” is typed in. Each of such incomplete assignments and its following member access are presented as a sample:

```
smp = < icpAsgn, memb >,
```

where *icpAsgn* is the incomplete assignment and *memb* is the member access that follows the incomplete assignment *icpAsgn*. Line 6 in Algorithm 2 extracts all such samples from a given assignment *asgn*.

Algorithm 1. Example Based Member Access Prediction

Input: *icpAsgn* //incomplete assignment to be completed
cdtSet //initial candidate set
proj //project under development

Output: *member*

```

1: //construct a sketch of the incomplete assignment
2: sketch ← constructSketch(icpAsgn)
3: //extract sample assignments from the project
4: smpSet ← extractSampleAssignments(proj)
5: for each smp in smpSet do
6:   //construct a sketch of the incomplete assignment in
    sample
7:   skch ← constructSketch(smp.icpAsgn)
8:   if skch.lType = sketch.lType and
9:     skch.lName = sketch.lName and
10:    skch.baseIns = sketch.baseIns then
11:      for each cdt in cdtSet do
12:        if cdt = smp.memb then
13:          cdt.frequency ++
14:        end if
15:      end for
16:    end if
17:  end for
18: //sort candidates by frequency in descending order
19: sort(cdtSet)
20: if cdtSet[0].frequency > 0 then
21:   member ← cdtSet[0]
22: else
23:   member ← null
24: end if
25: return member

```

Algorithm 2. Extraction of Sample Assignments

Input: *proj* //project under development

Output: *smpSet* //the set of samples

```

1: smpSet ← ∅
2: //retrieve all assignments from the project
3: asgns ← retrieveAsgns(proj)
4: for each asgn in asgns do
5:   //extract all samples from the assignment
6:   smps ← extractSamples(asgn)
7:   smpSet.add(smmps)
8: end for
9: return smpSet

```

Third, HeeNAMA enumerates samples in the set *smpSet*, and extracts their sketches (Line 7). Lines 8-10 select samples that are highly similar to the incomplete assignment (*icpAsgn*, the first input of the algorithm) by comparing

their sketches. A sample is regarded as highly similar to the incomplete assignment when the types of their left hand side expressions, the identifier names of their left hand side expressions and their base instances are the same, respectively. Lines 11-13 count the frequency of the candidate members in the resulting highly similar samples. Based on the frequency, HeeNAMA sorts the candidate set (*cdtSet*) in descending order (Line 19). If the top one in *cdtSet* has a frequency greater than zero, it is regarded as the most-likely member to be accessed (Lines 20-25).

Taking the incomplete assignment (*icpAsgn*) “*String name = person.*” as an illustrating example, HeeNAMA first extracts its sketch

```
sketch = < String, name, person, outside >,
```

where *outside* means the assignment is typed in outside the package of class *Person*, and then it retrieves all sample assignments from the project under development. Suppose that it retrieves four sample assignments

```

asgn1: String name = this.person.getName()
asgn2: String name = student.name
asgn3: String name = person.getName()
asgn4: String name = this.person.name.

```

From these sample assignments, the approach extracts six samples

```

smp11 = < skch11, person >
skch11 = < String, name, this, outside >
smp12 = < skch12, getName() >
skch12 = < String, name, person, outside >
smp2 = < skch2, name >
skch2 = < String, name, student, outside >
smp3 = < skch3, getName() >
skch3 = < String, name, person, outside >
smp41 = < skch41, person >
skch41 = < String, name, this, outside >
smp42 = < skch42, name >
skch42 = < String, name, person, outside > .

```

Among these samples, *smp₁₂*, *smp₃* and *smp₄₂* share the same *lType*, *lName*, and *baseIns* in their sketches with the given incomplete assignment *icpAsgn*, and thus they are taken as highly similar samples. HeeNAMA counts the occurrence frequency of members in the resulting highly similar samples. Because *getName()* has the highest frequency, HeeNAMA suggests to complete the incomplete assignment *icpAsgn* with *getName()*.

3.4 Heuristic 2: Type Based Reduction of Candidate Set

If the first heuristic *H*₁ fails, HeeNAMA would generate recommendations with the other heuristics, i.e., the second heuristic *H*₂ and the third heuristic *H*₃. To make an assignment syntactically correct, the right-hand side expression of the assignment should be type compatible with the left-hand side expression. Consequently, the predicted member access

(for a given incomplete assignment) should be type compatible with the left-hand side expression of the assignment if the member access is the final token of the assignment. However, if the member access is not the final token of the assignment, it is not necessarily type compatible with the left-hand side. When an incomplete assignment is typed in, code completion tools do not know whether the next token is the final token or not. Consequently, in theory code completion tools (including the proposed one) could not assume that the next member access is type compatible with the left-hand side of the assignment.

However, to simplify the prediction, HeeNAMA makes the assumption that the next member access is type compatible with the left-hand side expression (called type compatibility assumption). Although the type of the left-hand side expression is not by definition compatible with the next member access, we find that this is the case in more than 80 percent of member accesses (see Section 4.3.2 for details).

Based on the type compatibility assumption, HeeNAMA reduces the size of the initial candidate set by removing those elements that are not type compatible with the left-hand side expression of the enclosing assignment. If the resulting candidate set is empty, i.e., no candidate is type compatible with the left-hand side, HeeNAMA refuses to make any prediction.

3.5 Heuristic 3: Similarity Based Prediction

Identifiers chosen by developers convey rich information about the semantics of the software entities [39]. An empirical study also suggests that semantically related software entities, e.g., arguments and their corresponding parameters, often have lexically similar identifiers (entity names) [9]. The right-hand side of an assignment is semantically related to its left-hand side, and thus it is likely that they are lexically similar. Consequently, in this section we propose the third heuristic (H_3) to predict the next member access based on the similarity between the candidates and the left hand side variable of the assignment. Given an incomplete assignment (whose sketch is $\text{sketch} = \langle lType, lName, baseIns, memb \rangle$) and its candidate set ($cdtSet$) generated by Heuristic 2, HeeNAMA works as follows to predict the next member access:

- First, for each candidate cdt in $cdtSet$, HeeNAMA calculates the Levenshtein distance (notated as $\text{Lev}(cdt, lName)$) between the identifier of cdt and that of $lName$.
- Second, based on the resulting Levenshtein distance, HeeNAMA calculates the lexical similarity between cdt and $lName$ as follows:

$$\text{sim} = 1 - \frac{\text{Lev}(cdt, lName)}{\max(\text{len}(cdt), \text{len}(lName))},$$

where $\text{len}(cdt)$ is the length of cdt (in characters), $\text{len}(lName)$ is the length of $lName$, and $\text{Lev}(cdt, lName)$ is the Levenshtein distance.

- Third, HeeNAMA sorts candidates in $cdtSet$ in descending order according to their similarities, and suggests to use the top one as the next member access.

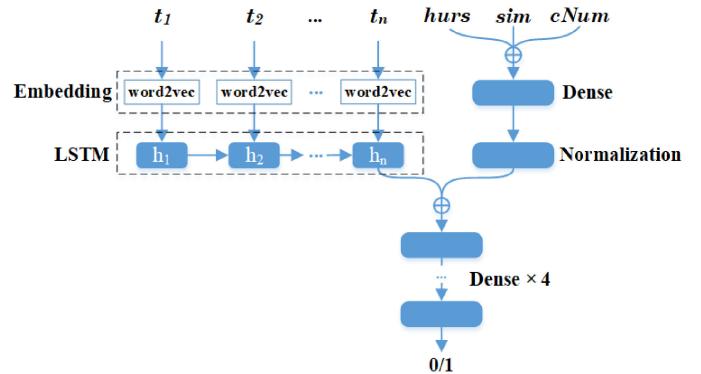


Fig. 2. Structure of the neural network based filter.

For the given example “*String name = person.*”, suppose that the candidate set $cdtSet$ contains four candidates: *name*, *getName()*, *age* and *getAge()*. HeeNAMA calculates their lexical similarity with the left-hand side variable “*name*”. The resulting similarities are 1.00, 0.57, 0.50, and 0.33, respectively. Since the first candidate *name* has the greatest similarity, HeeNAMA recommends *name* as the next token.

3.6 Neural Network Based Filtering

In the preceding sections, we present a sequence of heuristics to predict the next member access according to a given incomplete assignment. In this section, we present a neural network based filtering to filter out risky predictions that are likely incorrect.

The overall structure of the filter is presented in Fig. 2. On the left side is an LSTM [26] layer. Its input is a sequence of identifiers $\langle lType, lName, baseIns, memb \rangle$ where $lType$ is the type of left hand side expression, $lName$ is the identifier name of left hand side, $baseIns$ is the base instance on which the member is accessed, and $memb$ is the member predicted by heuristics (H_1 , H_2 , or H_3). To feed such identifiers into the neural network, we take the following measures. First, we tokenize such identifiers (i.e., $lType$, $lName$, $baseIns$ and $memb$) into sequences of tokens according to the camel case naming convention, notated as $st(lType)$, $st(lName)$, $st(baseIns)$, and $st(memb)$, respectively. For the example of $\langle \text{String}, \text{name}, \text{person}, \text{getName} \rangle$, we tokenize them into four sequences of $\langle \text{String} \rangle$, $\langle \text{name} \rangle$, $\langle \text{person} \rangle$ and $\langle \text{get}, \text{Name} \rangle$. Second, we lowercase all the tokens and concatenate such sequences as well as separators (notated as sep) into one sequence, noted as cst

$$\begin{aligned} cst &= \langle st(lType), sep, st(lName), sep, st(baseIns), \\ &\quad sep, st(memb) \rangle \\ &= \langle t_1, t_2, \dots, t_n \rangle, \end{aligned}$$

where n is the total number of tokens in $lType$, $lName$, $baseIns$, $memb$ and separators. Consequently, the resulted sequence for the given example is $\langle \text{string}, \text{sep}, \text{name}, \text{sep}, \text{person}, \text{sep}, \text{get}, \text{name} \rangle$.

Given the sequence $cst = \langle t_1, t_2, \dots, t_n \rangle$, we map the i th token t_i into a D -dimensional vector $e_i = W \cdot o(t_i)$, where $W \in \mathbb{R}^{D \times V}$ is an embedding matrix pre-trained with *word2vec* model [44] on identifiers that could be collected from sample assignments, and $o(t_i)$ is a one-hot encoder converting t_i into a vector of V dimensions. We embed such

tokens into numeric vectors with the well-known *word2vec* because of the following reasons. First, each of the identifiers are finally tokenized into a sequence of words that is essentially a short English phrase, and *word2vec* has been proved effective in vectoring short English phrases. Second, *word2vec* has been successfully employed by Nguyen *et al.* [45] to vectorize identifiers in source code. We pass such vectors into a recurrent neural network with long short-term memory units (LSTM) and compute the hidden vector at the i th time step as

$$h_i = f_{LSTM}(h_{i-1}, e_i), \quad i = 1, \dots, n, \quad (1)$$

where $h_i \in \mathbb{R}^D$ and f_{LSTM} is the LSTM function. Then we take the final hidden state h_n as the output of LSTM layer. We employ LSTM because of two reasons. First, LSTM has been proved effective and efficient in natural language processing. In our case, all of the involved identifiers (i.e., *lType*, *lName*, *baseIns* and *memb*) are essentially natural language descriptions. Second, LSTM accepts variable-length input. In our case, the length of the input depends on how many words the given identifiers contain, and it may change dramatically from assignment to assignment. Consequently, LSTM fits well for our case.

Algorithm 3. Training Process of the Filter

```

Input: projs //sample projects
        filter //filter (neural network) to be trained
Output: filter //updated filter
1: smpSet  $\leftarrow \emptyset$ 
2: for each proj in projs do
3:   //extract sample assignments from the project
4:   set  $\leftarrow$  extractSampleAssignments(proj)
5:   smpSet  $\leftarrow$  smpSet + set
6: end for
7: //generate a training set with sample assignments
8: trainSet  $\leftarrow$  generateTrainingSet(smpSet)
9: //train filter with the training set
10: filter  $\leftarrow$  filter.train(trainingSet)
11: return filter
12:
13: function generateTrainingSet(smpSet)
14:   trainingSet  $\leftarrow \emptyset$ 
15:   for each smp in smpSet do
16:     //make prediction by heuristics
17:     memb', hurs, sim, cNum  $\leftarrow$ 
18:       predict(smp.icpAsgn)
19:     skch  $\leftarrow$  constructSketch(smp.icpAsgn)
20:     input  $\leftarrow$  < skch.lType, skch.lName,
21:           skch.baseIns, memb', hurs, sim, cNum >
22:     if memb' = smp.memb then
23:       output  $\leftarrow$  1
24:     else
25:       output  $\leftarrow$  0
26:     end if
27:     item  $\leftarrow$  < input, output >
28:     trainingSet.add(item)
29:   end for
30:   return trainingSet
31: end function

```

On the right side is a normalization layer that normalizes *hurs*, *sim*, and *cNum*. *hurs* indicates which heuristic (H_1 or

H_3) makes the prediction. *sim* is the lexical similarity between *memb* and *lName*. *cNum* is the number of candidates in the initial candidate set generated according to Java syntax (as introduced in Section 3.2). The three numerical values are concatenated into a three-dimensional vector and fed into the Dense layer which converts them into a D -dimensional vector. The normalization layer used here is a learned layer-normalization. We normalize these data because some of them (e.g., *cNum*) are usually much larger than others (e.g., *sim*). The output of the LSTM layer and the normalization layer is merged by concatenation and fed into dense layers whose output is either one (suggesting that the prediction is safe) or zero (suggesting that the prediction is risky).

The neural network based filter could be trained in advance with examples from open-source applications. Algorithm 3 presents the training process. The training process consists of three steps, i.e., extracting sample assignments from sample projects (Lines 1-5), generating the training set (Line 8), and training filter with the training set (Line 10).

On the first step, we extract sample assignments from corpus (Line 4) in the same way as we did in Section 3.3. Based on the resulting samples (noted as *smpSet*), we generate a training set (noted as *traingSet*) on Line 8. The generation process is explained as follows. For each sample *smp*, we employ the heuristics (H_1 , H_2 and H_3) to make prediction for the incomplete assignment (*smp.icpAsgn*) on Lines 17-18. The output of the prediction includes the predicted member access (notated as *memb'*), the number of initial candidates (notated as *cNum*), the heuristic that makes the prediction (notated as *hurs*), and the lexical similarity between *memb'* and *smp.icpAsgn.lName* (notated as *sim*). With the output and the constructed sketch *skch* (Line 19), we construct an input (Lines 20-21) for the filter as

$$\begin{aligned} \text{input} = & < lType, lName, baseIns, memb', \\ & hurs, sim, cNum > . \end{aligned}$$

If the predicted member access (*memb'*) is exactly the same as that in sample (*smp.memb*), i.e., *memb' = smp.memb*, the expected output of the network (notated as *output*) is one (Lines 22-23). Otherwise, *output* is zero (Lines 24-25). The resulting training item *item* = < *input*, *output* > is added to the training set that is in turn used to train the neural network (Lines 27-28). If the prediction fails, i.e., no member access is recommended, we ignore the sample *smp*. No training items are generated based on this sample.

4 EVALUATION

In this section, we evaluate HeeNAMA on open-source applications.

4.1 Research Questions

The evaluation investigates the following research questions:

- *RQ1*: How often do project-specific API member accesses appear in the right hand side of assignments? How often are they stacked or unstacked?
- *RQ2*: How often are project-specific API members in the right hand side of assignments type compatible with the left hand side?

- RQ3: Does HeeNAMA outperform the state-of-the-art approach or the state-of-the-practice tool? If yes, to what extent?
- RQ4: How do the heuristics and the LSTM based filter influence the performance of HeeNAMA?
- RQ5: Can HeeNAMA be extended to recommend project-specific API member accesses nested in method invocations?
- RQ6: How well do API-specific approaches work in suggesting project-specific API member accesses if they are trained on within-project code?
- RQ7: How well does HeeNAMA work if trained with all API member accesses (considering both project-specific and public ones)?
- RQ8: How well does HeeNAMA perform on recently created applications?

HeeNAMA is based on the assumption that there are a large number of project-specific API member accesses on the right hand side of assignments (called member accesses on RHS for short). If the assumption does not hold, HeeNAMA will not be employed frequently, and thus it may be useless. Answering the research question RQ1 helps to validate the assumption. Notably member accesses are further divided into *stacked* and *unstacked*. For example, member access c in assignment $x=a.b.c$ is unstacked whereas b is stacked because b is followed immediately by another member access. Although HeeNAMA makes predictions for both stacked and unstacked member accesses, it is more challenging to predict stacked ones because the type based reduction of candidate sets (as introduced in Section 3.4) may not work for stacked ones. For example, while predicting b in assignment $x=a.b.c$, HeeNAMA filters candidates based on the assumption that the member access to be predicted (b in the example) is type compatible with the left hand side of the assignment (x in the example). However, it is not necessarily true for stacked member accesses: assignment $x=a.b.c$ requires c (instead of b) to be type compatible with x . Investigating how often member accesses are stacked or unstacked may help to reveal how often the assumption taken by HeeNAMA holds.

H_2 in Section 3.4 is based on the assumption that the next project-specific API member access in the right hand side of assignment is often type compatible with the left-hand side expression (type compatibility assumption). Answering the research question RQ2 helps to validate the assumption.

RQ3 concerns the performance of HeeNAMA against the state-of-the-art approach and the state-of-the-practice tool. To answer RQ3, we compare HeeNAMA against SLP-Core and Eclipse. SLP-Core is the implementation of the state-of-the-art approach proposed by Hellendoorn *et al.* [20]. Eclipse is a well-known and widely used IDE. SLP-Core and Eclipse are selected for comparison because of the following reasons. First, SLP-Core is the state-of-the-art approach whereas Eclipse is the state-of-the-practice IDE. Second, both SLP-Core and Eclipse can predict project-specific API member accesses. Third, both SLP-Core and Eclipse are publicly available online, which facilitates readers to repeat the evaluation. Although some advanced approaches are reported highly accurate [18], [19], [46], [47], they are not selected for comparison because we fail to get their replication packages or the packages could not be easily adapted to Java. Both

SLP-Core and Eclipse are generic, and they can predict all kinds of tokens or elements whereas HeeNAMA is confined to project-specific API member accesses on RHS. The purpose of the comparison is to investigate whether HeeNAMA can improve the performance of code completion by focusing on special cases and by taking specific and fine grained context of such cases.

As specified in Section 3, HeeNAMA is composed of a sequence of heuristics and a neural network based filter. Answering research question RQ4 helps to reveal how such heuristics and filter influence the performance (e.g., precision and recall) of HeeNAMA. We also conducted an experiment to explore how different learning algorithms influence the performance of the filter and HeeNAMA. The results can be found in the online appendix.¹ Answering the research question RQ4 also helps to explain why HeeNAMA works (or not works).

As specified in Section 1, HeeNAMA focuses on a specific but common case of code completion: suggesting the following member access whenever a project-specific API instance is followed by a dot on the right hand side of an assignment. Notably, there is a similar case where HeeNAMA could be applied: suggesting the following member access when a project-specific API instance is nested in a method invocation, e.g., suggesting the member b in the example of $m(a.b)$. We call such project-specific API member accesses nested in method invocations as nested member accesses for short. RQ5 investigates the possibility of extending HeeNAMA to recommend nested member accesses. Answering the research question RQ5 helps to validate the practical usefulness of HeeNAMA.

Research question RQ6 concerns the performance of HeeNAMA against API-specific approaches that are trained on within-project code. API-specific approaches are often highly accurate in recommending API member accesses because they can discover frequent patterns in training corpus. If we simply train API-specific prediction models on within-project code, however, they might discover project-specific patterns as well, and thus the resulting models might suggest project-specific member accesses as HeeNAMA does. Assuming that a model is requested to recommend a member of class C in method M , if the model is continually updated with the code in the project, it would have the knowledge of all the code in the project with the exception of method M . Thus, if there are usage patterns of class C in the project, the model would be able to make a good recommendation. That is the rationale for the investigation of RQ6. To answer RQ6, we compare HeeNAMA against an API-specific approach CSCC [35] (trained on within-project code) in suggesting project-specific API member accesses. CSCC [35] is the latest pattern mining based API-specific approach. In the evaluation, CSCC is incrementally trained with the code in the test project.

Research question RQ7 concerns the performance of HeeNAMA when trained with all project-specific and public API member accesses. Although HeeNAMA is proposed for prediction of project-specific API member access, it can be applied to train on public API member access as well. To

1. <https://github.com/CC-CG/HeeNAMA/tree/master/appendix>

answer RQ7, we train HeeNAMA with all API member accesses on RHS in subject applications and then evaluate its performance on project-specific and public API member accesses on RHS separately.

Research question RQ8 concerns the performance of HeeNAMA on recently created applications. To answer RQ8, we collect a new dataset of nine open-source Java applications which are created in recent years (i.e., since January 1, 2015). With the new dataset, we evaluate the performance of HeeNAMA against SLP-Core, Eclipse and CSCC again. The comparison helps to reveal the impact of replacing evaluation applications on the performance of HeeNAMA.

4.2 Setup

4.2.1 Subject Applications

We conduct the evaluation on nine open-source applications as shown in Table 1. We select such applications because they have been employed to evaluate code completion approaches successfully [7], [18], [23], [24]. An overview of the subject applications is presented in Table 1. They cover various domains such as software build, database management, and search engine. The size (LOC) of subject applications varies from 91,760 to 1,591,582. In all nine applications, there are 861,618 API member accesses in total, and 521,752 of them are project-specific API member accesses. In 521,752 project-specific API member accesses, 103,695 members are accessed on the right hand side of assignments. Notably, our evaluation is only conducted on project-specific API member accesses on the right hand side of assignments, i.e., evaluated approaches are requested for code completion on the 103,695 member accesses on RHS in subject applications.

4.2.2 Process

On the nine open-source applications presented in Table 1, we carry out a k -fold ($k = 9$) cross-validation. On each fold, a single application is used as testing data set (noted as *testSet*) whereas the others (eight applications) are used as training data (noted as *trainingSet*). Each of the subject applications is used as testing data set for once.

Each fold of the evaluation follows the following process:

- 1) SLP-Core and the filter in HeeNAMA are trained with *trainingSet* independently.
- 2) For each project-specific API member access on RHS in the *testSet*, we remove source code after the dot of member access (including the member) in the enclosing file. The resulting incomplete assignment is used as a query to HeeNAMA, SLP-Core and Eclipse. This step simulates the scenarios where source code in each file is typed in from the top to the bottom.
- 3) For each query, each code completion system is asked to return a prediction of the missing member access. A prediction is correct if and only if the predicted member access is exactly the same as that in the original source code. After prediction, the original member access is used to train SLP-Core and the first heuristic in HeeNAMA in a incremental way.
- 4) Based on such predictions, we calculate the performance (precision and recall) for these code completion approaches.

4.2.3 Configuration

We empirically set the embedding dimension (100) for *word2vec* and the dimension (10) for the intermediate Dense layers. We also empirically set the activation function (*ELU* [48]) for dense layers, and their optimizer (*Nadam* [49]). Other settings of the evaluation could be found in the implementation of HeeNAMA that is publicly available at <https://github.com/CC-CG/HeeNAMA> [50]. For comparison with SLP-Core, we employ the nested cache n-gram model in the dynamic setting which is reported best-in-class [20]. During evaluation, SLP-Core is incrementally trained with each member access in the test project once the member access has been recommended, i.e., we employ the dynamic setting as in [20]. For a given completion, those member accesses that have been recommended before are left in their usual place and thus can be learned by the nested cache n-gram model of SLP-Core. Concerning Eclipse, for each member access in the test project, we programmatically invoke the default code recommender in Eclipse (version 4.5).

4.2.4 Metrics

To answer research questions RQ3 to RQ8, we calculate the precision of top k recommendation for various approaches in recommending member accesses as follows:

$$\text{Precision}@k = \frac{N_{\text{accepted}}@k}{N_{\text{recommended}}}, \quad (2)$$

where $N_{\text{accepted}}@k$ is the number of the cases where one of items within the top k recommendation list is accepted, and $N_{\text{recommended}}$ is the number of cases the evaluated approach tries. The recall of top k recommendation is calculated as follows:

$$\text{Recall}@k = \frac{N_{\text{accepted}}@k}{N_{\text{tested}}}, \quad (3)$$

where N_{tested} is the number of tested member accesses. To answer RQ3 and RQ6, the value of k is set to 1, 3 and 5. While for the other research questions, we only present the precision and recall of top 1 recommendation. We also compute the F-measure to summarize the precision and recall values of top 1 recommendation as follows:

$$F_\beta = (\beta^2 + 1) \cdot \frac{\text{Precision}@1 \cdot \text{Recall}@1}{\beta^2 \cdot \text{Precision}@1 + \text{Recall}@1}, \quad (4)$$

where $\beta \in \mathbb{R}$ is a harmonic coefficient. In this paper, we set β to 0.5, 1 and 2 to get evaluating metrics $F_{0.5}$ -measure, F_1 -measure and F_2 -measure, respectively [51]. F_1 -measure integrates precision and recall values by the same weight. F_2 -measure assigns a larger weight to the precision value whereas $F_{0.5}$ -measure assigns a larger weight to the recall value. That is to say, F_2 -measure focuses more on the improvement of the recall value whereas $F_{0.5}$ -measure focuses more on the improvement of the precision value. The three metrics evaluate the integrated performance of the approach from different aspects.

TABLE 2
Popularity of Project-Specific API Member Accesses on RHS

Applications	All Accesses (N_{all})	Accesses on RHS (N_{RHS})	$\frac{N_{RHS}}{N_{all}}$	Unstacked Accesses on RHS ($N_{unstacked}$)	$\frac{N_{unstacked}}{N_{RHS}}$	$\frac{N_{unstacked}}{N_{all}}$
Ant	23,702	3,400	14.34%	3,064	90.12%	12.93%
Batik	24,031	5,158	21.46%	4,773	92.54%	19.86%
Cassandra	100,575	17,724	17.62%	13,786	77.78%	13.71%
Log4J	31,927	7,137	22.35%	4,963	69.54%	15.54%
Lucene-solr	266,040	56,623	21.28%	47,373	83.66%	17.81%
Maven2	10,070	1,411	14.01%	1,209	85.68%	12.01%
Maven3	18,067	2,351	13.01%	2,037	86.64%	11.27%
Xalan-J	22,621	4,154	18.36%	3,946	94.99%	17.44%
Xerces	24,719	5,737	23.21%	5,484	95.59%	22.19%
Total	521,752	103,695	19.87%	86,635	83.55%	16.60%

4.3 Results and Analysis

4.3.1 RQ1: Project-Specific API Member Access on RHS

To address RQ1, we count the number of member accesses in subject applications as well as those on the right-hand side of assignments (RHS). We also count the number of unstacked member accesses on the right-hand sides of assignments. The results are presented in Table 2. The first column presents the names of subject applications. The second column presents the number of member accesses in subject applications. The third column and the forth column present the number of member accesses on RHS and the ratio of member accesses on RHS to all member accesses, respectively. The number of unstacked member accesses on RHS is presented in the fifth column. The ratio of them to member accesses on RHS and the ratio of them to all member accesses are presented in the last two columns.

From this table we make the following observations:

- First, the number of member accesses on RHS is quite large. For the nine subject applications, the total number is as much as 103,695. Consequently, highly accurate prediction approaches (if there is any) for such member accesses would be employed frequently, and thus could be beneficial for developers.
- Second, member accesses on RHS account for a significant proportion of member accesses in the source code. On average, they account for $19.87\% = 103,695 / 521,752$ of the member accesses.
- Third, most (83.55 percent) of member accesses on RHS are unstacked, i.e., where HeeNAMA actually works. The total number of unstacked member accesses is as much as 86,635 for nine subject applications. They account for $16.60\% = 86,635 / 521,752$ of all member accesses.

From the analysis in the preceding paragraphs, we conclude that there is a large number of member accesses on the right-hand side of assignments. Consequently, code completion approaches/tools confined to such member accesses could be useful as long as they are accurate.

4.3.2 RQ2: Type Compatibility of Project-Specific API Member Accesses

To address RQ2, we count the number of member accesses on RHS and the number of those which are type compatible with the left hand side expression. We present the results in Table 3. The first column shows the names of subject applications. The

second column presents the number of member accesses on RHS. The third column presents the number of member accesses on RHS which are type compatible with the left hand side expression. The ratio of type compatible member accesses to those on RHS is presented in the last column.

From Table 3, we observe that the ratio is high. As shown in the table, the ratio varies from 70.42 to 92.27 percent. On average, $81.82\% = 84,842 / 103,695$ of member accesses on RHS are type compatible with the left hand side.

From the analysis in the preceding paragraph, we conclude that in most cases the type compatibility assumption is correct.

4.3.3 RQ3: Comparison Against Existing Approaches

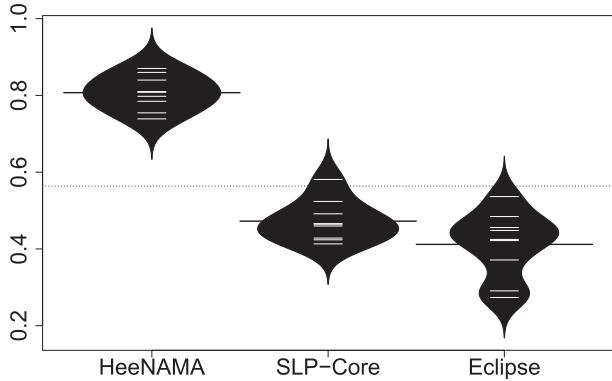
To address RQ3, we compare HeeNAMA against SLP-Core [20] and Eclipse IDE on nine open-source applications. The evaluation results are presented in Table 4 and Fig. 3. In the table, the first column presents the names of subject applications. The second to seventh columns present the precision and recall of HeeNAMA in the top k recommendation list, respectively. The precision of SLP-Core and Eclipse at top k is presented in the last six columns, respectively. Different from HeeNAMA, both SLP-Core and Eclipse always make recommendation whenever they are requested, i.e., the number of cases they try is equal to the number of tested member accesses. Consequently, for SLP-Core and Eclipse, recall is always equal to precision, and thus it is omitted from the table. Fig. 3 presents evaluation results at the top 1 recommendation with bean-plot. Each bean in Fig. 3 presents the resulting precision (sub-graph on the left) or recall (sub-graph on the right) of an evaluated approach on subjection

TABLE 3
Type Compatibility of Project-Specific API Member Accesses

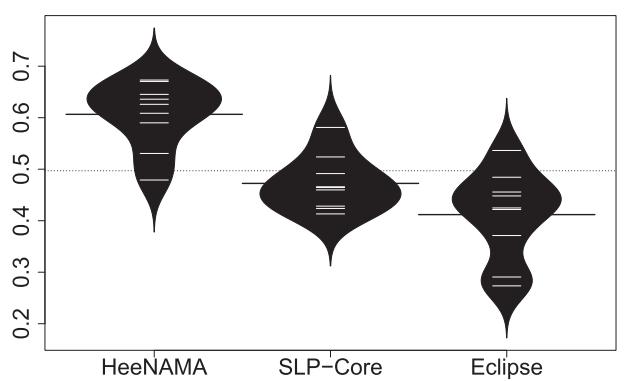
Applications	Accesses on RHS (N_{RHS})	Type Compatible Accesses (N_{TC})	$\frac{N_{TC}}{N_{RHS}}$
Ant	3,400	3,042	89.47%
Batik	5,158	4,567	88.54%
Cassandra	17,724	13,357	75.36%
Log4J	7,137	5,026	70.42%
Lucene-solr	56,623	46,554	82.22%
Maven2	1,411	1,204	85.33%
Maven3	2,351	2,045	86.98%
Xalan-J	4,154	3,833	92.27%
Xerces	5,737	5,214	90.88%
Total	103,695	84,842	81.82%

TABLE 4
Comparison Against Existing Approaches

Applications	HeeNAMA						SLP-Core			Eclipse		
	Precision			Recall			Precision			Precision		
	Top 1	Top 3	Top 5	Top 1	Top 3	Top 5	Top 1	Top 3	Top 5	Top 1	Top 3	Top 5
Ant	80.82%	85.17%	85.92%	63.59%	69.91%	71.44%	42.38%	51.12%	54.65%	48.44%	67.85%	78.15%
Batik	75.46%	83.53%	85.90%	60.86%	68.94%	71.56%	42.83%	51.01%	53.06%	42.48%	67.78%	76.93%
Cassandra	81.00%	84.96%	84.63%	59.00%	62.97%	63.48%	41.32%	48.10%	51.30%	44.81%	68.60%	79.87%
Log4J	84.00%	87.32%	88.02%	67.31%	79.04%	79.99%	49.15%	54.32%	56.79%	37.12%	59.41%	65.28%
Lucene-solr	86.00%	88.05%	87.94%	62.60%	67.05%	67.94%	52.38%	60.63%	63.61%	53.64%	75.64%	83.23%
Maven2	87.03%	91.92%	92.51%	67.04%	74.20%	75.27%	58.11%	65.27%	67.90%	45.57%	72.43%	82.99%
Maven3	73.89%	85.12%	85.46%	64.53%	72.27%	73.50%	45.98%	53.00%	57.38%	27.35%	64.99%	73.63%
Xalan-J	79.77%	79.57%	80.02%	53.06%	66.18%	67.96%	46.44%	53.71%	56.69%	42.18%	67.21%	75.83%
Xerces	78.47%	80.72%	80.69%	47.90%	59.32%	60.31%	46.59%	59.28%	62.40%	29.07%	47.25%	59.12%
Average	83.36%	86.39%	86.51%	61.16%	67.12%	68.11%	48.84%	56.80%	59.79%	47.74%	70.48%	79.09%



(a) Precision at Top 1



(b) Recall at Top 1

Fig. 3. Comparison against existing approaches.

applications. The white small lines represent the precision or recall on a single subject application, and the shape of the beans represents the distribution of the performance. The black lines crossing beans represent the average precision (or recall) of the evaluated approaches.

From Table 4 and Fig. 3, we make the following observations:

- First, HeeNAMA is precise. The precision at top 1 recommendation varies from 73.89 to 87.03 percent, and the average precision is as much as 83.36 percent. In other words, in most cases (more than 83 percent) the approach suggests the member access exactly the same as developers want.
- Second, HeeNAMA is significantly more precise than SLP-Core and Eclipse. On each of the subject applications, the precision of HeeNAMA at top k is always greater than that of SLP-Core and Eclipse. We also compare their precision at top 1 in Fig. 3a where the distance between different approaches is obvious. On average, HeeNAMA improves precision at top 1 significantly by $70.68\% = (83.36\% - 48.84\%) / 48.84\%$.
- Third, HeeNAMA improves recall at top 1 recommendation significantly. Although its recall varies dramatically from 47.90 to 67.31 percent, it is always greater than that of SLP-Core and Eclipse. The bean plot in Fig. 3b visually illustrates the distance among

such approaches. On average, it improves recall at top 1 by $25.23\% = (61.16\% - 48.84\%) / 48.84\%$.

Notably, SLP-Core and Eclipse work well on challenging project-specific API member accesses, achieving a precision/recall of 48.84 and 47.74 percent, respectively. One possible reason for the success of SLP-Core is that it leverages examples of member accesses in the test project (i.e., within-project code) because its nested cache n-gram model is continually updated while recommendation [20]. Eclipse succeeds frequently because it recommends member accesses according to type information of the left hand side which is also leveraged by HeeNAMA and thus it makes some correct recommendations.

HeeNAMA refuses to make recommendations when it lacks of confidence. We present the frequency of HeeNAMA refusing to make recommendations in Table 5. From this table, we observe that on more than a quarter (26.62 percent) cases HeeNAMA refuses to make recommendations. Comparing Table 4 against Table 5, we observe that the recall is influenced by the frequency of refusal. The results are reasonable in that if HeeNAMA makes fewer recommendations, it has smaller chance to make correct recommendations (and thus lower recall).

As suggested by Table 2, project-specific API member accesses could be further divided into stacked and unstacked ones. To this end, we further investigate how well HeeNAMA works at top 1 recommendation on such subsets. On

TABLE 5
Frequency of HeeNAMA Refusing to Make Recommendations

Applications	Accesses on RHS (N_{RHS})	Refused Accesses (N_{Ref})	$\frac{N_{Ref}}{N_{RHS}}$
Ant	3,400	725	21.32%
Batik	5,158	998	19.35%
Cassandra	17,724	4,814	27.16%
Log4J	7,137	1,418	19.87%
Lucene-solr	56,623	15,405	27.21%
Maven2	1,411	324	22.96%
Maven3	2,351	298	12.68%
Xalan-J	4,154	1,391	33.49%
Xerces	5,737	2,235	38.96%
Total	103,695	27,608	26.62%

the stacked ones, HeeNAMA achieves a precision of 74.86 percent and a recall of 45.84 percent whereas the precision (and recall) of SLP-Core and Eclipse IDE is 48.12 and 7.11 percent, respectively. On the unstacked ones, HeeNAMA achieves a precision of 84.71 percent and a recall of 64.18 percent whereas the precision (and recall) of SLP-Core and Eclipse IDE is 48.98 and 55.75 percent, respectively. The results suggest that HeeNAMA works much better on unstacked ones than on stacked ones. Stacking does not affect SLP-core because SLP-core is completely based on token sequences captured by n-gram models. However, stacking does affect Eclipse negatively because Eclipse leverages the type information of the left hand side to make recommendations and stacking makes such type information useless (or even misleading). However, the results also suggest that even on the stacked ones, HeeNAMA outperforms SLP-Core and Eclipse IDE as well.

From the analysis in the preceding paragraphs, we conclude that HeeNAMA is precise, and it significantly outperforms both the state-of-the-art approach and the state-of-the-practice tool in suggesting the next member access for assignments.

4.3.4 RQ4: Impacts of Heuristics and Filter

As introduced in Section 3, HeeNAMA is composed of three heuristics and neural network based filter. The evaluation in the preceding sections suggests that HeeNAMA as a whole is accurate. To investigate how the heuristics and filter influence the performance of HeeNAMA, we repeat the evaluation (including both the training and testing phase) for four times. On the first three times, we disable the heuristics (i.e., H_1 , H_2 , H_3), respectively. Finally, we disable the neural network based filter, and repeat the evaluation for the last time. For example, when disabling H_1 , the filter is presented with only type-compatible member accesses that are ranked first by H_3 according to lexical similarity.

The evaluation results are presented in Fig. 4 and Table 6. From Fig. 4 and Table 6, we make the following observations:

- First, disabling any of the three heuristics leads to significant reduction in recall. The reduction is as much as $14.60\% = (61.16\% - 52.23\%) / 61.16\%$, $36.35\% = (61.16\% - 38.93\%) / 61.16\%$, and $27.11\% = (61.16\% - 44.58\%) / 61.16\%$, respectively. The evaluation results suggest that all of the heuristics are critical for HeeNAMA to achieve high recall.

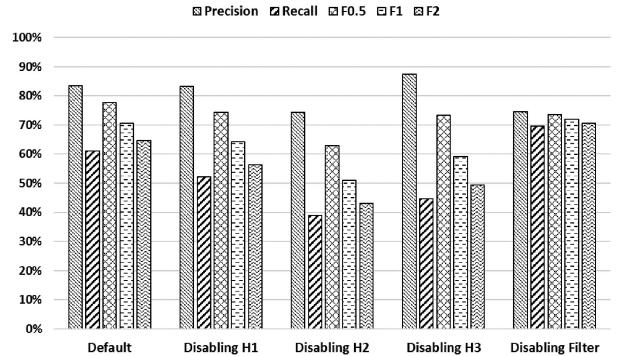


Fig. 4. Impacts of heuristics and filter.

- Second, disabling neural network based filter improves recall at the cost of reduced precision. Precision is reduced by $10.64\% = (83.36\% - 74.49\%) / 83.36\%$ whereas recall is improved by $13.73\% = (69.56\% - 61.16\%) / 61.16\%$. Although disabling the filter results in more balanced precision and recall, the filter is beneficial because of the following reasons. First, although the filter reduces F_2 (that biases for recall) from 70 to 65 percent, F_1 keeps stable and $F_{0.5}$ (that biases for precision) increases from 73 to 78 percent. Second, for code completion, high precision (and thus few uncorrect recommendations) is critical because incorrect recommendations are often misleading and even worse than no recommendation at all. Code complete tools that frequently make incorrect recommendations would lose trust of developers, and will be finally discarded by developers. To this end, we introduce the filter to improve precision (at the cost of moderate reduction in recall) by removing risky recommendations. The evaluation results suggest that the filter works as expected.
- Third, disabling H_1 and H_3 has little influence on the precision of HeeNAMA. One possible reason is that the neural network based filter (working at the final phase of HeeNAMA) can filter out most of the risky predictions, and thus guarantees the final precision.

We also present in Table 7 the frequency of HeeNAMA refusing to make recommendations when one of the heuristics or the filter is disabled. From the table, we observe that disabling any of the three heuristics improves the frequency of refusal while disabling the filter reduces the frequency, which is consistent with the observation from Table 6. When disabling heuristics, HeeNAMA refuses to make recommendations more frequently and thus it achieves lower recall. However, when disabling the filter, the frequency of HeeNAMA refusing to make recommendations reduces greatly so its recall improves.

We investigate the impact of the features leveraged by the filter and present the evaluation results in Table 8. From the table, we conclude that each of the features is useful. We also investigate the impact of reducing one or two hidden layers used by the filter. The results suggest that reducing one or two hidden layers would result in reduction (1.56 and 2.99 percent, respectively) in F_1 .

To further investigate the contribution of heuristics, we evaluate the performance of pair-wise disabling (i.e., activating each heuristic on subject applications). The evaluation

TABLE 6
Impacts of Heuristics and Filter

Applications	Default		Disabling H_1		Disabling H_2		Disabling H_3		Disabling Filter	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
Ant	80.82%	63.59%	82.75%	48.12%	77.53%	44.15%	89.00%	35.21%	71.27%	67.79%
Batik	75.46%	60.86%	77.34%	48.10%	61.04%	35.21%	85.27%	38.95%	68.59%	64.99%
Cassandra	81.00%	59.00%	80.28%	51.84%	81.05%	39.33%	85.61%	39.35%	73.07%	66.82%
Log4J	84.00%	67.31%	84.64%	60.08%	69.57%	42.19%	87.82%	53.13%	78.97%	73.28%
Lucene-solr	86.00%	62.60%	85.63%	53.46%	73.84%	38.81%	89.12%	48.65%	77.22%	71.79%
Maven2	87.03%	67.04%	87.49%	53.01%	90.09%	60.60%	91.02%	33.03%	77.87%	72.08%
Maven3	73.89%	64.53%	76.13%	50.32%	85.66%	58.70%	78.77%	31.09%	71.58%	67.59%
Xalan-J	79.77%	53.06%	80.98%	44.37%	65.52%	32.62%	86.21%	36.11%	63.89%	62.52%
Xerces	78.47%	47.90%	73.91%	44.00%	72.04%	26.18%	77.21%	35.14%	62.82%	61.74%
Average	83.36%	61.16%	83.14%	52.23%	74.32%	38.93%	87.45%	44.58%	74.49%	69.56%

TABLE 7
Frequency of HeeNAMA Refusing to Make Recommendations

Applications	Accesses on RHS	Default	Disabling H_1	Disabling H_2	Disabling H_3	Disabling Filter
Ant	3,400	21.32%	41.85%	43.05%	60.44%	4.88%
Batik	5,158	19.35%	37.81%	42.32%	54.32%	5.25%
Cassandra	17,724	27.16%	35.43%	51.47%	54.04%	8.55%
Log4J	7,137	19.87%	29.02%	39.36%	39.50%	7.21%
Lucene-solr	56,623	27.21%	37.57%	47.44%	45.41%	7.03%
Maven2	1,411	22.96%	39.41%	32.73%	63.71%	7.44%
Maven3	2,351	12.68%	33.90%	31.47%	60.53%	5.57%
Xalan-J	4,154	33.49%	45.21%	50.21%	58.11%	2.14%
Xerces	5,737	38.96%	40.47%	63.66%	54.49%	1.72%
Total	103,695	26.62%	37.18%	47.62%	49.02%	6.62%

results are presented in Table 9. The table does not present the option of activating the filter alone because the filter cannot work without the candidate items generated by heuristics. From the table, we observe that single activation results in significant reduction in performance. For example, activating H_1 only increases precision slightly by $1.36\% = (84.51\%-83.36\%)/83.36\%$ but reduces recall significantly by $23.09\% = (61.16\%-47.04\%)/61.16\%$. Single activation for H_2 or H_3 significantly reduces both precision and recall.

We conclude from the preceding analysis that the proposed heuristics and the filter are useful.

4.3.5 RQ5: Performance on Nested Member Accesses

To address RQ5, we evaluate HeeNAMA on nested member accesses from nine open-source applications. We also compare the performance of HeeNAMA against SLP-Core and Eclipse IDE. The evaluation results are presented in Table 10. In the table, the first column presents the names of

subject applications. The second column presents the number of member accesses in subject applications. The third column and the forth column present the number of nested member accesses and the ratio of nested member accesses to all member accesses, respectively. The fifth column and the sixth column present the precision and recall of HeeNAMA, respectively. The precision of SLP-Core and Eclipse is presented in the last two columns. For SLP-Core and Eclipse, recall is always equal to precision, and thus it is omitted from the table.

From Table 10, we make the following observations:

- First, nested member accesses are popular. On average, nested member accesses account for a significant proportion (25.89 percent) of all member accesses. Consequently, applying HeeNAMA to such member accesses would make it more general and therefore also much stronger.
- Second, HeeNAMA is precise in suggesting nested member accesses. It achieves a precision of 71.64 percent, which is much higher than SLP-Core and Eclipse.
- Third, the recall of HeeNAMA is higher than that of Eclipse but lower than that of SLP-Core. One possible reason is that the filter of HeeNAMA improves precision at the cost of reduced recall.

From the analysis in the preceding paragraph, we conclude that HeeNAMA can be extended to recommend nested member accesses, which improves the practical usefulness of HeeNAMA.

TABLE 8
Impact of the Features Leveraged by the Filter

Settings	Performance of HeeNAMA	
	Precision	Recall
Default	83.36%	61.16%
Disabling <i>hurs</i>	75.16%	59.06%
Disabling <i>sim</i>	73.34%	56.05%
Disabling <i>cNum</i>	80.95%	60.86%
Disabling all	69.97%	51.28%

TABLE 9
Impacts of Heuristics

Applications	Default		Activating H_1		Activating H_2		Activating H_3	
	Precision	Recall	Precision	Recall	Precision	Recall	Precision	Recall
Ant	80.82%	63.59%	87.29%	35.97%	42.96%	41.18%	28.48%	28.35%
Batik	75.46%	60.86%	82.48%	39.90%	40.38%	38.52%	25.97%	25.71%
Cassandra	81.00%	59.00%	83.79%	42.60%	45.93%	41.38%	29.25%	29.14%
Log4J	84.00%	67.31%	85.76%	55.11%	51.48%	46.11%	27.76%	27.73%
Lucene-solr	86.00%	62.60%	87.52%	50.67%	52.83%	47.49%	29.56%	29.39%
Maven2	87.03%	67.04%	76.63%	35.08%	53.02%	46.70%	39.65%	38.98%
Maven3	73.89%	64.53%	69.01%	33.35%	56.64%	52.40%	40.82%	40.49%
Xalan-J	79.77%	53.06%	79.68%	38.81%	35.56%	34.95%	26.33%	26.29%
Xerces	78.47%	47.90%	66.86%	42.41%	26.48%	25.83%	32.32%	32.30%
Average	83.36%	61.16%	84.51%	47.04%	48.37%	44.10%	29.58%	29.44%

TABLE 10
Performance on Nested Member Accesses

Applications	All Accesses (N_{all})	Nested Accesses (N_{nested})	$\frac{N_{nested}}{N_{all}}$	HeeNAMA		SLP-Core	Eclipse
				Precision	Recall		
Ant	23,702	5,329	22.48%	75.89%	32.01%	48.13%	19.23%
Batik	24,031	3,997	16.63%	70.15%	17.99%	28.72%	11.88%
Cassandra	100,575	29,143	28.98%	75.93%	36.37%	43.50%	33.17%
Log4J	31,927	9,626	30.15%	70.59%	33.01%	46.88%	17.52%
Lucene-solr	266,040	69,924	26.28%	69.38%	43.26%	51.47%	29.96%
Maven2	10,070	3,045	30.24%	75.05%	40.69%	51.17%	23.84%
Maven3	18,067	5,912	32.72%	76.13%	46.33%	49.81%	14.09%
Xalan-J	22,621	3,685	16.29%	75.22%	21.17%	40.87%	15.25%
Xerces	24,719	4,440	17.96%	76.17%	36.64%	44.21%	29.73%
Total	521,752	135,101	25.89%	71.64%	39.11%	48.01%	27.57%

4.3.6 RQ6: Comparison Against API-Specific Approaches Trained on Within-Project Code

To address RQ6, we compare HeeNAMA against CSCC on subject applications. Notably, we incrementally train CSCC with within-project member accesses in the evaluation, i.e., trained with member accesses that have been recommended before the current one in the test project. The evaluation results are presented in Table 11. In the table, the first column presents the names of subject applications. The second to seventh columns present the precision and recall of HeeNAMA at the top k recommendation, respectively. The last six columns present the precision and recall of CSCC at the top k recommendation, respectively.

From the table, we make the following observations:

- First, CSCC works well on project-specific API member accesses. It achieves a high precision of 64.40 percent at top 1, 77.01 percent at top 3 and 79.48 percent at top 5 recommendation.
- Second, HeeNAMA is significantly more precise than CSCC in predicting project-specific API member accesses. On each application, its precision at top k is always higher than the precision of CSCC. For example, at top 1 recommendation, it improves precision by $29.44\% = (83.36\% - 64.40\%) / 64.40\%$.

- Third, HeeNAMA achieves higher recall than CSCC. The recall at top 1, 3 and 5 recommendation is improved by $20.46\% = (61.16\% - 50.77\%) / 50.77\%$, $10.58\% = (67.12\% - 60.70\%) / 60.70\%$ and $8.72\% = (68.11\% - 62.65\%) / 62.65\%$, respectively.

From the analysis in the preceding paragraph, we conclude that HeeNAMA significantly outperforms API-specific approaches in suggesting project-specific API member accesses even if they are trained on within-project code.

4.3.7 RQ7: Performance of HeeNAMA When Trained With All API Member Accesses

To answer RQ7, we train HeeNAMA with all API member accesses and evaluate it on project-specific and non-project-specific (i.e., public API) member accesses separately. On project-specific member accesses, the precision and recall of HeeNAMA are 81.28 and 64.85 percent, respectively. On non-project-specific member accesses, the precision and recall of HeeNAMA are 86.00 and 57.67 percent, respectively. The performance of HeeNAMA on non-project-specific member accesses is comparable to that on project-specific accesses because HeeNAMA takes non-project-specific member accesses as project-specific ones in fact. The performance on non-project-specific member accesses is not significantly higher than that on project-specific accesses

TABLE 11
Comparison Against API-Specific Approach

Applications	HeeNAMA						CSCC					
	Precision			Recall			Precision			Recall		
	Top 1	Top 3	Top 5	Top 1	Top 3	Top 5	Top 1	Top 3	Top 5	Top 1	Top 3	Top 5
Ant	80.82%	85.17%	85.92%	63.59%	69.91%	71.44%	57.20%	67.20%	70.58%	39.85%	46.82%	49.18%
Batik	75.46%	83.53%	85.90%	60.86%	68.94%	71.56%	61.44%	71.76%	73.59%	42.26%	49.36%	50.62%
Cassandra	81.00%	84.96%	84.63%	59.00%	62.97%	63.48%	58.06%	70.09%	72.08%	42.19%	50.93%	52.38%
Log4J	84.00%	87.32%	88.02%	67.31%	79.04%	79.99%	69.27%	81.38%	84.32%	55.51%	65.22%	67.58%
Lucene-solr	86.00%	88.05%	87.94%	62.60%	67.05%	67.94%	68.13%	81.47%	84.04%	57.40%	68.63%	70.80%
Maven2	87.03%	91.92%	92.51%	67.04%	74.20%	75.27%	56.29%	64.33%	66.29%	38.70%	44.22%	45.57%
Maven3	73.89%	85.12%	85.46%	64.53%	72.27%	73.50%	51.29%	58.98%	60.81%	34.62%	39.81%	41.05%
Xalan-J	79.77%	79.57%	80.02%	53.06%	66.18%	67.96%	51.62%	65.34%	68.17%	39.50%	50.00%	52.17%
Xerces	78.47%	80.72%	80.69%	47.90%	59.32%	60.31%	57.24%	69.49%	71.63%	37.74%	45.81%	47.22%
Average	83.36%	86.39%	86.51%	61.16%	67.12%	68.11%	64.40%	77.01%	79.48%	50.77%	60.70%	62.65%

TABLE 12
Performance on Project-Specific Member Accesses in the New Dataset

Applications	All Accesses (N_{all})	Accesses on RHS (N_{RHS})	$\frac{N_{RHS}}{N_{all}}$	HeeNAMA		SLP-Core	Eclipse	CSCC	
				Precision	Recall			Precision	Recall
Atlas	11,466	2,062	17.98%	81.11%	65.81%	44.76%	26.72%	53.12%	35.94%
Carbondata	30,718	6,416	20.89%	81.31%	65.96%	40.10%	44.08%	58.41%	45.46%
Fluo	7,263	1,234	16.99%	86.72%	62.97%	49.92%	52.84%	70.01%	52.59%
Giraph	13,028	2,200	16.89%	82.95%	49.09%	46.68%	39.91%	61.10%	32.27%
Ignite	284,169	53,702	18.90%	83.85%	54.34%	45.83%	30.71%	57.66%	47.70%
Johnzon	4,025	765	19.01%	79.20%	46.80%	53.20%	34.90%	61.12%	51.37%
Nifi	107,159	22,751	21.23%	79.43%	60.04%	41.84%	33.42%	68.26%	44.18%
Plc4x	9,189	1,614	17.56%	84.40%	65.37%	57.43%	48.39%	65.87%	37.67%
Streams	16,370	3,268	19.96%	38.76%	36.66%	33.51%	25.46%	40.79%	33.14%
Total	483,387	94,012	19.45%	80.31%	56.27%	44.36%	32.85%	59.52%	45.49%

because HeeNAMA does not leverage any unique properties of non-project-specific APIs, e.g., patterns of API usage.

4.3.8 RQ8: Performance on the New Dataset

To answer RQ8, we evaluate HeeNAMA against SLP-Core, Eclipse and CSCC on nine open-source Java applications that are recently created. The evaluation results are presented in Table 12. In the table, the first column presents the names of subject applications. The second column presents the number of all project-specific member accesses in subject applications. The third column and the forth column present the number of member accesses on RHS and the ratio of them to all project-specific member accesses, respectively. The fifth column and the sixth column present the precision and recall of HeeNAMA, respectively. The precision of SLP-Core and Eclipse is presented in the seventh and eighth columns, respectively. The last two columns present the precision and recall of CSCC. For SLP-Core and Eclipse, recall is always equal to precision, and thus it is omitted from the table.

From Table 12, we make the following observations:

- First, the ratio of project-specific member accesses on RHS in the new dataset is close to that of the original dataset (19.45 versus 19.87 percent).
- Second, the performance of HeeNAMA on the new dataset is comparable to its default performance on the original dataset. The precision and recall of HeeNAMA on the new dataset are 80.31 and 56.27 percent, respectively.
- Third, HeeNAMA significantly outperforms SLP-Core, Eclipse and CSCC on the new dataset. It improves the precision by $34.93\% = (80.31\% - 59.52\%) / 59.52\%$ and the recall by $23.70\% = (56.27\% - 45.49\%) / 45.49\%$, respectively.

4.4 Threats to Validity

A threat to the external validity is that only nine applications are involved in the evaluation and such applications may be unrepresentative. Consequently, the evaluation results may not hold if other subject applications are involved. To reduce the threat, we reuse the subject applications that have been

successfully employed in related work, and such applications contain more than one hundred thousand training items that have been used to evaluate HeeNAMA. Another threat to the external validity is that HeeNAMA is only evaluated on Java applications. Conclusions on Java applications may not hold for applications written in other languages, e.g., C++.

A threat to construct validity is that the evaluation is based on the assumption: the involved assignments in the subject applications are correct. We evaluate the suggested member access against that chosen by the original developers (i.e., the member access in the downloaded source code). If they are identical, we say the prediction is correct. Otherwise, it is declared incorrect. However, it is likely that the original developers may have chosen incorrect member access (and thus caused a bug), which makes the evaluation potentially incorrect. To reduce the threat, we select mature and well-known applications for evaluation because such applications are likely to contain fewer bugs.

A threat to internal validity is that the simulated code completion scenarios may be different from real scenarios. In the evaluation, we simulate the scenarios where source code in each document is typed in from the top to the bottom. In other words, context for code completion is the source code before the current cursor (where the suggested token will be inserted). However, in real world, especially in software maintenance and bug fixing phases, there may exist source code after the current cursor, and such code could be exploited for code completion as well. Another threat to internal validity is that in the simulated scenarios Java source code files (*.java) are created in the alphabetical order of the file names which may not be the real case. The creation order may influence the performance of code completion because it exploits existing code within the enclosing project. We take such an order because we fail to find their creation time from the web where the source code is downloaded.

5 CONCLUSIONS AND FUTURE WORK

In this paper we highlight the necessity of code completion for project-specific API member access on the right hand side of assignment. We also propose an automatic and accurate approach to suggesting the next member access whenever a project-specific API instance is followed by a dot on the right hand side of an assignment. The approach is accurate because it takes full advantages of the context of the code completion, including the type of the left hand side expression of the assignment, the identifier on the left hand side, the type of the base instance, and similar assignments typed in before. It also employs a neural network to filter out risky prediction, which guarantees high precision of code completion. HeeNAMA has been evaluated on nine open-source applications. Our evaluation results suggest that compared to the state-of-the-art approach and the state-of-the-practice tool HeeNAMA improves both precision and recall significantly.

Findings presented in the paper are valuable to the research community in the following aspects:

- First, we empirically reveal that public API member accesses are less popular than project-specific API

member accesses (39 versus 61 percent), which is consistent with the conclusion of a recent empirical study conducted on C# repositories [22]. Considering that most of the related work (as introduced in Section 2) focuses on APIs, this finding may help researchers identify better target scenarios for their research: focusing on project-specific API member accesses could be more fruitful and more useful.

- Second, we empirically reveal that applying API specific approaches to suggest project-specific API member accesses without significant adaption often results in inaccuracy. This finding may serve as an open call for automatic code completion approaches on project-specific APIs.
- Third, by focusing on a special case of code completion (project-specific API member accesses on the right hand side of assignments), the proposed approach significantly outperforms existing generic ones on this special case. The results may inspire future research on highly accurate code completion by focusing on special cases, like suggestion of parameters, recommendation of declarations, and completion of conditional statements.

A limitation of HeeNAMA is that it applies only to a specific but common case (accounting for 19.87 percent of project-specific API member accesses): suggesting the following member access whenever a project-specific API instance is followed by a dot on the right hand side of an assignment. Although we extend HeeNAMA to recommend project-specific API member accesses nested in method invocations (accounting for 25.89 percent of project-specific API member accesses) in Section 4.3.5, there are still about 52 percent project-specific API member accesses that HeeNAMA cannot recommend. Recommendation for such cases can be more challenging since little context information could be leveraged to make predictions. For example, when the developer types a dot after a base instance at the first line of a method, we can hardly extract any useful information within the method body. However, in the future, considering all kinds of project-specific API member accesses would make HeeNAMA much stronger.

Future work is needed to evaluate HeeNAMA further. More subject applications should be involved in further evaluation. Evaluation of HeeNAMA on other programming languages such as C++ and C# should also be involved in the future work. HeeNAMA should also be evaluated in real scenarios, and get feedback from developers. The final target of code completion is to facilitate coding. Consequently, it is critical for the success of HeeNAMA that developers are willing to use it in practice. In the future, HeeNAMA could also be combined with API-specific techniques (e.g., CSCC) to support both public API and project-specific API member access recommendations.

ACKNOWLEDGMENT

The work was supported by the National Natural Science Foundation of China (61690205, 61772071), and the National Key Research and Development Program of China (2016YFB1000801).

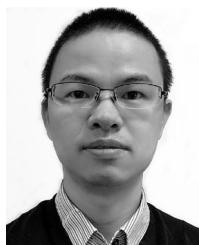
REFERENCES

- [1] R. Robbes and M. Lanza, "How program history can improve code completion," in *Proc. 23rd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2008, pp. 317–326. [Online]. Available: <http://dx.doi.org/10.1109/ASE.2008.42>
- [2] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 213–222. [Online]. Available: <http://doi.acm.org/10.1145/1595696.1595728>
- [3] G. C. Murphy, M. Kersten, and L. Findlater, "How are Java software developers using the eclipse IDE?" *IEEE Softw.*, vol. 23, no. 4, pp. 76–83, Jul./Aug. 2006. [Online]. Available: <http://dx.doi.org/10.1109/MS.2006.105>
- [4] D. Hou and D. M. Pletcher, "An evaluation of the strategies of sorting, filtering, and grouping API methods for code completion," in *Proc. 27th IEEE Int. Conf. Softw. Maintenance*, 2011, pp. 233–242.
- [5] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Learning natural coding conventions," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 281–293. [Online]. Available: <http://doi.acm.org/10.1145/2635868.2635883>
- [6] S. Han, D. R. Wallace, and R. C. Miller, "Code completion from abbreviated input," in *Proc. IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2009, pp. 332–343.
- [7] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 837–847. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2337223.2337322>
- [8] T. Gvero, V. Kuncak, I. Kuraj, and R. Piskac, "Complete completion using types and weights," in *Proc. 34th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2013, pp. 27–38. [Online]. Available: <http://doi.acm.org/10.1145/2491956.2462192>
- [9] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo, "Nomen est Omen: Exploring and exploiting similarities between argument and parameter names," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 1063–1073. [Online]. Available: <http://doi.acm.org/10.1145/2884781.2884841>
- [10] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: Helping to navigate the API jungle," in *Proc. ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2005, pp. 48–61. [Online]. Available: <http://doi.acm.org/10.1145/1065010.1065018>
- [11] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: Current status and future directions," *Data Mining Knowl. Discov.*, vol. 15, no. 1, pp. 55–86, Aug. 2007. [Online]. Available: <http://dx.doi.org/10.1007/s10618-006-0059-1>
- [12] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "CSCC: Simple, efficient, context sensitive code completion," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 71–80.
- [13] S. Proksch, J. Lerch, and M. Mezini, "Intelligent code completion with Bayesian networks," *ACM Trans. Softw. Eng. Methodol.*, vol. 25, no. 1, pp. 3:1–3:31, Dec. 2015. [Online]. Available: <http://doi.acm.org/10.1145/2744200>
- [14] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," in *Proc. 23rd Eur. Conf. Object-Oriented Program.*, 2009, pp. 318–343. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-03013-0_15
- [15] A. T. Nguyen et al., "Graph-based pattern-oriented, context-sensitive source code completion," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 69–79. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2337223.2337232>
- [16] C. D. Manning and H. Schütze, *Foundations of Statistical Natural Language Processing*. Cambridge, MA, USA: MIT Press, 1999.
- [17] M. Allamanis and C. Sutton, "Mining source code repositories at massive scale using language modeling," in *Proc. 10th Work. Conf. Mining Softw. Repositories*, 2013, pp. 207–216.
- [18] T. T. Nguyen, A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A statistical semantic language model for source code," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, pp. 532–542. [Online]. Available: <http://doi.acm.org/10.1145/2491411.2491458>
- [19] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 419–428. [Online]. Available: <http://doi.acm.org/10.1145/2594291.2594321>
- [20] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 763–773. [Online]. Available: <http://doi.acm.org/10.1145/3106237.3106290>
- [21] C. Franks, Z. Tu, P. Devanbu, and V. Hellendoorn, "CACHECA: A cache language model based code suggestion tool," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 705–708. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2819009.2819143>
- [22] V. J. Hellendoorn, S. Proksch, H. C. Gall, and A. Bacchelli, "When code completion fails: A case study on real-world completions," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 960–970.
- [23] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 269–280. [Online]. Available: <http://doi.acm.org/10.1145/2635868.2635875>
- [24] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Commun. ACM*, vol. 59, no. 5, pp. 122–131, Apr. 2016. [Online]. Available: <http://doi.acm.org/10.1145/2902362>
- [25] Z. C. Lipton, "A critical review of recurrent neural networks for sequence learning," *CoRR*, vol. abs/1506.00019, 2015. [Online]. Available: <http://arxiv.org/abs/1506.00019>
- [26] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, Nov. 1997. [Online]. Available: <http://dx.doi.org/10.1162/neco.1997.9.8.1735>
- [27] M. White, C. Vendome, M. Linares-Vásquez, and D. Poshyvanyk, "Toward deep learning software repositories," in *Proc. 12th Work. Conf. Mining Softw. Repositories*, 2015, pp. 334–345. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2820518.2820559>
- [28] R. Karampatsis and C. A. Sutton, "Maybe deep neural networks are the best choice for modeling source code," *CoRR*, vol. abs/1903.05734, 2019. [Online]. Available: <http://arxiv.org/abs/1903.05734>
- [29] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 858–868. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2818754.2818858>
- [30] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen, "Graph-based mining of multiple object usage patterns," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2009, pp. 383–392. [Online]. Available: <http://doi.acm.org/10.1145/1595696.1595767>
- [31] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Learning API usages from bytecode: A statistical approach," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 416–427. [Online]. Available: <http://doi.acm.org/10.1145/2884781.2884873>
- [32] L. Rabiner and B. Juang, "An introduction to hidden Markov models," *IEEE ASSP Mag.*, vol. 3, no. 1, pp. 4–16, Jan. 1986.
- [33] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API patterns as partial orders from source code: From usage scenarios to specifications," in *Proc. 6th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng.*, 2007, pp. 25–34. [Online]. Available: <http://doi.acm.org/10.1145/1287624.1287630>
- [34] T. Cover and P. Hart, "Nearest neighbor pattern classification," *IEEE Trans. Inf. Theory*, vol. IT-13, no. 1, pp. 21–27, Jan. 1967.
- [35] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "A simple, efficient, context-sensitive approach for code completion," *J. Softw. Evol. Process*, vol. 28, no. 7, pp. 512–541, Jul. 2016. [Online]. Available: <https://doi.org/10.1002/smr.1791>
- [36] S. Butler, M. Wermelinger, Y. Yu, and H. Sharp, "Exploring the influence of identifier names on code quality: An empirical study," in *Proc. 14th Eur. Conf. Softw. Maintenance Reeng.*, 2010, pp. 156–165. [Online]. Available: <http://dx.doi.org/10.1109/CSMR.2010.27>
- [37] C. Caprile and P. Tonella, "Nomen est Omen: Analyzing the language of function identifiers," in *Proc. 6th Work. Conf. Reverse Eng.*, 1999, pp. 112–122.
- [38] B. Caprile and P. Tonella, "Restructuring program identifier names," in *Proc. Int. Conf. Softw. Maintenance*, 2000, pp. 97–107. [Online]. Available: <http://dl.acm.org/citation.cfm?id=850948.853439>
- [39] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, "What's in a name? A study of identifiers," in *Proc. 14th IEEE Int. Conf. Program Comprehension*, 2006, pp. 3–12.
- [40] M. Pradel and T. R. Gross, "Detecting anomalies in the order of equally-typed method arguments," in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 232–242. [Online]. Available: <http://doi.acm.org/10.1145/2001420.2001448>

- [41] M. Pradel and T. R. Gross, "Name-based analysis of equally typed method arguments," *IEEE Trans. Softw. Eng.*, vol. 39, no. 8, pp. 1127–1143, Aug. 2013. [Online]. Available: <http://dx.doi.org/10.1109/TSE.2013.7>
- [42] W. W. Cohen, P. Ravikumar, and S. E. Fienberg, "A comparison of string distance metrics for name-matching tasks," in *Proc. Int. Conf. Inf. Integr. Web*, 2003, pp. 73–78. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3104278.3104293>
- [43] A. T. Nguyen, H. A. Nguyen, and T. N. Nguyen, "A large-scale study on repetitiveness, containment, and compositability of routines in open-source projects," in *Proc. 13th Int. Conf. Mining Softw. Repositories*, 2016, pp. 362–373. [Online]. Available: <http://doi.acm.org/10.1145/2901739.2901759>
- [44] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, vol. abs/1301.3781, 2013. [Online]. Available: <http://arxiv.org/abs/1301.3781>
- [45] T. D. Nguyen, A. T. Nguyen, H. D. Phan, and T. N. Nguyen, "Exploring API embedding for API usages and applications," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 438–449. [Online]. Available: <https://doi.org/10.1109/ICSE.2017.47>
- [46] V. Raychev, P. Bielik, M. Vechev, and A. Krause, "Learning programs from noisy data," in *Proc. 43rd Annu. ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 2016, pp. 761–774. [Online]. Available: <http://doi.acm.org/10.1145/2837614.2837671>
- [47] P. Bielik, V. Raychev, and M. Vechev, "PHOG: Probabilistic model for code," in *Proc. 33rd Int. Conf. Int. Conf. Mach. Learn.*, 2016, pp. 2933–2942. [Online]. Available: <http://dl.acm.org/citation.cfm?id=3045390.3045699>
- [48] D.-A. Clevert, T. Unterthiner, and S. Hochreiter, "Fast and accurate deep network learning by exponential linear units (ELUs)," *Computerence*, 2015.
- [49] T. Dozat, "Incorporating Nesterov momentum into Adam," in *Proc. 4th Int. Conf. Learn. Representations*, 2016.
- [50] CC-CG, "Cc-CG/HeeNAMA v1.0," Dec. 2019. [Online]. Available: <https://doi.org/10.5281/zenodo.3559330>
- [51] A. Maratea, A. Petrosino, and M. Manzo, "Adjusted F-measure and kernel scaling for imbalanced data learning," *Inf. Sci.*, vol. 257, pp. 331–341, 2014. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0020025513003137>



Lin Jiang received the BS degree from the College of Information and Electrical Engineering, China Agricultural University, Beijing, China, in 2016. She is currently working toward the PhD degree in the School of Computer Science and Technology, Beijing Institute of Technology, Beijing, China, under the supervision of Dr. H. Mei. She is interested in software evolution and computer programming.



Hui Liu received the BS degree in control science from Shandong University, Jinan, China, in 2001, the MS degree in computer science from Shanghai University, Shanghai, China, in 2004, and the PhD degree in computer science from Peking University, Beijing, China, in 2008. He is a professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. He was a visiting research fellow with the Centre for Research on Evolution, Search and Testing (CREST), University College London, United Kingdom. He served on the program committees and organizing committees of prestigious conferences, such as ICSME and RE. He is particularly interested in software refactoring, software evolution, and software quality. He is also interested in developing practical tools to assist software engineers.



He Jiang (Member, IEEE) received the PhD degree in computer science from the University of Science and Technology of China, Hefei, China. He is currently a professor with the Dalian University of Technology and an adjunct professor with the Beijing Institute of Technology. He is also a member of the ACM and the CCF (China Computer Federation). He is one of the ten supervisors for the Outstanding Doctoral Dissertation of the CCF in 2014. His current research interests include search-based software engineering (SBSE) and mining software repositories (MSR). His work has been published at premier venues like ICSE, SANER, and GECCO, as well as in major IEEE transactions like the *IEEE Transactions on Software Engineering*, the *IEEE Transactions on Knowledge and Data Engineering*, the *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, the *IEEE Transactions on Cybernetics*, and the *IEEE Transactions on Services Computing*.



Lu Zhang received the BSc and PhD degrees in computer science from Peking University, Beijing, China, in 1995 and 2000, respectively. He is a professor with the School of Electronics Engineering and Computer Science, Peking University, P.R. China. He was a postdoctoral researcher with Oxford Brookes University and University of Liverpool, United Kingdom. He served on the program committees of many prestigious conferences, such as FSE, OOPSLA, ISSTA, and ASE. He was a program co-chair of SCAM2008 and a program co-chair of ICSM17. He has been on the editorial boards of the *Journal of Software Maintenance and Evolution: Research and Practice* and the *Software Testing, Verification and Reliability*. His current research interests include software testing and analysis, program comprehension, software maintenance and evolution, software reuse, and program synthesis.



Hong Mei (Fellow, IEEE) received the BA and MS degrees from the Nanjing University of Aeronautics and Astronautics, Nanjing, China, in 1984 and 1987, respectively, and the PhD degree in computer science from Shanghai Jiao Tong University, Shanghai, China, in 1992. From 1992 to 1994, he was a postdoctoral research fellow with Peking University. He is a professor with the Beijing Institute of Technology, Shanghai Jiao Tong University, and Peking University. He was the dean of the School of EECS, Peking University from 2006–2014, the vice president for research with Shanghai Jiao Tong University from 2013 to 2016, and had been the vice president for human resource and international affairs with the Beijing Institute of Technology since 2016. His research interests include software engineering and system software. He is a member of the Chinese Academy of Sciences, a fellow of the CCF and TWAS.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csl.

Deep Learning Based Program Generation From Requirements Text: Are We There Yet?

Hui Liu^{ID}, Mingzhu Shen, Jiaqi Zhu, Nan Niu^{ID}, Ge Li, and Lu Zhang^{ID}

Abstract—To release developers from time-consuming software development, many approaches have been proposed to generate source code automatically according to software requirements. With significant advances in deep learning and natural language processing, deep learning-based approaches are proposed to generate source code from natural language descriptions. The key insight is that given a large corpus of software requirements and their corresponding implementations, advanced deep learning techniques may learn how to translate software requirements into source code that fulfill such requirements. Although such approaches are reported to be highly accurate, they are evaluated on datasets that are rather small, lack of diversity, and significantly different from real-world software requirements. To this end, we build a large scale dataset that is composed of longer requirements as well as validated implementations. We evaluate the state-of-the-art approaches on this new dataset, and the results suggest that their performance on our dataset is significantly lower than that on existing datasets concerning the common metrics, i.e., BLEU. Evaluation results also suggest that the generated programs often contain syntactic and semantical errors, and none of them can pass even a single predefined test case. Further analysis reveals that the state-of-the-art approaches learn little from software requirements, and most of the successfully generated statements are popular statements in the training programs. Based on this finding, we propose a popularity-based approach that always generates the most popular statements in training programs regardless of the input (software requirements). Evaluation results suggest that none of the state-of-the-art approaches can outperform this simple statistics-based approach. As a conclusion, deep learning-based program generation requires significant improvement in the future, and our dataset may serve as a basis for future research in this direction.

Index Terms—Software requirements, code generation, deep learning, data set

1 INTRODUCTION

SOFTWARE development is the process of writing and maintaining source code according to software requirements [1]. The resulting source code is in turn compiled automatically into executable applications that finally fulfill the requirements. However, with the increase in software complexity, software development is often expensive and error-prone [1] although many engineering approaches have been proposed to guide the development.

To release human beings from challenging, time-consuming, and error-prone software development (especially coding), many approaches have been proposed to generate source code automatically. During the last twenty years of the twentieth century, researchers proposed mathematics-based formal methods [2], [3] and tools [4], [5] to generate source code automatically according to formal specifications [6], [7]. Although formal methods are highly reliable, it remains

challenging to create formal specifications that should be described in formal languages, e.g., Z [8]. Consequently, researchers turn to less formal approaches, e.g., Model Driven Architecture (MDA) [9]. MDA attempts to generate source code [10] according to models described in modeling languages, e.g., Unified Modelling Language (UML) [11]. UML is a graphical modelling language, and shares most of the concepts with object-oriented programming languages. Consequently, developers are willing to use UML compared to formal languages. However, because UML models usually focus on the architecture of the system under development, it is quite often that MDA generates nothing but sketch (e.g., signatures of methods) of the system. Detailed implementation, especially the body of methods, still has to be typed in manually in most cases. Extending UML with action semantics [12] makes it possible to present more detailed semantics in UML models, and thus we may generate more complete source code from UML models. However, they often employ DSMLs instead of general-purpose languages. It is also challenging and time-consuming to construct action semantics with the extended UML.

With significant advances in deep learning, researchers are turning to learning-based approaches to generate source code. The key insight of such approaches is that given a corpus of software requirements and their corresponding implementation (source code), advanced deep learning techniques may learn how to translate software requirements into source code that fulfill such requirements [13], [14]. Existing approaches have successfully generated source code from software requirements in natural language descriptions [15], [16],

- Hui Liu, Mingzhu Shen, and Jiaqi Zhu are with the School of Computer Science and Technology, Beijing Institute of Technology, Beijing 100081, China. E-mail: {liuhui08, 3120181025, zhujiaqi}@bit.edu.cn.
- Nan Niu is with the Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, OH 45221 USA. E-mail: nan.niu@uc.edu.
- Ge Li and Lu Zhang are with the Key Laboratory of High Confidence Software Technologies, Ministry of Education, Peking University, Beijing 100871, China. E-mail: lige@pku.edu.cn, zhanglu@sei.pku.edu.cn.

Manuscript received 27 Apr. 2020; revised 31 July 2020; accepted 18 Aug. 2020. Date of publication 21 Aug. 2020; date of current version 18 Apr. 2022. (Corresponding author: Hui Liu.) Recommended for acceptance by H. Rajan. Digital Object Identifier no. 10.1109/TSE.2020.3018481

images of Graphical User Interface (GUI) [17], [18], and input-output examples [19], [20]. In this paper, we focus on natural language requirements (requirements text) because in most cases software requirements in the industry are described in natural languages [21]. Deep learning-based code generation approaches have been reported to be highly accurate. For example, the syntactic neural model (SNM) proposed by Yin and Neubig [22] reaches a high Bilingual Evaluation Understudy (BLEU) [23] (0.845) in translating natural language descriptions into Python programs. In contrast, Google neural machine translation (GNMT), the widely used state-of-the-practice language translator, results in much smaller BLEU (0.4) in translating English into French [24]. For example, if the reference translation is “*I know tomorrow is another day*” and the generated translation is “*I know tomorrow is a new day*”, the resulting BLEU is 0.42.

However, such deep learning-based code generation approaches have not yet been extensively evaluated, which prevents us from knowing the state of the art. The reported evaluation of such approaches is often conducted on datasets that are rather small, lack of diversity, and significantly different from real-world software requirements. For example, the widely used dataset *HS* [14] is composed of source code (and the associated ‘requirements’) from a single software project, which results in poor diversity. The average length of the source code in dataset *Django* [25] is 33 characters only, suggesting that the software programs in the dataset have very limited complexity. Natural language descriptions in dataset *CoNaLa* [16] are how-to questions automatically extracted from *Stack Overflow*, instead of real-world software requirements. As a result, evaluating existing approaches on such datasets may fail to reveal the state of the art.

To this end, in this paper, we build a large scale dataset and evaluate deep learning-based code generation approaches on it. Compared to existing datasets, our dataset is composed of longer and more comprehensive software requirements accompanied by their validated implementations (source code). We also develop an assisting tool to assess comprehensively the quality of the generated source code instead of simply counting the lexical similarity between the generated source code and reference implementations. The benefits of this dataset and its assisting tool are twofold. On one side, we can reassess the state of the art of deep learning-based code generation with the resulting dataset and assisting tool. On the other side, the resulting data set may serve as a publicly available training/testing dataset for future research in code generation. Lacking of large scale and high-quality datasets is preventing deep learning-based code generation approaches from reaching their maximal potential. The resulting dataset is an initial attempt to solve this problem.

We reassess the state of the art in code generation with our new dataset. Evaluation results suggest that the performance of the state-of-the-art approaches on our dataset is significantly lower than that on existing datasets. The programs generated by such approaches are significantly different from reference implementations, often contain syntactic and semantical errors, and fail to pass even a single predefined test case. We replace the input (requirements) with random noise, and the performance of the evaluated approaches is still comparable to that before the replacement. It may suggest that the evaluated approaches learn little from software

requirements. Further analysis of the generated source code suggests that most of the successfully generated statements are popular statements in the training programs. Based on this finding, we propose a popularity-based approach that always generates the most popular statements in the training programs regardless of the input (software requirements). Evaluation results suggest that none of the state-of-the-art approaches can outperform this simple statistics-based approach.

The paper makes the following contributions:

- A large scale dataset for learning-based code generation. Compared to existing ones, it is larger and has improved diversity as well as validated programs. Besides that, the requirements in the dataset are longer than those in existing datasets.
- An assisting tool kit to assess the quality of generated programs. Unlike existing approaches that heavily rely on lexical similarity, the tool kit employs static syntactic checking, dynamic cross-validation, and lexical comparison to comprehensively assess the quality of generated source code.
- A comprehensive reassessment of the state of the art of deep learning-based code generation. Based on the new dataset and associated tool kit, we evaluate the state-of-the-art approaches. Evaluation results suggest that the generated programs are significantly different from references, and none of them can pass even a single test case associated with the dataset. It may suggest that deep learning-based program generation requires significant improvement in the future. Our dataset, as well as the assisting tool, may serve as a basis for future research in this direction.

The rest of the paper is structured as follows. Section 2 introduces related research. Section 3 introduces how we construct a new dataset. Section 4 specifies how we assess the state-of-the-art approaches on the resulting dataset whereas Section 5 presents the results. Section 6 discusses related issues. Section 7 makes conclusions.

2 RELATED WORK

2.1 Generating Source Code From Requirements Text

As introduced in the preceding section, automatic generation of source code from requirements text has recently been a hot topic in both software engineering and artificial intelligence communities. To reduce the complexity of code generation, researchers try to limit the complexity of programming languages. As a result, many code generation approaches employ domain-specific languages (DSLs) to describe the generated source code [26], [27], [28], [29]. DSLs are much simpler than general-purpose programming languages, and thus DSL-based approaches often result in high accuracy in generating source code. However, DSLs are specific to predefined domains, and it is challenging to apply them to other domains [14].

Compared to DSLs, generating source code in general-purpose programming languages is more challenging. However, employing such languages results in a number of significant advantages [14]. First, such languages, e.g., Java,

are well-known and widely used by developers, and thus developers can read and modify the generated source code expediently. Second, such languages are broadly applicable across domains. Third, such programming languages have better expression ability than DSLs, and thus could describe complex applications. Because of such significant advantages, researchers have proposed a number of approaches to generating source code in general-purpose programming languages [14], [22], [29], [30]. Ling *et al.* [14] propose a latent predictor network-based approach (called LPN) to generate source code in Python or Java. Evaluation results on MTG, HS, and Django suggest that the approach is accurate and the average BLEU is up to 0.776. To the best of our knowledge, it is the first one that generates source code in general-purpose programming languages. Yin and Neubig [22] propose a syntactic neural model (SNM), which for the first time leverages the syntax of target language as prior knowledge. Later, they propose TRANX that generalizes SNM from Python to other languages [31]. Rabinovich *et al.* [29] propose an abstract syntax network-based approach (called ASN). To the best of our knowledge, they are the first to employ multiple decoders in code generation, where different types of elements in abstract syntax trees are generated by different decoders. Stehnii [30] proposes a *tree-to-tree* model for code generation. The key insight of the approach is that requirements in English could be parsed into trees as well, and the parsing can help neural networks to better understand the requirements. Dong and Lapata [32] propose a structure-aware neural architecture (called *Coarse-to-Fine*) for code generation. They are the first to divide the decoding process of code generation into two stages: generating a sketch on the first stage, and generating other information (e.g., variables and parameters) on the second stage. Hayati *et al.* [33] propose an approach called ReCode. For the first time, they leverage the nearest neighbors for code generation. The key insight of the approach is that two highly similar requirements are likely to result in highly similar implementations. GrammerCNN [34] proposed by Sun *et al.* is the first to employ CNN-based decoders in code generation. Their evaluation results suggest that their approach improves the state of the art by five percentage.

Text-based code generation is also a hot topic in the software engineering community. Gvero and Kuncak [35] propose an approach, called *anyCode*, to synthesize Java expressions from free-form queries containing a mixture of English and Java. The purpose of *anyCode* is to help developers, especially new developers, to achieve a task of interest by leveraging related APIs. For example, the developer may type in “*copy file fname to bname*” where *fname* and *bname* are given file names. *AnyCode* would return Java expressions like “*FileUtils.copyWith(new File(fname), new File(bname))*”. For a given query, *anyCode* selects a set of most likely API declarations according to the query and unigram models. After that, *anyCode* leverages probabilistic context free grammar and unigram model to unfold the declaration arguments of the selected APIs. Raghothaman *et al.* [36] propose another approach, called SWIM, to generate code snippets for given API-related natural language queries such as “*generate md5 hash code*”. Different from *anyCode* that generates a single expression, SWIM can generate a code snippet containing a few statements. SWIM maps textual query into a set of APIs

by leveraging a statistical model. To construct code snippets from the suggested APIs, SWIM collects structural call sequences for each API data type in projects on Github. From such pre-extracted call sequences, SWIM retrieves the one that is most similar to the suggested APIs based on cosine similarity. *T2API* proposed by Nguyen *et al.* [37] is also a statistics-based approach to synthesize API code snippets from textual queries. It differs from SWIM in the following aspects. First, it conducts context expansion to expand the related APIs. For example, if *Socket.open()* is in the initial set of APIs associated with the given query, *T2API* will add *Socket.close()* as related APIs as well because it frequently follows *Socket.open()*. Second, *T2API* presents code snippets as graphs, and generates code graphs instead of retrieving graphs/code snippets from a given library. Consequently, it may generate new API usages. Yan *et al.* [38] build a dataset and its associated tools for fair and convenient comparison among different query-based code search methods. Such approaches differ from the evaluated approaches in that they are often confined to APIs [35], [36], [37] and generate short code snippets (or even a single expression) only. As a result, they are not suitable for our scenario and thus they are not involved in the evaluation in Section 4.

2.2 Datasets for Code Generation From Requirements Text

It is well recognized that high-quality datasets are critical for learning-based code generation [14]. Consequently, researchers have built a number of datasets that contain textual description (requirements) as well as their implementations (source code). Table 1 presents an overview of existing datasets. The first column presents the names of the datasets. The second column presents a short explanation. The third column presents the program languages employed to describe the programs. The fourth and fifth columns present the numbers of software requirements and software programs, respectively. The last two columns present the average length of requirements and programs, respectively. Sample items (both requirements and their corresponding implementations) are presented in Table 2.

According to the programming languages involved in the datasets, existing datasets could be classified into two categories. The first category of datasets (i.e., ATIS, GEO, JOBS, and IFTTT) describes source code in domain-specific languages. ATIS [13] was initially built to evaluate air travel information systems. It is composed of database queries in English and the source code to accomplish the queries. Later, it is employed as a dataset for code generation [28] where the queries are taken as software requirements and the Lambda style source code is taken as reference implementation. GEO [39] and JOBS [15] are similar to ATIS [13] in that all of them are composed of database queries and their accomplishing source code in DSLs. IFTTT [40] is another DSL-based dataset. Source code (applets) within IFTTT follows a predefined pattern: IF *this* THEN *that*, and it is the reason why the dataset is called IFTTT. Such kind of applets are widely used to control devices (e.g., watches, smart phone, and lights). These DSL-based datasets together have significantly advanced research in code generation [28]. However, such datasets are domain-specific, and thus models trained on such datasets may not work in other domains.

TABLE 1
Existing Datasets for Requirements Text-Based Code Generation

Dataset	Explanation	Programming Language	#REQs	# Programs	Average Length of REQs (in tokens)	Average Length of Programs character/LOC
ATIS	database query of traveling info	Lambda-style DSL	5,373	5,373	11	64 / 1
JOBS	database query of jobs	Prolog-style DSL	640	640	9	52 / 1
GEO	database query of geography	Lambda-style DSL	880	880	7	45 / 1
IFTTT	IF-this-THEN-that applets	If-Then Recipes	86,960	86,960	7	62 / 1
MTG	features from Magic the Gathering	Java	13,297	13,297	58	981 / 31.4
HS	features from Hearthstone	Python	665	665	34	300 / 7.4
Django	pseudo-code vs. code	Python	18,805	18,805	14	33 / 1
StaQC	how-to questions	Python	1,441	2,169	10	247 / 10.1
		SQL	1,221	2,056	10	218 / 10.2
CoNaLa	how-to questions	Python	2,879	2,879	14	39 / 1.1

*REQs: Requirements.

The second category of datasets (i.e., Django, MTG, HS, StaQC, and CoNaLa) describes source code in general-purpose programming languages, e.g., Java and Python. Oda *et al.* [25] propose an automatic approach to generating pseudo-code from source code. They collect source code (Python statements) of a Python web framework called *Django* (available at <https://www.djangoproject.com/>), and generate pseudo-code automatically for each of the downloaded Python statements. Such Python statements accompanied by corresponding pseudo-code are later employed as code generation dataset [14], [22], [30], [30]. Different from *Django* that is a byproduct of a pseudo-code generation approach, MTG and HS are intentionally built for code generation [14]. MTG is built on a trading card game called *Magic the Gathering* [41]. Each item in MTG is composed of two parts: Textual description of a card (in English) and the source code associated with the card. HS is highly similar to MTG. The only difference is that HS is based on another card game called *Hearthstone* [42]. MTG and HS are frequently used in code generation tasks [14], [22], [29], [30]. Different from MTG and HS that are built on a given software project, StaQC [43] and CoNaLa [16] are created by mining QA forums (e.g., *Stack Overflow* [44]), i.e., extracting how-to questions and their code fragments in accepted answers.

Although such datasets employ general-purpose programming languages, they still have the following limitations:

- First, software requirements included in such datasets are essentially different from real ones in the industry. The ‘requirements’ in *Django* are pseudo-code that is highly similar to the associated source code. Such pseudo-code is significantly different from requirements text. Translating requirements text into source code is much more challenging than the translation from pseudo-code to source code. Although programs in MTG are longer than those in our dataset, the ‘requirements’ in MTG (and HS as well) are rather special: all of the ‘requirements’ in the dataset together

constitute the real complete requirements for a single application (*Magic the Gathering*). Consequently, models trained on MTG can generate only additional source code (i.e., expansion) for the given program. It is unlikely for them to generate programs that are irrelevant to the given program (*Magic the Gathering*). The ‘requirements’ in StaQC and CoNaLa are automatically extracted how-to questions that are significantly different from common requirements text.

- Second, the requirements and their associated source code may not match exactly. For example, the source code extracted automatically from QA forums may not exactly fulfill the how-to questions in StaQC and CoNaLa.
- Finally, as shown in Table 1, programs within such datasets are rather short. It may suggest that programs in such datasets are of limited complexity.

As a result of the limitations, models trained on such datasets may fail to generate complex implementation for real-world software requirements. Assessing the state of the art on such datasets also suffers from significant threats to external validity.

2.3 Code Generation Based On Examples and Contexts

Code complete is to generate code expressions or short code snippets based on contexts, e.g., the source code preceding the locations where the suggested code should be inserted. Type-based code complete is widely supported by IDEs. For example, while developers type in “*System.*”, IDEs like Eclipse would suggest a list of members (fields and methods) that could be accessed via *System*. More advanced approaches, like statistical language models, have been proposed to improve the accuracy of code complete [45], [46]. Such approaches are based on the assumption that source code, like natural languages, is likely to be repetitive and predictable [47]. To this end, the statistical language models, like n-gram, are employed to predict the next token or the next expression in code complete. Besides such generic

TABLE 2
Sample Requirements and Implementations From Existing Datasets

Dataset	Sample Requirements	Corresponding Implementations
ATIS	dallas to san francisco leaving after 4 in the afternoon please	(lambda \$0 e (and (>(departure time \$0) 1600:ti) (from \$0 dallas:ci) (to \$0 san francisco:ci)))
JOBS	what microsoft jobs do not require a bscs?	answer(company(J, 'microsoft'), job(J), not((req deg(J, 'bscs'))))
GEO	what is the population of the state with the largest area?	(population:i (argmax \$0 (state:t \$0) (area:i \$0)))
IFTTT	Turn on heater when temperature drops below 58 degree	TRIGGER:Weather - Current temperature drops below - ((Temperature (58)) (Degrees in (f))) ACTION: WeMo Insight Switch - Turn on - ((Which switch? ("")))
MTG	NAME: Mox Jet ATK: NIL DEF: NIL COST: 0 DUR: NIL TYPE: Artifact PLAYER_CLS: Limited Edition Alpha RACE: 262 RARITY: R TAP: Add B to your mana pool.	<pre> public class MoxJet extends CardImpl { public MoxJet(UUID ownerId) { super(ownerId, 262, "Mox Jet", Rarity.RARE, new CardType[]{CardType.ARTIFACT, "{0}"}); this.expansionSetCode = "LEA"; this.addAbility(new BlackManaAbility()); } public MoxJet(final MoxJet card) { super(card); } @Override public MoxJet copy() { return new MoxJet(this); } } </pre>
HS	NAME: Acidic Swamp Ooze ATK: 3 DEF: 2 COST: 2 DUR: -1 TYPE: Minion PLAYER_CLS: Neutral RACE: NIL RARITY: Common Battlecry: Destroy your opponent's weapon.	<pre> class AcidicSwampOoze(MinionCard): def __init__(self): super().__init__("Acidic Swamp Ooze", 2, CHARACTER_CLASS.ALL, CARD_RARITY.COMMON, battlecry=Battlecry(Destroy(), WeaponSelector(EnemyPlayer()))) def create_minion(self, player): return Minion(3, 2) </pre>
Django	call the function conf.copy, substitute it for params.	params = conf.copy()
StaQC	How to limit a number to be within a specified range?	def clamp(n, minn, maxn): return max(min(maxn, n), minn)
CoNaLa	How to convert a list of multiple integers into a single integer?	r = int("".join(map(str, x)))

code complete approaches, some task-specific approaches have been proposed successfully to suggest specific tokens, like method names [48], [49] and arguments [50].

Code generation is also closely related to program synthesis that generates programs automatically according to input/output examples [51], called *programming by examples* [52]. For example, researchers have successfully synthesized string editing programs according to input/output examples [53], [54]. Feng *et al.* [55] propose a component-based approach to synthesis scripts from examples for table consolidation and transformation tasks. Feng *et al.* [56] propose a conflict-driven program synthesis technique that learns from past mistakes. Lee *et al.* [57] accelerate search-

based program synthesis using learned probabilistic models. A systematic review of search-based program synthesis is available at [52]. Neural network-based program synthesis is one of the most promising directions in program synthesis [51], [58]. Balog *et al.* [59], however, propose *DeepCoder* that combines neural network-based program synthesis and search-based program synthesis. Their evaluation results suggest that the combination leads to an order of magnitude speedup over the Recurrent Neural Network approaches. The performance of such learning-based program synthesis approaches depends heavily on the performance of training data [58]. To improve the quality of training data, Shin *et al.* [58] propose an automatic approach

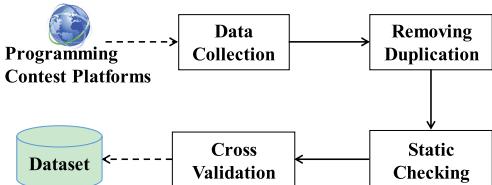


Fig. 1. Dataset creation.

to generate a high-quality dataset so that models trained on the resulting dataset could learn the full semantics of the selected DSL.

Representation of source code is also closely related to code generation [60]. The intuitive and straightforward representation of source code is to take it as natural language text (tokens) [47]. However, such plain text-based representation ignores the semantics of programs and the structures of source code. To this end, new approaches have been proposed to represent source code based on abstract syntax trees (AST) [60]. More advanced approaches can even leverage the paths within the AST trees [61], [62] and dependency among different source code elements [63].

3 NEW DATASET

As introduced in Section 2.2, existing datasets are preventing deep learning-based code generation approaches from reaching their maximal potential. To this end, in this section, we build a new dataset, as well as an assisting tool kit, for learning-based code generation.

3.1 Overview

Fig. 1 presents an overview on how we create the dataset for code generation. First, we extract task descriptions (software requirements) and their associated submissions from programming contest platforms. Second, we detect and remove duplicate tasks and duplicate implementations from the resulting dataset. Third, we compile the downloaded source code to make sure that the remaining source code is compilable. Third, we apply cross-validation to exclude incorrect implementations. Details of the creation are presented in the following sections.

3.2 Data Collection

We collect data from two programming contest platforms, i.e., Codeforces [64] and HackerEarth [65] because of the following reasons:

- First, the contests (software requirements) and their corresponding submissions (source code) on such platforms are publicly available;
- Second, the contests cover different topics instead of being confined to a specific domain, which may increase the diversity of the resulting dataset;
- Third, such platforms have manually designed test cases for each of the contests to ensure the correctness of the submissions, which may reduce the likelihood to include incorrect implementations in the resulting dataset;
- Finally, the contests are moderately challenging for automatic code generation. On one side, they are

much more complex than most of the existing datasets whose implementation is often composed of only a couple of lines. Consequently, compared against existing datasets, the resulting dataset is more complex. On the other side, such contests are intentionally designed for beginners, and thus the complexity is limited. The limited complexity makes it potentially practical for deep learning techniques to generate the source code automatically.

With a Python-based crawler, we collect programming tasks (in English) from the selected platforms. We also collect their submissions (implementations) that have passed all of the associated test cases. The submissions are described in different programming languages, e.g., Java, Python, and C/C++. Comments within the source code are removed automatically because we focus only on code generation in our current work. Notably, for a single contest, there are often a large number (hundreds) of submissions. We only download its first N submissions for each programming language. This number (empirically set to ten) is a result of the balance between the diversity of implementations and the size of the resulting dataset. We manually check the diversity of implementations (i.e., differences in algorithms, program structures, and identifiers) and find that the diversity increases significantly when N increases from 1 to 10 whereas the diversity increases slightly when N increases from 10. Consequently, we empirically set N to 10. Notably, the diversity of dataset is not to increase the challenges in code generation, but to prevent overfitting of machine learning models.

An illustrating example task¹ is presented as follows:

You are given array consisting of n integers. Your task is to find the maximum length of an increasing subarray of the given array. A subarray is the sequence of consecutive elements of the array. Subarray is called increasing if each element of this subarray strictly greater than previous.

Input: The first line contains single positive integer n — the number of integers. The second line contains n positive integers a_1, a_2, \dots, a_n ($1 \leq a_i \leq 10^9$).

Output: Print the maximum length of an increasing subarray of the given array.

3.3 Removing Duplication

First, we detect and remove duplicate or nearly duplicate tasks from the resulting dataset. To avoid the pairwise comparison among thousands of tasks, we employ the well-known fingerprint algorithm *SimHash* [66] to transform textual description of each task into a fixed-length hash value (called fingerprint). The algorithm guarantees that fingerprints of nearly duplicate texts differ from each other in a small number of bit positions [66]. For each pair of the highly similar fingerprints, we manually check the corresponding tasks to exclude duplicate tasks only. Manual checking is conducted because two different tasks may happen to be lexically similar to each other, but the functionality of the intended software applications are essentially different. Consequently, the first two authors manually check the highly similar tasks. Two tasks ts_1 and ts_2 are duplicate if applications conforming to ts_1 conform to ts_2 as well, and vice versa.

1. <http://codeforces.com/problemset/problem/702/A>

TABLE 3
Resulting Dataset

Dataset	Explanation	#REQs	# Programs	Programming Language	Average Length of REQs (in tokens)	Average Length of Programs character/LOC
ReCa	from programming contests	5,149	20,554	C	185	458 / 35.7
			35,092	C++		578 / 37.2
			32,306	Java		1,121 / 63.8
			16,673	Python		205 / 13.9

Second, we detect and remove duplicate implementations. For each of the tasks, we compare each pair of its submissions to exclude duplicate or nearly duplicate submissions. The comparison is based on the well-known edit distance [67] between two source code fragments:² if the distance is small, i.e., changing a few characters in one fragment can turn it into the other fragment, they are reported as potentially redundant implementations. Before removing such potentially duplicate implementations, the first two authors also manually check them to exclude false positives: two implementations of the same task are duplicate if and only if they are identical except for the difference in format (e.g., blank lines) and/or code comments.

3.4 Static Checking

Both of the websites have a long history, and thus some of the old submissions to the websites may be out of date and cannot be compiled with the up to date compilers. Assuming that up-to-date code generation approaches may target up-to-date compilers only, we filter out such outdated submissions by compiling them with the up-to-date compilers.

By compiling the submissions, we also remove low-quality submissions that result in warnings. Such submissions could be compiled and thus may be executed. However, warnings (e.g., dead code) reported by compilers suggest that such submissions deserve improvement. Consequently, to guarantee high-quality of the resulting dataset, we exclude such submissions that result in compiler's warnings. The exclusion, in turn, may improve the quality of code generation models trained on the resulting dataset (i.e., fewer compiler's warnings on the generated source code).

3.5 Cross-Validation by Software Testing

One of the biggest challenges in building code generation datasets is to guarantee that the included programs act exactly the same as what their associated software requirements specify. In other words, such programs should be accepted as correct implementations by users who propose the requirements. In our case, all submissions are specifically designed for the given tasks, and the websites have run some manually predefined test cases to guarantee that the submissions satisfy the requirements in the most common scenarios. This helps to improve the reliability of the resulting dataset. However, the number of such manually designed test cases is rather small, and thus it is likely that some buggy submissions can still pass such test cases.

2. More advanced tools like MOSS (<http://theory.stanford.edu/~aiken/moss/>) may ease the work.

To further improve the reliability, we carry out cross-validation by software testing. For each of the task t , the cross-validation is conducted as follows:

- First, according to the requirements we manually create a template to specify the input parameters, including their data types and value ranges.
- Second, based on the template, we automatically create test cases with fuzz testing, i.e., create random data as inputs to the programs under test.
- Third, for each test case, we automatically run each of the submissions (to the given task t) with the inputs in the test case. If the submissions result in different outputs, we manually check the results and remove the buggy submissions.

Notably, we do not employ popular test case generation tools like *EvoSuite*. The downloaded programs often receive input from *console* with statements like '*input()*' (Python) and '*scanf*' (C/C++). However, existing test case generation tools like *EvoSuite* generate test cases (more specially, input of the programs) according to parameters instead of console input. As a result, applying such tools to the downloaded programs results in few test cases. To this end, we manually create a template for each task to explicitly specify its input (including those from *console*), and generate test cases automatically based on the template.

3.6 Resulting Dataset

We call the resulting dataset *large scale dataset for Requirements text based Code generation* (ReCa). Details of the dataset is presented in Table 3. By comparing Table 3 against existing datasets in Table 1 (especially MTG, HS, Django, StaQC, and CoNala that describe implementations in general-purpose programming languages), we observe that our dataset has the following advantages:

- First, ReCa is composed of longer requirements of independent software applications. The average length of requirements in our dataset is significantly longer than that of existing datasets. As analyzed in the preceding sections, the textual descriptions in MTG, HS, Django, StaQC, or CoNala are not real-world software requirements. Instead, they are incremental features of a single software project (MTG and HS), pseudo-code (Django), or how-to questions (StaQC and CoNala). In contrast, each of the textual descriptions in our dataset represents a requirement of an independent software application. Software engineers have developed the intended applications successfully according to such textual descriptions.

- Second, ReCa contains more programs. Our dataset contains more than one hundred thousand software programs, much larger than existing datasets.
- Third, ReCa has longer programs. For example, the average length of Java programs in our dataset is 63.8 (LOC), much longer (at least twice) than that of existing datasets. Notably, however, such programs are still significantly smaller than real-world software applications in the industry. These real-world applications may contain millions of lines of source code, which makes them extremely challenging (if not impossible) to be generated automatically by up-to-date deep learning models.
- Fourth, the implementations in ReCa are in multiple general-purpose programming languages. For most of the requirements in our dataset, we provide corresponding implementations in different programming languages at the same time, e.g., Java and Python. It may facilitate the research on cross language code generation, as well as research on the impact of programming languages on code generation.
- Fifth, the implementations in ReCa are validated. Each of the implementations in our dataset has been validated by static checking as well as dynamic software testing to guarantee that they satisfy the declared requirements and they are of high quality.
- Sixth, ReCa provides multiple implementations for the same requirements. A single requirement has up to ten independent implementations in the same language (e.g., Java). Trained on such a dataset, learning-based code generation algorithms may learn equivalence among different code fragments, and thus may be smarter in appreciating the context while generating the next code token. Multiple reference implementations also facilitate more reasonable and more comprehensive quality assessment on generated programs by comparing them against diverse references. Existing approaches often assess the quality of a generated program by computing its lexical similarity (e.g., BLEU) with a single reference because existing datasets often provide a single reference only. The assessment is risky because two semantically equivalent programs may happen to be significantly different in text. Providing a number of diverse references helps to reduce the risk.

3.7 Quality Assessment and Tool Kit

To facilitate research on code generation, we develop an assisting tool to comprehensively assess the quality of generated programs. The tool computes automatically a list of quality metrics for the generated program against its reference programs. The first quality metrics are BLEU (bilingual evaluation understudy) [23] that is widely employed by existing approaches. BLEU was initially proposed to assess the quality of machine translation [23]. For code generation, BLEU scores are calculated by comparing the generated source code against a set of reference programs. The scores range between 0 and 1, suggesting how lexically similar the generated program is to the reference programs. Notably, BLEU for a generated program p is the maximal similarity (BLEU) between p and any of its reference programs: If it is

highly similar (or even identical) to any of its reference programs, the generated program is of high quality even if it is essentially different from other reference programs.

The second code metrics are the number of errors and warnings compilers produce on the generated source code. Existing approaches rarely employ such metrics because most of the generated source code cannot be compiled successfully at all, i.e., they often contain syntactic errors. One of the reasons for such syntactic errors is that most of the reference programs (e.g., code fragments from Stack Overflow) in existing datasets are incomplete and thus cannot be compiled successfully. Consequently, code generation models trained on such datasets rarely generate compilable programs.

The third code metrics are the percentage of passed test cases, i.e., what percentages of the test cases the generated program has passed. In our dataset, we have generated automatically a large number of test cases for each of the tasks (requirements). Consequently, we can run such test cases on the generated programs to assess the extent to which the generated programs satisfy the functional requirements.

The fourth quality metrics are the edit distance-based lexical similarity. Levenshtein distance is widely employed to measure the minimum number of single-character edits (i.e., insertions, deletions, or substitutions) required to change one text (the generated source code in our case) into the other (reference implementation in our case). The edit distance-based lexical similarity (noted as S_{ed}) turns the Levenshtein distance (note as dis) into a similarity varying from zero to one: $S_{ed}(gc, ref) = 1 - dis(gc, ref) / \max(|gc|, |ref|)$ where gc is the generated source code and ref is a reference implementation.

BLEU is selected because it is widely employed by existing approaches to evaluate the quality of code generation [23]. The number of compiler errors and warnings (the second metrics) is selected because it represents the syntactic quality of the generated source code. The percentage of passed test cases (the third metrics) is selected because it represents the functional quality of the generated source code. The edit distance is selected because it is widely used to measure the similarity between source code. BLEU and edit distance concern the lexical similarity between generated programs and references whereas the number of compiler errors and the percentages of passed test cases concerns the syntaxes and functionality of the generated programs, respectively. Consequently, employing such diverse metrics facilitates comprehensive assessment of the generated programs. To facilitate more comprehensive assessment, however, the tool kit also provides additional metrics, i.e., NIST, WER, and Subtree Metric [68].

We employ additional quality metrics (as introduced in preceding paragraphs) besides BLEU for assessing the quality of generated source code because of the following reasons:

- First, although BLEU is frequently employed to assess the quality of generated source code, it has significant limitations [69] for assessing source code. Unlike natural languages, source code has less tolerance for noise and poor syntax/semantics. Consequently, programs with high BLEU could be syntactically incorrect and essentially different from reference programs in semantics.

- Second, even the implementations for the same task (requirements) are often diverse in text. Consequently, computing the lexical similarity between the generated source code and its diverse reference implementations may fail to reveal the quality of code generation.

4 EXPERIMENTAL SETUP

As introduced in Section 2, researchers have achieved great advances recently in deep learning-based code generation. A number of approaches have been proposed, and evaluation results on different datasets suggest that they are highly accurate. For example, the syntactic neural model proposed by Yin and Neubig achieves a high BLEU (0.845) on *Django* dataset [22], which suggests that the generated source code is very close to the reference implementation. However, as introduced in the preceding sections, such datasets employed in the evaluations have significant limitations and thus good performance on such datasets may not necessarily lead to good performance in handling real-world software requirements. To assess the state of the art, in this section we evaluate such approaches on our new dataset.

4.1 Validation Questions

The evaluation investigates the following questions:

- Q1: How accurate are the state-of-the-art approaches on the new dataset?
- Q2: How often do the generated programs pass syntactic checking?
- Q3: How often do the generated programs pass pre-defined test cases?
- Q4: Is the generated source code useful for developers?
- Q5: Where and why do state-of-the-art approaches succeed?
- Q6: To what extent do state-of-the-art approaches understand software requirements?
- Q7: Can we propose a simple and intuitive approach whose performance is comparable to (or even better than) that of the state-of-the-art approaches?
- Q8: Can we improve the performance of the evaluated approaches if we keep only a single solution per requirement?
- Q9: Can we improve the performance of the evaluated approaches by unifying identifiers in requirements and associated source code?

Research question Q1 concerns the performance of the state-of-the-art approaches on our new dataset. Many of the state-of-the-art approaches are reported to be highly accurate on existing datasets [22]. Answering this question may reveal whether the reported high performance is owned to the limitations of the involved datasets.

Research question Q2 investigates how often the deep learning-based approaches generate syntactically correct programs, and how often such programs could be executed without exceptions. Investigating Q2 would reveal to what extent such approaches learn automatically the syntax of target programming languages.

Research question Q3 investigates to what extent the generated programs are semantically correct, i.e., consistent with the given software requirements. The investigation would reveal to what extent the approaches can learn the semantics of requirements that are described in English, and turn such semantics into implementations.

Research question Q4 investigates the usefulness of the generated programs. It is likely that developers cannot use the generated code as-is. However, if the effort to modify it to make it work is much smaller than the effort to write the correct code from scratch, the generated source code (and the generation approaches) could be considered useful.

Research question Q5 investigates what kind of tokens could be generated correctly, and potential reasons for the success. The investigation will reveal the strength of existing approaches, and the rational for the strength.

Research question Q6 investigates the influence of the input (textual requirements) on the output (generated source code). It is challenging for computers to fully understand natural languages. Consequently, it is likely that the deep learning-based code generation approaches cannot fully understand the requirements in English. Answering this question may reveal whether natural language understanding is the major obstacle to deep learning-based code generation.

Research question Q7 concerns the substitutability of the state-of-the-art deep learning-based complex approaches. Answering this question may reveal whether such deep learning-based complex approaches are really better (or much better) than simple and intuitive approaches.

Research question Q8 concerns the effect of removing redundant implementations for the same requirement. While answering the preceding research questions, we provide the evaluated approaches with multiple code snippets/solutions for the same requirement. However, this has not been done by the authors of the evaluated approaches and the loss function of the approaches is not prepared for this. Consequently, such neural networks may fail to learn anything from such different implementations. To this end, we repeat the evaluation after removing the redundant implementations, i.e., we keep only a single solution per requirement.

Research question Q9 concerns the identifiers in requirements and their associated implementation (source code). Such identifiers do not influence the syntax or semantics of programs. However, replacing them with unified tokens e.g., var_0 and var_1 , could significantly reduce the size of vocabularies employed by automated code generation approaches. Research question Q9 investigates the effect of the preprocessing.

4.2 Evaluated Approaches

We select Seq2Seq [28], SNM [22], Tree2Tree [30], TRANX [31] and Coasr-to-Fine [32] for the evaluation because of the following reasons.

- First, they could generate source code in general-purpose programming languages according to software requirements in English, which makes it practical for them to work on our dataset.
- Second, their implementation is publicly available, which significantly facilitates the evaluation. Some well-known approaches [14], [29], [34] that could

generate source code from requirements text are not selected for evaluation because we either fail to get their implementations [14], [29] or fail to make them work on our dataset [34].

- Third, SNM [22], Tree2Tree [30], TRANX [31] and Coarse-to-Fine [32] were proposed recently, and represent the state of the art. To the best of our knowledge, they are the latest approaches that 1) have publicly available implementations and 2) can work on our dataset to generate Python programs according to requirements text.
- Although Seq2Seq [28] was initially proposed for semantic parsing, it is widely employed as a baseline in code generation [33]. Consequently, we include it for the evaluation as well.

4.3 Process

To investigate questions Q1, Q2, Q3, and Q5, we conduct the first empirical study as follows:

- First, we select all tasks for evaluation from our dataset that are accompanied by Python source code. To the best of our knowledge, no publicly available deep learning-based models/implementations can transform requirements text into programs in general-purpose programming languages other than Python. Although LPN [14] generates Java programs, its implementation is unavailable. Consequently, we select only Python programs for the empirical study. The resulting dataset is noted as *selected dataset*. It is composed of 2,740 tasks (requirements) and 16,673 Python programs.
- Second, from the selected dataset, we randomly select 300 tasks as testing dataset, 200 tasks as validation dataset, and other as training dataset.
- Third, the selected dataset is preprocessed. For the textual requirements, we leverage NLTK [70], [71] to replace acronyms (e.g., “what’s”) with separated words (e.g., “what is”), to turn characters into lowercase, to split the text into a sequence of word by word segmentation and special characters (e.g., splitting “Java.System” according to “.”), to remove stop words, and to apply lemmatization on the resulting words. For example, the requirement:

“Some natural number was written on the board. Its sum of digits was not less than k. But you were distracted a bit, and someone changed this number to n, replacing some digits with others. It’s known that the length of the number didn’t change. You have to find the minimum number of digits in which these two numbers can differ.”

is finally turned into:

“some natural number write on board. its sum of digit not less than k. but you distract bit, someone change number to n, replace some digit with others. it know length of number do not change. you have to find minimum number of digit in which these two number can differ.”

after the preprocessing. For the selected source code, we remove comments and copyright declarations, and

Listing 1. Example of Source Code Preprocessing

```

1 /* Code before preprocessing */
2 #_get_number
3 w=_int(input())
4 if_w%2==0_and_w!=2:
5     _print('YES')
6 else:
7     _print('NO')
8
9 /* Code after preprocessing */
10 w=_int(input())
11 if_w%2==0_and_w!=2:
12     _print('YES')
13 else:
14     _print('NO')
```

format the source code (with Autoprep8 [72]). Listing 1 presents an illustrating example of source code preprocessing; the code before and after preprocessing.

- Fourth, for each of the selected approaches, we train it on the training and validation datasets, and test it on the testing dataset.
- Finally, we evaluate the quality of generated source code with the tool kit introduced in the preceding sections. The quality metrics generated by the tool kit are subsequently employed to answer the research questions.

To maximize the potential of the evaluated approaches, we perform hyper parameter tuning for each of the evaluated approaches. Basically, we follow the grid-search tuning approach [73] but pick up grids (i.e., to-be-tested values of parameters) dynamically and empirically to speed up the tuning process. Notably, for each of the to-be-tested setting, we train the selected approach with the given setting on the given training data (all of the requirements-code pairs regardless of their topics), and then validate the performance on the validation set. Based on the validation, we empirically select the next to-be-tested setting. For a given setting, we train the associated approach with the setting once and for all (instead of repeating the training and validation for several times) because the training is highly time-consuming. For each of the evaluated approaches, it takes more than one week to tune its hyper parameters on a GPU server (OS: Ubuntu 14.04.5; CPU: 56 * Intel(R) Xeon(R) CPU E5-2683 v3 @ 2.00GHz; GPU: 2* TITAN Xp; RAM: 64GB). The final parameters are presented in Table 4 where N/A suggests that the implementation of the given approach does not contain the parameter or the parameter does not deserve tuning.

To investigate question Q4, we randomly select eleven tasks from the dataset and invite thirty developers to conduct a controlled experiment. The participants have rich experience in Python. They did not know the intent of the experiment in advance, which helps to reduce potential bias. The experiment is conducted as follows:

- First, each of the participants is requested to code from scratch for a selected task (noted as *preTest-Task*), and we record the time that developers take to finish the assigned task. Notably, a task is finished only if the submitted program has passed all predefined test cases. The top five (who spend the shortest

TABLE 4
Final Parameters for Evaluated Approaches

Approaches \ Parameters	Embedding Size	Hidden Size	Epoch	Batch Size	Decoder Dropout	Learning Rate	Learning Rate Decay
Seq2Seq	200	N/A	120	20	0.4	0.01	0.98
SNM	256	256	100	7	0.4	0.001	N/A
Tree2Tree	300	256	100	8	0.2	0.001	N/A
TRANX	128	256	100	10	0.3	0.001	0.5
Coarse-to-Fine	250	N/A	75	10	0.3	0.002	0.99

TABLE 5
Evaluation Results on Our Dataset

Approaches	BLEU on New Dataset	BLEU on Django	BLEU on HS	Syntactically Correct Programs	Executable Programs	Functionally Correct Programs
Seq2Seq	0.138	0.673	0.550	44.7%	6.0%	0%
SNM	0.188	0.845	0.758	93.0%	16.7%	0%
Tree2Tree	0.150	0.825	0.716	83.7%	14.3%	0%
TRANX	0.184	0.856	0.695	81.7%	9.0%	0%
Coarse-to-Fine	0.176	0.854	0.640	10.0%	2.7%	0%
Average	0.167	0.811	0.672	62.6%	9.7%	0%

time in finishing the given task) and the bottom five (who spent the longest time) are excluded from further evaluation. The other twenty participants are divided into two equally sized groups according to their coding speed: participants in *Group A* is faster than anyone from *Group B*. The purpose of this step is to construct two participant groups where participants within the same group have similar coding speed. Grouping the participants by coding speed may reduce the bias introduced by the difference in participants' programming ability/speed.

- Second, for each of the selected participants, we request him/her to code from scratch for five out of the remaining ten tasks (i.e., all selected tasks except for *preTestTask*), and to complete the other five tasks based on programs generated by SNM. SNM is selected because it achieves the best performance among the evaluated approaches (see Section 5.1 and Table 5 for details). The assignments of the tasks guarantee that exactly half of the participants from each group finish a task from scratch and another half the participants from the same group finish the same task by modifying the generated program.
- Third, we record the time that developers take to finish the assigned tasks, and analyze the results of the two groups.

To investigate question Q6, i.e., to what extent the input (software requirements) is exploited by code generation approaches, we conduct the third empirical study as follows:

- First, for each of the selected approaches, we train it on the training and validation datasets, and test it on the testing dataset (noted as *tdata*). We compute their performance, and call it *original performance* (noted as P_{orig}). Notably, it is the same as we do in the previous empirical study.
- Second, for each item (composed of requirements and their implementations) in the testing dataset *tdata*, we replace the requirements with random noise. The

noise is generated automatically by picking tokens randomly from a large corpus. The length (in tokens) of the noise is equal to the original requirements. The resulting testing dataset is noted as *tdata'*.

- Third, on the revised testing dataset *tdata'*, we evaluate the resulting models that are trained on the first step. The resulting performance (noted as P_{noise}) is compared against the original one (i.e., P_{orig}).

If replacing requirements with random noise fails to reduce the performance significantly, it is likely that the code generation approaches learn little from the requirements.

To investigate question Q8, i.e., the effect of removing redundant implements, we conduct the fourth empirical study as follows:

- First, for each of the tasks in the dataset, we randomly select and keep one of its implementations. Other implementations are removed from the dataset. We call the resulting dataset as *nonredundant dataset*
- Second, we repeat the first empirical study on the *nonredundant dataset*.
- Third, we compare the resulting performance on *nonredundant dataset* against that on the original dataset where multiple implementations for the same tasks are exploited.

5 RESULTS AND ANALYSIS

5.1 Q1: Significant Reduction in BLEU

To answer question Q1, we evaluate the state-of-the-art code generation approaches on our new dataset, and evaluation results are presented in Table 5. The first column presents the evaluated approaches. The second column presents BLEU of such approaches on our new dataset. To facilitate comparison, we also present their BLEU on existing datasets (i.e., Django and HS) on the third and fourth columns. Columns 5-7 present the syntactical and semantic checking results of the generated source code (on our dataset).

From the first four columns, we make the following observations:

```

1n, m = map(int, input().split())
2e = [[] for x in range(n + 1)]
3for i in range(m):
4    x, y = map(int, input().split())
5    e[x].append(y)
6    e[y].append(x)
7c = set(range(1, n + 1))
8val = 2**n
9while c:
10    s = c.pop()
11    dfs = [s]
12    val -= 2
13    while dfs:
14        cur = dfs.pop()
15        for nxt in e[cur]:
16            if nxt in c:
17                dfs.append(nxt)
18                c.remove(nxt)
19print(val)

```

```

1n, m = map(int, input().split())
2a = list(map(int, input().split()))
3ans = 0
4for i in range(n):
5    ans += a[i]

```

Fig. 2. Visual comparison between reference implementation (left) and generated program (right).

- First, BLEU of the evaluated approaches is rather low. It varies from 0.138 to 0.188, with an average of 0.167. Such a low BLEU suggests that the generated source code is often significantly different from reference implementations, i.e., the validated implementations in the dataset.
- Second, switching from existing datasets to our new dataset reduces BLEU significantly. The average BLEU is reduced significantly from 0.811 (on Django) and 0.646 (on HS) to 0.167. The reduction is up to $79\% = (0.811 - 0.167) / 0.811$, and $75\% = (0.672 - 0.167) / 0.672$, respectively.

To figure out the reason for low BLEU, we employ *diff*, a popular and powerful tool, to visualize the difference between the generated programs and their references. A typical example is presented in Fig. 2. The right part of the figure presents the program generated by *SNM*. The left part presents a reference implementation that has the greatest BLEU with the generated one. The common part (i.e., successfully generated statements) is shown on white background. Missing part (i.e., statements that should have been generated) is shown on red background, added part (i.e., statements that should not have been generated) is on green background, and the modified part is on yellow background.

We randomly sample 100 generated programs for visual comparison. Based on the comparison, we make the following observations:

- First, most statements are not generated successfully. Around 75 percent of the statements in reference programs are missing in the generated programs. For example, in Fig. 2 sixteen out of the nineteen ($84\% = 16/19$) lines of source code in the reference implementation are on red background, suggesting that the evaluated approach fails to generate the majority of the reference implementation.
- Second, most of the generated source code is irrelevant, i.e., having no counterparts in the reference implementations. Around 81 percent of the generated source code is irrelevant (on green background). For example, in Fig. 2 three out of the six ($50\% = 3/6$) lines of source code in the generated program are on green background.

To validate whether the observations could be generalized to all generated programs, we compute automatically how often tokens in reference implementations are missed (i.e., shown on red or yellow background), and how often

TABLE 6
Mismatch Between Generated Programs and References

Approaches	Missing Tokens	Irrelevant Tokens
Seq2Seq	81.8%	87.5%
SNM	71.2%	74.8%
Tree2Tree	75.4%	81.0%
TRANX	74.4%	81.5%
Coarse-to-Fine	81.6%	88.5%
Average	76.9%	82.7%

tokens in the generated programs are irrelevant (i.e., shown on green or yellow background). Results are presented in Table 6. The first column presents evaluated approaches. The second column presents the percentages of the tokens in the reference programs that are missed by the generated programs. The third column presents the percentages of the tokens in the generated programs that are irrelevant, i.e., having no counterparts in the reference implementations. From this table, we observe that on average, 76.9 percent of the tokens in reference implementations are missed, and 82.7 percent of the tokens in generated programs are irrelevant. In other words, only $23.1\% = (1 - 76.9\%)$ of the tokens in the reference implementations are generated successfully, and only $17.3\% = (1 - 82.7\%)$ of the generated programs tokens are really useful. The statistics confirm our preceding observation that the generated programs are often significantly different from references.

We also employ additional metrics [68] (i.e., NIST, WER, and Subtree Metric) besides BLEU. Evaluation results are presented in Table 7. The results confirm the conclusions drawn on the preceding paragraphs: the performance of the evaluated approaches is not promising on the new dataset.

One potential cause of the low accuracy could be the irregularity of the tokens in the dataset. If tokens in the testing dataset are often missing in the training dataset, it is likely that machine learning model cannot generate tokens accurately. To this end, we compare the vocabularies of the training dataset and testing dataset. The comparison results suggest that 96 percent of the (requirements) text tokens in the testing dataset are actually observed in the training dataset whereas 79 percent of the source code tokens are observed in the training dataset. The results may suggest that the difference in vocabularies of training data and testing data is not the major reason for low accuracy.

It is quite intuitive that the longer the text and programs are, the lower the generation accuracy would be. To quantitatively verify this, we partition the tasks into four equally sized groups according to their length of requirements and length of reference programs, respectively. Notably, for a single task, we have multiple reference implementations (programs). Consequently, we classify the task based on the average length of its reference programs (instead of the length of a single reference program). Evaluation results are presented in Tables 8 and 9. On Table 8, we present how the length of requirements influences the performance (BLEU). Q1 contains 75 = 300/4 tasks that have the shortest requirements whereas Q4 contains 75 tasks with the longest requirements. Each row of the table presents the performance (BLEU) of an evaluated approach on different groups of tasks. From this table, we make the following observations:

TABLE 7
Evaluation Results with Additional Metrics

Metrics Approaches	NIST	WER	Subtree
Seq2Seq	1.369	8.089	0.117
SNM	1.453	0.785	0.200
Tree2Tree	1.185	0.866	0.139
TRANX	1.721	1.179	0.160
Coarse-to-Fine	1.988	1.643	0.116
Average	1.543	2.513	0.146

- First, all of the evaluated approaches result in the lowest performance on Q4 group that is composed of the longest requirements. It may suggest that extremely long requirements (varying from 228 tokens to 391 tokens) have significant negative impact on the performance of code generation.
- Second, all of the evaluated approaches result in the highest performance on Q2 group where the length of requirements varies from 125 tokens to 171 tokens. In contrast, they result in significantly lower performance on Q1 that is composed of the shortest requirements (varying from 16 tokens to 125 tokens). The results may suggest that the following assumption is not necessarily true: the shorter the requirements text is, the higher the generation accuracy would be.

Table 9 presents the influence of programs' length where Q1 contains 75 tasks with the shortest reference programs. From this table, we observe that the performance decreases with the increase of programs' length. The average BLEU reduces from 0.218 on Q1 (where the length of programs varies from 6 tokens to 77 tokens) to 0.0726 on Q4 (where the length of programs varies from 199 to 822 tokens). The evaluation results may suggest that the length of programs has a significant negative impact on the performance of automated code generation.

Based on the preceding analysis, we conclude that concerning the common performance metrics (i.e., BLEU) the state-of-the-art code generation approaches cannot reach a high performance on the new dataset as they do on existing datasets. Concerning other performance metrics like NIST, WER, STM, and Subtree metrics, the evaluated approaches also result in poor performance on the new dataset. Most tokens in reference implementations are missed whereas most of the generated tokens are irreverent. One possible reason for the significant reduction in performance is that

TABLE 8
Impact of Requirements' Length on BLEU

Length of Requirements Approaches	Q1	Q2	Q3	Q4
Seq2Seq	0.131	0.157	0.141	0.125
SNM	0.195	0.214	0.185	0.156
Tree2Tree	0.155	0.173	0.146	0.125
TRANX	0.198	0.204	0.177	0.158
Coarse-to-Fine	0.164	0.192	0.182	0.167
Average	0.1686	0.188	0.1662	0.1462

TABLE 9
Impact of Programs' Length on BLEU

Length of Reference Programs Approaches	Q1	Q2	Q3	Q4
Seq2Seq	0.150	0.181	0.146	0.076
SNM	0.287	0.240	0.170	0.054
Tree2Tree	0.257	0.192	0.120	0.029
TRANX	0.223	0.231	0.194	0.088
Coarse-to-Fine	0.173	0.215	0.201	0.116
Average	0.218	0.2118	0.1662	0.0726

the new dataset is more complex and more diverse than existing ones.

5.2 Q2: Syntactic Checking

To answer question Q2, we conduct syntactic checking on the generated programs. The checking is composed of two parts. In the first part, we conduct static syntactic checking on the generated programs with the state-of-the-practice tool *Pylint* [74]. For convenience, we call programs that pass the static checking as *syntactically correct programs*. In the second part, we try to execute the programs (with sample input specified in the requirements) that pass the static checking on the first step. If the execution results in any syntactic error or runtime exception, the programs are non-executable. Results of the static syntactic checking are presented in the fifth column of Table 5 whereas the execution results are presented in the sixth column. From these two columns, we make the following observations.

The first observation is that most (up to 93.0 percent) of the programs generated by AST-based approaches (i.e., SNM, Tree2Tree, and TRANX) pass the static syntactic checking whereas programs generated by other approaches have significantly smaller chance (less than fifty percentage) to pass the static syntactic checking. The results may suggest that generating ASTs (and then transferring them into source code) helps much in avoiding syntactic errors. In contrast, generating source code (as generic text) directly is much riskier because the state-of-the-art approaches could not yet automatically recognize the complete syntax of programming languages that is embedded in the training programs.

To figure out what kind of syntax such approaches fail to learn automatically, we manually analyze the syntactic errors generated by such approaches. In general, the syntactic checking on generated programs (i.e., to compute how many of the generated programs are syntactically correct and how many of them are executable) is completely automated, and no manual checking is required. Manual checking is only employed to empirically reveal the common syntax errors in the generated programs. The results of the manual analysis suggest that *undefined-variable* is dominating. Undefined variable refers to usage of variables that have not yet been defined before the usage. An illustrating example is presented in Fig. 3 where *s* on Line 7 is *undefined*. *Undefined-variable* accounts for 59, 78, 70, 53, and 82 percent of the syntactical errors generated by Seq2Seq, SNM, Tree2Tree, TRANX, and Coarse-to-Fine, respectively. On average, it accounts for 68% (=2003/2965) of the syntactical errors we

```

1 n = int(input())
2 a = list(map(int, input().split()))
3 ans = 0
4 for i in range(n):
5     x, y = map(int, input().split())
6     a[i] += 1
7 print(s)

```

Fig. 3. *Undefined-variable* in generated program.

encounter during the evaluation. Consequently, to further improve the state of the art, researchers in the future should pay more attention to such kind of syntactic errors. For example, to reduce *Undefined-variable* errors, we may request the deep learning models to select from a short list of variables declared in the generated source code when variables are expected. In contrast, existing models always select tokens from a generic large vocabulary, which often results in undefined-variables.

The second observation is that only a small part (less than 10 percent) of the generated programs could be executed without exceptions. Notably, Python is a dynamically typed programming language, and thus many type errors could not be identified by static syntactic checking. As a result, programs that pass static syntactic checking may still fail to run successfully. To figure out what kind of problems are preventing such programs from successful execution, we manually analyze the runtime exceptions that we encounter while executing such programs. Notably, we do not execute those who fail to pass static syntactic checking because they are bound to fail. The results of our analysis suggest that most of the exceptions are *ValueError*. According to Python documents [75], *ValueError* exception is ‘raised when an operation or function receives an argument that has the right type but an inappropriate value, and the situation is not described by a more precise exception such as *IndexError*’. An illustrating example is presented in Fig. 4. The first input statement on Line 1 expects a string that could be parsed into an integer. However, the actual input ‘RYBGRYBGR’ fails, and thus a *ValueError* is raised. *ValueError* exceptions account for 72.5% (=578/797) of the exceptions encountered during the evaluation.

Based on the preceding analysis, we conclude that AST-based code generation approaches have a great chance to generate syntactically correct Python programs. However, such programs are often non-executable because of various runtime exceptions.

5.3 Q3: Dynamic Validation

To answer question Q3, we run test cases in the dataset on the generated programs. Results are presented in the last column of Table 5. From this column, we observe that none of the generated programs passes any test case in the

```

# Actual input: RYBGRYBGR
1 n = int(input())
2 a = [list(map(int, input().split())) for i in range(n)]
3 for i in range(n):
4     for j in range(n):
5         if a[i][j] == 1 and a[i][j] == 1:
6             print('YES')
7             break
8 print('NO')

```

Fig. 4. *ValueError* exception thrown by generated program.

```

1 n = int(input())
2 a = list(map(int, input().split()))
3 ans = 0
4 for i in range(1, n):
5     if a[i] == a[i]:
6         ans += 1
7 print(ans)

```

(a) Generated Program

```

Actual input:
4
1 2 3 5

Expected output:
++-

```

Actual output:

```

3

```

(b) Failed Test Case

Fig. 5. Sample program and failed test case.

dataset. The results may suggest that even if some of the generated programs are syntactically correct and executable, they fail to fulfill the given requirements. One of the possible reasons for the failure is that the evaluated approaches do not really understand the software requirements (details are presented in Section 5.6). As a result of the incomprehension, such approaches cannot generate programs that fulfill the requirements. An illustrating example is presented in Fig. 5 where the expected output is a sequence of ‘+’ and ‘-’. However, the generated program outputs a single integer (*ans* on Line 7).

Notably, the requirements in the dataset have explicitly specified the format of programs’ input and output, and thus the failure should not be owned to the flexibility in the design of program interfaces. For the given example in Fig. 5, developers could figure out the exact format of the expected output based on the specification: “*Output: In a single line print the sequence of n characters ‘+’ and ‘-’, where the ith character is the sign that is placed in front of number a_i*”

Based on the preceding analysis, we conclude that the generated programs have little chance to pass the associated test cases. Consequently, manual interference (especially code revision and validation) is indispensable even if such state-of-the-art automatic code generation approaches are employed.

5.4 Q4: Usefulness of Generated Programs

To answer question Q4, we record the time that developers take to finish the tasks, with and without generated programs, respectively. Results are presented as box plots in Fig. 6 (for Group A) and Fig. 7 (for Group B). The blue boxes are associated with cases where developers create source code from scratch. The red ones are associated with cases where generated programs are modified to make them work.

From the box plots, we fail to observe significant difference between the two development models (i.e., coding from scratch or based on generated programs). For Group A, coding from scratch took 652 minutes in total whereas revision based on generated programs took 655.5 minutes. For Group B, coding from scratch took 714.5 minutes whereas revision based on generated programs took 707.4 minutes. Overall, the difference between the two coding models is minor. We also perform a significance test on the resulting data. Results suggest that there is no significant difference between the two coding models: the p-value=0.9696 and F=0.0015 for Group A and p-value=0.9318 and F=0.0074 for Group B. For both groups, the p-value is significantly greater than 0.05. We also compute the effect size (Cohen’s d), and results suggest the effect size (-0.0077 for Group A and 0.0173 for Group B) is small.

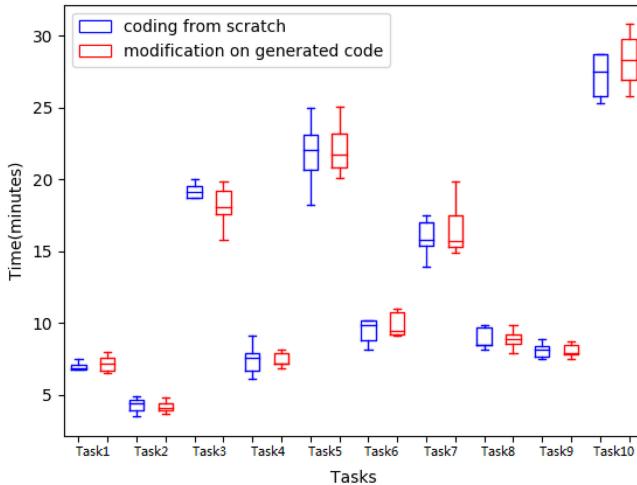


Fig. 6. Usefulness of generated programs (Group A).

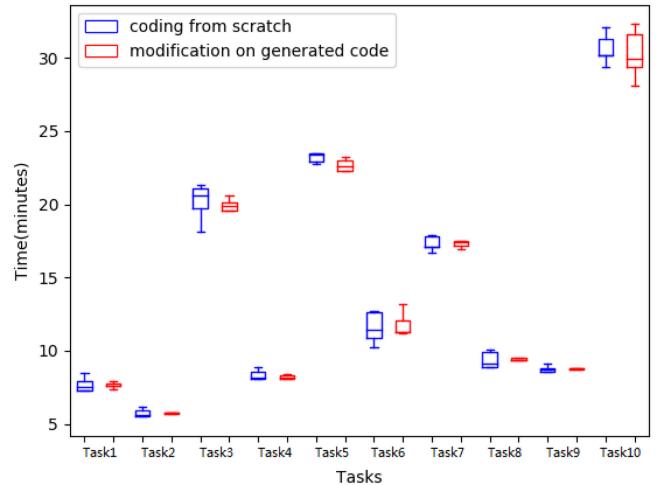


Fig. 7. Usefulness of generated programs (Group B).

We conclude based on the preceding analysis that the generated source code cannot significantly reduce the cost (time) of programming, i.e., modification of the generated programs is not significantly easier than creating programs from scratch.

5.5 Q5: Where and Why They Succeed

To answer question Q5, we manually analyze the generated source code. It is highly challenging and time-consuming to manually compare all of the 1,500 ($=300 \times 5$) generated programs against 3,000 ($=300 \times 10$) reference implementations. Consequently, we take the following measures to simplify the manual checking. First, we randomly select 30 (out of 300) software requirements from the testing dataset, and confine the manual checking to this subset. Second, for each generated program on this subset, we only compare it against one of its reference implementations that has the greatest similarity (BLEU) with it. We employ *diff* to visualize the difference (and common ground as well) between a generated program and its reference implementation. A typical example is presented in Fig. 2.

Based on the manual checking, we observe that the evaluated approaches often succeed or partially succeed in generating *input*, *output*, and *for* statements. As suggested by Fig. 2, SNM generates the *input* statement correctly ('*n,m = map(int, input().split())*'), and places it in the right place, i.e., the very beginning of the program. It also succeeds in generating *output* statement '*print(val)*' and *for* statement (Line 3 on the left part of Fig. 2) except for the variable names. Table 10 presents how often *input*, *output*, and *for* statements

are generated successfully. The first column of Table 10 presents different approaches. The second column presents how often the evaluated approaches succeed or partially succeed (inside parentheses) in generating *input* statements. If the generated *input* statement is identical to that in the reference implementation, we say the generation is correct. Otherwise, we manually assess whether the generation is partially correct (with slight difference) or incorrect. The third and the fourth columns present how often the evaluated approaches succeed or partially succeed in generating *for* and *output* statements, respectively. From this table, we observe that all of the evaluated approaches are good at generating such statements. On average, around one fifth of the *input* and *for* statements are generated correctly, and more than half of them are generated partially successfully. Although *output* statements are more difficult to generate correctly (because of variables involved in the statements), in most cases (84 percent on average) the evaluated approaches know that an *output* statement (i.e., '*print(*)*') should be generated and placed at the end of the generated programs.

One of the possible reasons for the success in generating *input*, *output*, and *for* statements is that such statements are highly popular in the training data. The popularity of related statements is presented in Table 11. The first column presents the popular statement (or part of a statement). The second column presents their popularity in training programs, i.e., how many percentages of the programs in the training dataset contain such statements. The third column presents their popularity in testing programs. Columns 4-8

TABLE 10
Well Generated Statements

Approaches	<i>Input Statement correct (partially correct)</i>	<i>For Statement correct (partially correct)</i>	<i>Output Statement correct (partially correct)</i>
Seq2Seq	17% (83%)	24% (60%)	3% (90%)
SNM	17% (83%)	26% (67%)	7% (83%)
Tree2Tree	13% (87%)	11% (54%)	7% (83%)
TRANX	30% (70%)	22% (59%)	3% (83%)
Coarse-to-Fine	27% (70%)	26% (70%)	7% (83%)
Average	21% (79%)	22% (62%)	5% (84%)

TABLE 11
Popularity of Well Generated Statements

Statements	In Training Programs	In Testing Programs	In Generated Programs				
			Seq2Seq	SNM	Tree2Tree	Tranx	Coarse-to-Fine
input()	97%	97%	99%	100%	99%	100%	98%
print(*)	99%	99%	86%	86%	95%	89%	91%
for * in *	76%	75%	84%	83%	67%	89%	92%
for i in range	50%	47%	77%	79%	57%	85%	85%

TABLE 12
Change of BLEU When Normal Input Is Replaced with Random Noise

	Seq2Seq	SNM	Tree2Tree	TRANX	Coarse-to-Fine	Average
Normal Input	0.138	0.188	0.150	0.184	0.176	0.167
Random Noise	0.152	0.173	0.147	0.178	0.169	0.164

TABLE 13
Popularity of Well Generated Statements (Random Noise)

Statements	Seq2Seq	SNM	Tree2Tree	Tranx	Coarse-to-Fine
input()	99%	100%	99%	99%	99%
print(*)	97%	90%	95%	90%	97%
for * in *	82%	80%	76%	85%	96%
for i in range	78%	76%	64%	82%	88%

present their popularity in programs generated by different approaches. From the table, we observe that the *output* statement "print(*)" (where * is a wildcard character) appears in almost all of the training and testing programs, and thus the deep learning-based approaches learn to generate this statement frequently. For example, SNM and TRANX include this statement in each of their generated programs. The same is true for *input* statement "input()" and *for* statement "for * in *".

Based on the preceding analysis, we conclude that the state-of-the-art approaches have the ability to generate highly popular statements, like *input*, *output*, and *for* statements.

5.6 Q6: Little Learned From Requirements

To investigate to what extent the evaluated approaches understand software requirements (input of the approaches), we replace the requirements in the testing data with random noise, and repeat the evaluation. The random noise is created as follows. First, we collect all unique tokens from requirements in the training data, noted as S_{token} . Second, for each requirement r_i in the testing data, we generate an empty noise $noise(r_i)$. Third, we randomly select a token from S_{token} , and append it to $noise(r_i)$. This step is repeated until $noise(r_i)$ and r_i are equally sized.

Evaluation results are presented in Table 12 where the second and third rows present the BLEU of the evaluated approaches with normal input and noise input, respectively. From this table, we observe that replacing normal input with random noise results in small changes in BLEU of the evaluated approaches. The average BLEU (0.164) with random noise is comparable to that (0.167) with normal input. We also notice that the random noise even increases the performance of Seq2Seq, improving its BLEU from 0.138 to 0.152.

We also investigate how often the most popular statements (e.g., *input*, *print*, and *for* statements) are generated by the evaluated approaches when normal input is replaced with random noise. Results are presented in Table 13. From this table, we observe that such popular statements are generated frequently as well. By comparing Table 13 against Table 11, we conclude that replacing requirements text with random noise does not prevent the evaluated approaches from generating the most popular statements.

Based on the preceding analysis, we conclude that the evaluated approaches learn little from input requirements.

5.7 Q7: Simple Alternative Approach

As suggested by the preceding analysis in Section 5.5, the evaluated approaches work well in generating popular statements. Consequently, an intuitive and simple way to simulate the evaluated approaches is to generate popular statements only. We call it *popularity-based approach*.

The approach works as follows. First, it computes the average length of the programs in training data. In our case, the average length is 13 lines of source code, noted as $n = 13$. Second, for each unique line of source code in the training data, the approach computes its popularity, i.e., how often it appears in the training programs. Third, it sorts the unique lines according to their popularity, and inserts the top n lines into a new program p . Finally, the approach always returns this program (p) as the *generated program* regardless of the input (requirements). Notably, this approach completely ignores the input (requirements), and thus it is of little value in practice. However, it may intuitively reveal the state of the art by comparing it against the state-of-the-art approaches.

We apply this simple popularity-based approach to our dataset. Evaluation results suggest it achieves a BLEU of

TABLE 14
Evaluation Results on Nonredundant Dataset

Approaches	BLEU on New Dataset	NIST	WER	Subtree	Syntactically Correct Programs	Executable Programs	Functionally Correct Programs
Seq2Seq	0.108	0.667	5.886	0.060	19.7%	4.3%	0%
SNM	0.151	0.738	0.873	0.133	27.3%	10.0%	0%
Tree2Tree	0.129	0.642	0.968	0.133	59.0%	0.3%	0%
TRANX	0.149	0.836	1.292	0.113	41.0%	9.3%	0%
CoasetoFine	0.155	1.006	1.602	0.029	3.0%	1.0%	0%
Average	0.138	0.778	2.124	0.093	30.0%	5.0%	0%

TABLE 15
Effect of Unifying Identifiers

Applications	Unifying Identifiers				Without Unifying Identifiers (Default Setting)			
	BLEU	Syntactically	Executable	Functionally	BLEU	Syntactically	Executable	Functionally
Seq2Seq	0.135	47.3%	7.7%	0%	0.138	44.7%	6.0%	0%
SNM	0.181	80.0%	16.3%	0%	0.188	93.0%	16.7%	0%
Tree2Tree	0.177	72.3%	12.0%	0%	0.150	83.7%	14.3%	0%
TRANX	0.187	84.3%	4.0%	0%	0.184	81.7%	9.0%	0%
Coarse-to-Fine	0.151	3.3%	0.1%	0%	0.176	10.0%	2.7%	0%
Average	0.166	57.4%	8.0%	0%	0.167	62.6%	9.7%	0%

0.211, significantly higher than any of the evaluated deep learning-based approaches (as shown in Table 5). The comparison intuitively reveals the state of the art: the advanced deep learning-based code generation approaches cannot even outperform this intuitive and impractical approach.

Based on the preceding analysis, we conclude that it is likely for simple and intuitive approaches to outperform the state-of-the-art deep learning-based approaches concerning the common performance metrics BLEU.

5.8 Q8: Removing Redundant Implementations Does Not Help

Evaluation results on the nonredundant dataset are presented in Table 14. By comparing this table against Table 5 (performance on the original dataset where multiple implementations for the same tasks are exploited), we make the following observations:

- First, removing redundant implementations does not help. For example, all of the evaluated approaches result in lower BLEU on the nonredundant dataset than that on the original dataset. It reduces from 0.138 to 0.108 (Seq2Seq), from 0.188 to 0.151 (SNM), from 0.15 to 0.129 (Tree2Tree), from 0.176 to 0.149 (TRANX), and from 0.167 to 0.155 (CoasetoFine). The same is true for other performance metrics.
- Second, no functionally correct programs could be generated even if the evaluated approaches are fed with the nonredundant dataset.

Based on the preceding analysis, we conclude that removing redundant implementations from the dataset may not improve the performance of code generation.

5.9 Q9: Impact of Unifying Identifiers

To investigate the impact of identifier unification, we unify identifiers in requirements and source code (in the same way

as TRANX unifies identifiers [31]), and repeat the first empirical study as introduced in Section 4.3. First, we replace constant strings (like “URL is required”) that appear in both requirements and associated source code with unified tokens “str_i”. Second, we replace variables that appear in both requirements and associated source code with unified tokens “var_i”. The variables are not further divided according to their types because Python is not a statically typed programming language (like Java). Notably, the same identifier unification is conducted on all of the data. Evaluation results of the identifier unification are presented in Table 15. To facilitate the comparison, we also present the performance of the default setting (i.e., without unifying identifiers). From Table 15, we make the following observations:

- First, unifying identifiers has minor and diverse impact on the performance of the evaluated approaches. For example, it improves the BLEU of TRANX and Tree2Tree slightly from 0.184 to 0.187 and from 0.15 to 0.177, respectively. In the same time, however, it also decreases BLEU of Seq2seq (from 0.138 to 0.135), SNM (from 0.188 to 0.181), and Coarse-to-Fine (from 0.176 to 0.151). Overall, unifying identifiers slightly reduces the average BLEU of the evaluated approaches from 0.167 to 0.166. The same is true for other performance metrics, e.g., syntactically correct programs.
- Second, no functionally correct programs could be generated regardless of the application of unifying identifiers.

6 DISCUSSIONS

6.1 Potential Reasons for Reduced Performance

Evaluation results in Section 5 suggest that switching from existing datasets to ours significantly reduces the performance of existing approaches. Potential reasons are discussed as follows.

First, some special characters in existing datasets facilitate learning-based code generation. For example, the requirements (pseudo-code) in *Django* are quite similar to their implementations. On average, 49.4 percent of the tokens in a program (source code) could be copied from the requirements associated with the program. As a result, learning-based approaches may achieve high performance by copying tokens from requirements to generated programs. In *HS*, different programs are highly similar to each other, which also significantly facilitates code generation. Because of the similarity, learning-based approaches can learn the common structures (also known as templates), and frequently generate source code successfully by ‘*filling learned code templates from training data with arguments copied from input*’ [22].

Second, the requirements in our dataset are much more complex than the existing ones. As discussed in Section 5.6, a great challenge in code generation is natural language understanding (NLU), i.e., to understand requirements. The longer the requirements are, the harder NLU is. Compared to existing datasets, our dataset is composed of much longer and more complex requirements. The average length of such requirements is 185 tokens compared to 14 and 34 in *Django* and *HS*, respectively.

Third, the diversity of our dataset has a significant negative impact on the evaluated approaches. Such approaches have been trained in a specific domain with similar requirements. However, our dataset has very diverse requirements with no common tasks. As a result, except for the generic programming skills (especially algorithm related programming skills), little could be learned about the implementation of specific tasks. However, learning the generic programming skills (i.e., the ability to turn textual requirements into source code as a human developer does) is highly challenging. As a result, the performance of program generation is significantly reduced.

Fourth, the size of our dataset may have prevented the evaluated approaches from reaching their maximal potential. In total, the dataset is composed of 16,673 requirements-code pairs, making it comparable to other data sets that have been employed by the authors of the evaluated approaches. For example, SNM was originally evaluated on JBOS (with 640 items), GEO (with 880 items), ATIS (with 5,373 items), and IFTTT (with 86,960 items), independently. Our dataset is significantly bigger than such datasets except for IFTTT. However, our dataset contains 2,740 unique requirements only, which makes it smaller than ATIS and IFTTT concerning the number of unique requirements. Besides that, the increased complexity of the requirements and source code, together with the limited number of unique requirements, could prevent the evaluated approaches from reaching their maximal potential.

Fifth, our tuning of the hyper parameters for the evaluated approaches could be less effective than the tuning conducted by the original authors of the evaluated approaches. Such approaches have been fine-tuned on given datasets that were leveraged for evaluation by their authors, which often results in high performance on the given dataset. The original tuning is effective because the experts who tuned the parameters were familiar with the approaches. In contrast, we tuned the parameters without deep understanding

of the evaluated approaches, and thus the tuning could be more time-consuming and less effective. This, in turn, prevents the evaluated approaches from reaching their maximal potential.

6.2 Experiment on More Datasets

There is a clear need for an empirical study on various datasets with the proposed approach and evaluate them by comparing it with other approaches. The experiment is conducted on a single dataset that we create in Section 3, which may limit its validity. As introduced in Section 2.2, existing datasets have significant limitations, and thus assessing the state of the art on such datasets may result in severe threats to validity. To this end, we create a new dataset. With this dataset, we assess the state of the art in code generation. To reduce threats to external validity, however, we should conduct similar experiments on other qualified datasets in the future when such datasets are available. Notably, we do not compare the proposed approach (popularity-based code generation) against other approaches on existing datasets. For example, each of the reference programs in *Django* is composed of a single unique statement, which makes it impractical to select the most *popular* statements in the dataset. As a result, the popularity-based approach cannot work on *Django*.

Other threats to validity exist as well, e.g., the size of the involved dataset and the representativeness of the evaluated approaches. The size of the involved dataset may influence the performance of the evaluated approaches. It is likely that increasing the size of the dataset could improve the performance. However, we have not yet investigated its exact influence. Selecting different code generation approaches for the evaluation may result in different conclusions because their performance on the same dataset (i.e., our new dataset) could vary significantly. To reduce the threats to validity, we select multiple state-of-the-art approaches for the evaluation.

6.3 Limited Diversity of the New Dataset

As specified in Section 3, the new dataset is created based on programming contest platforms, which may limit its diversity. Although the programming contest platforms do not post any explicit limitations on the domain of contests, most of the contests concern data structures, sorting algorithms, mathematic computation, text processing, or database management. They are rarely related to any specific application domains, e.g., financial systems, office software, or image processing. As a result, the diversity of the resulting dataset is limited. Approaches trained on such dataset may fail to generate applications whose creation strongly depends on domain knowledge.

Besides the limited diversity, the source code within the dataset could be different from applications in the industry in the following ways. First, most of the code in the new dataset is coded by novice programmers and the skillset levels of these developers are low when compared with industry standards. Second, most of the code written by programmers participating in such contests tend to algorithm driven and end up being implementations of some data structures. Third, real-world systems have a lot of inter-dependencies among the task whereas a majority of tasks in programming contests tend to be orthogonal in nature. Finally, there is lot

of importance given to certain qualities in programming contests which is not necessarily true in real-world systems.

One future work to strengthen the dataset is to exploit additional data sources. Extracting additional data will increase both the size and diversity of the resulting dataset, and thus may help to facilitate the training of deep learning-based code generation models. It is also interesting to include non-English software requirements, and to investigate multiple-language code generation.

Although it is not novel to create dataset by crawling web pages, creating and publishing the dataset is valuable. On one side, the resulting dataset has significant advantages compared to existing ones. On the other side, publishing it releases other researchers from grueling and time-consuming dataset creation.

6.4 Performance Metrics for the Empirical Study

Besides BLEU, we also employ the number of compilation errors, the number of compilation warnings, and the number of failed/passed test cases to assess the quality of generated programs as presented in Table 5. However, such metrics are not suitable for existing datasets (e.g., *Django* and *HS*) because the reference programs (code fragments) within such datasets are incomplete and incompliable. Consequently, it is unfair/unpractical to require models trained on such datasets to generate complete and compilable/runnable programs. However, programs in our new dataset are complete and syntactically correct, and thus we compute such performance metrics for the evaluation on the new dataset.

6.5 Threats to Validity

Besides the threats (limitations) discussed in the preceding sections, the evaluation (especially the case study to evaluate the usefulness of generated programs) is subjected to the following threats to validity. A threat to external validity is that only ten programming tasks and twenty participants were involved in the evaluation. Conclusions drawn on such limited number of subjects may not be generalizable. We failed to increase the number of programming tasks or participants because it is time-consuming for participants to finish the selected programming tasks, and it is challenging for us to recruit a large number of qualified participants. A threat to internal validity is that the observations (coding speed) could be significantly influenced by the characters (e.g., knowledge in Python and programming skills) besides the investigated factor (i.e., with or without the generated programs). To reduce the threat, we recruited thirty participants, excluded the top and bottom ones (concerning their performance) with a pretest, and divided the remaining participants into two independent groups according to their performance in the pretest. As a result, the participants within the same group had similar performance in the pretest.

7 CONCLUSION AND FUTURE WORK

Deep learning-based code generation is potentially promising, and a few approaches have been proposed. Although existing evaluations suggest that such approaches are highly accurate, they are evaluated on small datasets where ‘requirements’ are quite different from real-world requirements in the industry. To assess the state of the art, in this

paper, we build a large scale dataset. Compared to existing ones, it is larger and more diverse. Besides the dataset, we also build an assisting tool to measure the quality of generated programs. We not only compute the widely used plain text-based metrics (BLEU), but also employ syntax sensitive static checking as well as test based dynamic cross-validation. Based on the resulting dataset and assisting tool, we reassess the state of the art in natural language-based program generation. Evaluation results suggest that the state-of-the-art approaches successfully learn to generate popular statements. However, the generated programs are often significantly different from their references. Besides that, they often contain syntactic and semantical errors, and none of them can pass even a single test case. Further analysis suggests that they learn little from the input (requirements). Consequently, to further improve the state of the art, researchers should pay more attention to the encoders of the neural networks that are in charge of requirements’ interpretation. The resulting dataset, the assisting tool, and evaluated approaches (all of them are publicly available at <https://github.com/ds4an/CoDas4CG>) could serve as a basis for future research in this direction.

One future work is to design more effective metrics to assess quality of code generation. It is well-known that BLEU alone is insufficient for assessing the quality of code generation [69] because source code has little tolerance for poor syntax or semantics. To this end, in this paper we propose additional metrics to assess the syntax and semantics of generated programs, i.e., the number of compilation errors, number of compilation warnings, and number of failed/passed test cases. However, as suggested by the empirical study in Section 5, most of the programs generated by the state-of-the-art approaches are not executable, which significantly prevents the proposed execution-based metrics from reaching their maximal potential. Consequently, it remains an open question to design effective metrics in the future to accurately and quantitatively assess the quality of programs automatically generated by the state-of-the-art approaches.

In the future, it is worthwhile to explore larger (not necessarily more complex) datasets to investigate whether the performance of deep learning-based program generation could be improved if there are more sample program implementations available for each task.

It is interesting to change the evaluation setting and repeat the evaluation in the future. The evaluation setting is that some task requirement-implementation pairs are used to train the deep learning models, and use other different task requirements to test if the resulting models can generate useful code. Such a setting is quite realistic but highly challenging. However, if we build a smaller dataset containing similar tasks only, repeating the same evaluation could result in significantly improved performance of the evaluated approaches because in this case the testing tasks are similar to those leveraged for model training.

Finally, further investigation into the weakness of the evaluated approaches could be valuable. In the evaluation, we analyze where and why the evaluated approaches work. However, we have no yet investigated where and why such approaches fail.

ACKNOWLEDGMENTS

The authors would like to thank the associate editor and the anonymous reviewers for their insightful comments and constructive suggestions. This work was sponsored in part by the National Key Research and Development Program of China (2017YFB1001803), the National Natural Science Foundation of China (61772071, 61690205), and the National Science Foundation (CCF-1350487).

REFERENCES

- [1] I. Sommerville, *Software Engineering*. Boston, MA, USA: Addison-Wesley, 1992.
- [2] M. W. Whalen, "An approach to automatic code generation for safety-critical systems," in *Proc. 14th IEEE Int. Conf. Autom. Softw. Eng.*, 1999, pp. 315–318.
- [3] M. W. Whalen, "High-integrity code generation for state-based formalisms," in *Proc. Int. Conf. Softw. Eng.*, 2000, pp. 725–727.
- [4] D. Harel *et al.*, "STATEMATE: A working environment for the development of complex reactive systems," *IEEE Trans. Softw. Eng.*, vol. 16, no. 4, pp. 403–414, Apr. 1990.
- [5] H. Mei and L. Zhang, "Can big data bring a breakthrough for software automation?" *Sci. China Inf. Sci.*, vol. 61, no. 5, 2018, Art. no. 056101.
- [6] G. O'Regan, *Concise Guide to Formal Methods: Theory, Fundamentals and Industry Applications*. Berlin, Germany: Springer, 2017.
- [7] J. M. Wing, "A specifier's introduction to formal methods," *Computer*, vol. 23, no. 9, pp. 8–22, Sep. 1990.
- [8] P. Linz, *An Introduction to Formal Languages and Automata*. Burlington, MA, USA: Jones and Bartlett Learning, 2011.
- [9] R. Soley and the OMG Staff Strategy Group, "Model driven architecture," Object Management Group, Needham, MA, Rep. no. omg/00-11-05, Nov. 2000.
- [10] F. A. Kraemer, "Engineering Android applications based on UML activities," in *Proc. 14th Int. Conf. Model Driven Eng. Lang. Syst.*, 2011, pp. 183–197.
- [11] G. Booch, J. Rumbaugh, and I. Jacobson, *The Unified Modelling Language User Guide*. Boston, MA, USA: Addison-Wesley Professional, 2005.
- [12] G. Sunyé, A. L. Guennec, and J.-M. Jézéquel, "Using UML action semantics for model execution and transformation," *Inf. Syst.*, vol. 27, no. 6, pp. 445–457, 2002. [Online]. Available: <http://www.sciencedirect.com/science/article/pii/S0306437902000145>
- [13] D. A. Dahl *et al.*, "Expanding the scope of the ATIS task: The ATIS-3 corpus," in *Proc. Workshop held at Plainsboro Hum. Lang. Technol.*, 1994, pp. 43–48. [Online]. Available: <http://aclweb.org/anthology/H/H94/H94-1010.pdf>
- [14] W. Ling *et al.*, "Latent predictor networks for code generation," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, 2016, pp. 599–609. [Online]. Available: [http://aclweb.org/anthology/P/P16-1057.pdf](http://aclweb.org/anthology/P/P16/P16-1057.pdf)
- [15] L. R. Tang and R. J. Mooney, "Using multiple clause constructors in inductive logic programming for semantic parsing," in *Proc. 12th Eur. Conf. Mach. Learn.*, 2001, pp. 466–477. [Online]. Available: https://doi.org/10.1007/3-540-44795-4_40
- [16] P. Yin, B. Deng, E. Chen, B. Vasilescu, and G. Neubig, "Learning to mine aligned code and natural language pairs from stack overflow," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 476–486. [Online]. Available: <http://doi.acm.org/10.1145/3196398.3196408>
- [17] Y. Deng, A. Kanervisto, J. Ling, and A. M. Rush, "Image-to-markup generation with coarse-to-fine attention," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 980–989. [Online]. Available: <http://proceedings.mlr.press/v70/deng17a.html>
- [18] C. Chen, T. Su, G. Meng, Z. Xing, and Y. Liu, "From UI design image to GUI skeleton: A neural machine translator to bootstrap mobile GUI implementation," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 665–676. [Online]. Available: <http://doi.acm.org/10.1145/3180155.3180240>
- [19] S. Gulwani, "Automating string processing in spreadsheets using input-output examples," in *Proc. 38th ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 2011, pp. 317–330. [Online]. Available: <http://doi.acm.org/10.1145/1926385.1926423>
- [20] C. Shu and H. Zhang, "Neural programming by example," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 1539–1545.
- [21] J. Dick, E. Hull, and K. Jackson, *Requirements Engineering*. 4th ed., Berlin, Germany: Springer, Aug. 23, 2017.
- [22] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics*, 2017, pp. 440–450. [Online]. Available: <https://doi.org/10.18653/v1/P17-1041>
- [23] K. Papineni, S. Roukos, T. Ward, and W. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics*, 2002, pp. 311–318. [Online]. Available: <http://www.aclweb.org/anthology/P02-1040.pdf>
- [24] Y. Wu *et al.*, "Google's neural machine translation system: Bridging the gap between human and machine translation," *CoRR*, 2016. [Online]. Available: <http://arxiv.org/abs/1609.08144>
- [25] Y. Oda *et al.*, "Learning to generate pseudo-code from source code using statistical machine translation (T)," in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2015, pp. 574–584. [Online]. Available: <https://doi.org/10.1109/ASE.2015.36>
- [26] P. Liang, M. I. Jordan, and D. Klein, "Learning dependency-based compositional semantics," in *Proc. 49th Annu. Meeting Assoc. Comput. Linguistics*, 2011, pp. 590–599. [Online]. Available: <http://www.aclweb.org/anthology/P11-1060>
- [27] A. Wang, T. Kwiatkowski, and L. S. Zettlemoyer, "Morpho-syntactic lexical generalization for CCG semantic parsing," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2014, pp. 1284–1295. [Online]. Available: <http://aclweb.org/anthology/D/D14/D14-1135.pdf>
- [28] L. Dong and M. Lapata, "Language to logical form with neural attention," in *Proc. 54th Annu. Meeting Assoc. Comput. Linguistics*, 2016, pp. 33–43. [Online]. Available: <http://aclweb.org/anthology/P/P16/P16-1004.pdf>
- [29] M. Rabinovich, M. Stern, and D. Klein, "Abstract syntax networks for code generation and semantic parsing," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics*, 2017, pp. 1139–1149. [Online]. Available: <https://doi.org/10.18653/v1/P17-1105>
- [30] A. Stehnii, "Generation of code from text description with syntactic parsing and Tree2Tree model," Master's thesis, Dept. Comput. Sci., Ukrainian Catholic University, Lviv, Ukraine, 2018.
- [31] P. Yin and G. Neubig, "TRANX: A transition-based neural abstract syntax parser for semantic parsing and code generation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2018, pp. 7–12. [Online]. Available: <https://arxiv.org/abs/1810.02720>
- [32] L. Dong and M. Lapata, "Coarse-to-fine decoding for neural semantic parsing," in *Proc. 56th Annu. Meeting Assoc. Comput. Linguistics*, 2018, pp. 731–742. [Online]. Available: <http://aclweb.org/anthology/P18-1068>
- [33] S. A. Hayati, R. Olivier, P. Avvaru, P. Yin, A. Tomasic, and G. Neubig, "Retrieval-based neural code generation," in *Proc. Conf. Empir. Methods Natural Lang. Process.*, 2018, pp. 925–930. [Online]. Available: <https://www.aclweb.org/anthology/D18-1111/>
- [34] Z. Sun, Q. Zhu, L. Mou, Y. Xiong, G. Li, and L. Zhang, "A grammar-based structural CNN decoder for code generation," in *Proc. AAAI Conf. Artif. Intell.*, 2019, vol. 33, pp. 7055–7062.
- [35] T. Gvero and V. Kuncak, "Synthesizing Java expressions from free-form queries," in *Proc. ACM SIGPLAN Int. Conf. Object-Oriented Program. Syst. Lang. Appl.*, 2015, pp. 416–432. [Online]. Available: <http://doi.acm.org/10.1145/2814270.2814295>
- [36] M. Raghethaman, Y. Wei, and Y. Hamadi, "SWIM: Synthesizing what i mean - code search and idiomatic snippet synthesis," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 357–367.
- [37] A. T. Nguyen, P. C. Rigby, T. Nguyen, D. Palani, M. Karanfil, and T. N. Nguyen, "Statistical translation of english texts to API code templates," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2018, pp. 194–205.
- [38] S. Yan, H. Yu, Y. Chen, B. Shen, and L. Jiang, "Are the code snippets what we are searching for? A benchmark and an empirical study on code search with natural-language queries," in *Proc. IEEE 27th Int. Conf. Softw. Anal. Evol. Reengineering*, 2020, pp. 344–354.
- [39] J. M. Zelle and R. J. Mooney, "Learning to parse database queries using inductive logic programming," in *Proc. 13th Nat. Conf. Artif. Intell.*, 1996, pp. 1050–1055. [Online]. Available: <http://www.aaai.org/Library/AAAI/1996/aaai96-156.php>
- [40] C. Quirk, R. J. Mooney, and M. Galley, "Language to code: Learning semantic parsers for if-this-then-that recipes," in *Proc. 53rd Annu. Meeting Assoc. Comput. Linguistics*, 2015, pp. 878–888. [Online]. Available: <http://aclweb.org/anthology/P/P15/P15-1085.pdf>
- [41] Magic the gathering, 2016. [Online]. Available: <http://github.com/magefree/mage/>

- [42] Hearthstone, 2016. [Online]. Available: <http://github.com/danielyule/hearthbreaker/>
- [43] Z. Yao, D. S. Weld, W. Chen, and H. Sun, "StaQC: A systematically mined question-code dataset from stack overflow," in *Proc. World Wide Web Conf. World Wide Web*, 2018, pp. 1693–1703. [Online]. Available: <http://doi.acm.org/10.1145/3178876.3186081>
- [44] Stack Overflow, 2019. [Online]. Available: <https://stackoverflow.com/>
- [45] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *Proc. 35th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2014, pp. 419–428. [Online]. Available: <http://doi.acm.org/10.1145/2594291.2594321>
- [46] J. Li, Y. Wang, M. R. Lyu, and I. King, "Code completion with neural attention and pointer networks," in *Proc. 27th Int. Joint Conf. Artif. Intell.*, 2018, pp. 4159–4165. [Online]. Available: <https://doi.org/10.24963/ijcai.2018/578>
- [47] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 837–847. [Online]. Available: <http://dl.acm.org/citation.cfm?id=2337223.2337322>
- [48] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 38–49. [Online]. Available: <http://doi.acm.org/10.1145/2786805.2786849>
- [49] L. Jiang, H. Liu, and H. Jiang, "Machine learning based automated method name recommendation: How far are we," in *Proc. 34th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2019, pp. 602–614.
- [50] H. Liu, Q. Liu, C.-A. Staicu, M. Pradel, and Y. Luo, "Nomen est Omen: Exploring and exploiting similarities between argument and parameter names," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 1063–1073. [Online]. Available: <http://doi.acm.org/10.1145/2884781.2884841>
- [51] X. Chen, C. Liu, and D. Song, "Towards synthesizing complex programs from input-output examples," in *Proc. Int. Conf. Learn. Representations*, 2018. [Online]. Available: <https://openreview.net/forum?id=Skp1ESxRZ>
- [52] R. Alur, R. Singh, D. Fisman, and A. Solar-Lezama, "Search-based program synthesis," *Commun. ACM*, vol. 61, no. 12, pp. 84–93, Nov. 2018. [Online]. Available: <https://doi.org/10.1145/3208071>
- [53] J. Devlin, J. Uesato, S. Bhupatiraju, R. Singh, A. Mohamed, and P. Kohli, "RobustFill: Neural program learning under noisy I/O," in *Proc. 34th Int. Conf. Mach. Learn.*, 2017, pp. 990–998.
- [54] E. Parisotto, A. Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, "Neuro-symbolic program synthesis," in *Proc. 5th Int. Conf. Learn. Representations*, 2017. [Online]. Available: <https://www.microsoft.com/en-us/research/publication/neuro-symbolic-program-synthesis-2/>
- [55] Y. Feng, R. Martins, J. Van Geffen, I. Dillig, and S. Chaudhuri, "Component-based synthesis of table consolidation and transformation tasks from examples," in *Proc. 38th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2017, pp. 422–436. [Online]. Available: <https://doi.org/10.1145/3062341.3062351>
- [56] Y. Feng, R. Martins, O. Bastani, and I. Dillig, "Program synthesis using conflict-driven learning," in *Proc. 39th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2018, pp. 420–435. [Online]. Available: <https://doi.org/10.1145/3192366.3192382>
- [57] W. Lee, K. Heo, R. Alur, and M. Naik, "Accelerating search-based program synthesis using learned probabilistic models," in *Proc. 39th ACM SIGPLAN Conf. Program. Lang. Des. Implementation*, 2018, pp. 436–449. [Online]. Available: <https://doi.org/10.1145/3192366.3192410>
- [58] R. Shin *et al.*, "Synthetic datasets for neural program synthesis," in *Proc. Int. Conf. Learn. Representations*, 2019. [Online]. Available: <https://openreview.net/forum?id=ryeOSnAqYm>
- [59] M. Balog, A. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "DeepCoder: Learning to write programs," in *Proc. Int. Conf. Learn. Representations*, 2017, pp. 1–20.
- [60] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 783–794.
- [61] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," in *Proc. 46th ACM SIGPLAN Symp. Princ. Program. Lang.*, 2019, pp. 1–29. [Online]. Available: <http://doi.acm.org/10.1145/3290353>
- [62] L. Mou, G. Li, Z. Jin, L. Zhang, and T. Wang, "TBCNN: A tree-based convolutional neural network for programming language processing," in *Proc. 30th AAAI Conf. Artif. Intell.*, 2016, pp. 1287–1293.
- [63] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," *CoRR*, 2017. [Online]. Available: <http://arxiv.org/abs/1711.00740>
- [64] Codeforces, 2019. [Online]. Available: <http://codeforces.com/>
- [65] HackerEarth, 2019. [Online]. Available: <https://www.hackerearth.com/>
- [66] G. S. Manku, A. Jain, and A. D. Sarma, "Detecting near-duplicates for web crawling," in *Proc. 16th Int. Conf. World Wide Web*, 2007, pp. 141–150. [Online]. Available: <http://doi.acm.org/10.1145/1242572.1242592>
- [67] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," *Soviet Phys. Doklady*, vol. 10, no. 8, pp. 707–710, 1966.
- [68] D. Liu and D. Gildea, "Syntactic features for evaluation of machine translation," in *Proc. Workshop Intrinsic Extrinsic Eval. Measures Mach. Transl. Summarization*, 2005, pp. 25–32.
- [69] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *Proc. ACM Int. Symp. New Ideas New Paradigms Reflections Program. Softw.*, 2014, pp. 173–184. [Online]. Available: <http://doi.acm.org/10.1145/2661136.2661148>
- [70] S. Bird, E. Klein, and E. Loper, *Natural Language Processing with Python*. Sebastopol, CA, USA: O'Reilly, 2009. [Online]. Available: <http://www.oreilly.de/catalog/9780596516499/index.html>
- [71] Natural language toolkit, 2020. [Online]. Available: <https://github.com/nltk/nltk/>
- [72] autopep8, 2018. [Online]. Available: <https://pypi.org/project/autopep8/>
- [73] P. M. Lerman, "Fitting segmented regression models by grid search," *Appl. Statist.*, vol. 29, no. 1, pp. 77–84, 1980.
- [74] Pylint, 2018. [Online]. Available: <https://www.pylint.org/>
- [75] Python documents, 2019. [Online]. Available: <https://docs.python.org/3/library/exceptions.html>



Hui Liu received the BS degree in control science from Shandong University, China, in 2001, the MS degree in computer science from Shanghai University, China, in 2004, and the PhD degree in computer science from the Peking University, China, in 2008. He is a professor with the School of Computer Science and Technology, Beijing Institute of Technology, China. He was a visiting research fellow in centre for research on evolution, search and testing (CREST) at University College London, United Kingdom. He served on the program committees and organizing committees of prestigious conferences, such as ICSME, RE, ICSR, and COMPSAC. He is serving as associate editor for the *IET Software*, and guest editor for the *Empirical Software Engineering* and the *Journal of Systems and Software*. He is particularly interested in deep learning-based software engineering, software refactoring, and software quality. He is also interested in developing practical tools to assist software engineers.



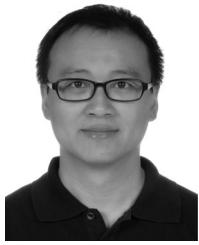
Minzhu Shen received the BS degree from the information management and system program, Northwest A&F University, China, in 2018. She is currently working toward the master's degree in the School of Computer Science and Technology, Beijing Institute of Technology, China, under the supervision of Dr. Hui Liu. Her current research interests include software testing and AI-based software engineering.



Jiaqi Zhu received the BS degree from the College of Information Engineering, Northwest A&F University, China, in 2017. She is currently working toward the master's degree in the School of Computer Science and Technology, Beijing Institute of Technology, China, under the supervision of Dr. Hui Liu. Her current research interests include code generation and software evolution.



Nan Niu received the BEng degree in computer science and engineering from the Beijing Institute of Technology, Beijing, China, in 1999, the MSc degree in computing science from the University of Alberta, Edmonton, AB, Canada, in 2004, and the PhD degree in computer science from the University of Toronto, Toronto, ON, Canada, in 2009. He is currently an associate professor with the Department of Electrical Engineering and Computer Science, University of Cincinnati, Cincinnati, Ohio. His current research interests include software requirements engineering, information seeking in software engineering, and human-centric computing. He was a recipient of the U.S. National Science Foundation Faculty Early Career Development (CAREER) Award, the IEEE International Requirements Engineering Conference's Best Research Paper Award, in 2016, and the Most Influential Paper Award, in 2018.



Ge Li received the PhD degree from Peking University, China, in 2006, and had been a visiting associate professor at Stanford University, Stanford, California, in 2013-2014. He is an associate professor with the Department of Computer Science and Technology, School of EECS. He is currently the deputy secretary general of CCF Software Engineering Society and the founder of the Software Program Generation Study Group. He was one of the earliest researchers engaged in the study of the computer program language model based on deep neural network, and the study of end-to-end program code generating techniques. His current research mainly concerns applications of probabilistic methods for machine learning, including program language process, natural language process, and software engineering.



Lu Zhang received the both BSc and PhD degrees in computer science from Peking University, China, in 1995 and 2000 respectively. He is a professor with the School of Electronics Engineering and Computer Science, Peking University, P.R. China. He was a postdoctoral researcher in Oxford Brookes University and University of Liverpool, United Kingdom. He served on the program committees of many prestigious conferences, such as FSE, OOPSLA, ISSTA, and ASE. He was a program co-chair of ICSM17. He has been on the editorial boards of the *Journal of Software Maintenance and Evolution: Research and Practice* and the *Software Testing, Verification and Reliability*. His current research interests include software testing and analysis, program comprehension, software maintenance and evolution, software reuse, and program synthesis.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

Clustering Crowdsourced Test Reports of Mobile Applications Using Image Understanding

Di Liu, Yang Feng^{ID}, Xiaofang Zhang^{ID}, James A. Jones, and Zhenyu Chen^{ID}

Abstract—Crowdsourced testing has been widely used to improve software quality as it can detect various bugs and simulate real usage scenarios. Crowdsourced workers perform tasks on crowdsourcing platforms and present their experiences as test reports, which naturally generates an overwhelming number of test reports. Therefore, inspecting these reports becomes a time-consuming yet inevitable task. In recent years, many text-based prioritization and clustering techniques have been proposed to address this challenge. However, in mobile testing, test reports often consist of only short test descriptions but rich screenshots. Compared with the uncertainty of textual information, well-defined screenshots can often adequately express the mobile application's activity views. In this paper, by employing image-understanding techniques, we propose an approach for clustering crowdsourced test reports of mobile applications based on both textual and image features to assist the inspection procedure. We employ Spatial Pyramid Matching (SPM) to measure the similarity of the screenshots and use the natural-language-processing techniques to compute the textual distance of test reports. To validate our approach, we conducted an experiment on 6 industrial crowdsourced projects that contain more than 1600 test reports and 1400 screenshots. The results show that our approach is capable of outperforming the baselines by up to 37 percent regarding the APFD metric. Further, we analyze the parameter sensitivity of our approach and discuss the settings for different application scenarios.

Index Terms—Crowdsourced testing, mobile testing, test report processing

1 INTRODUCTION

CROWDSOURCED testing has received extensive attention from the industrial practice. By employing a large population of crowd workers in a short period, crowdsourced testing is capable of simulating the various real usage scenarios and providing feedback of real users [1]. These advantages are particularly suitable for mobile application testing, which has a rapid development cycle, and also requires testing on a diverse set of applications, hardware, and software platforms. Under this situation, several commercial crowdsourced testing platforms, such as uTest,¹ Testin,² Testflight,³ Baidu Crowd Test,⁴ Alibaba Crowd Test,⁵ and TestIO,⁶ have emerged in recent years.

Typically, in crowdsourced testing, crowd workers perform micro-tasks that are posted on the crowdsourcing platforms by the requesters, and are required to provide their feedback in the form of test reports. Because the crowdsourced technique inherently depends on a large workforce, requesters naturally receive massive amounts of reports, and thus, inspecting and understanding these test reports becomes a tedious yet inevitable task.

In the past decades, to mitigate this problem, software engineering researchers have proposed many techniques to identify and group similar reports. These works can be classified into two categories based on information they employed to measure the similarity between reports. The first category focuses on natural language information and leverages text-analysis techniques, such as language modeling [2], [3], text mining [4], topic modeling [5], and information retrieval [6], [7], to identify similar reports. Many later works in this direction tried to enhance the accuracy of similar report detection based on metrics on textual similarity [8], [9], identification strategies [10], [11], [12], or extra information [13]. On the other hand, as many software applications, e.g., Microsoft Windows, Firefox, and Internet Explorer, have provided features to automatically record execution traces for field bugs and send reports to their producers, it is natural to use such execution information to identify similar reports. Execution traces mainly contain dynamic behaviors of the program, like stack trace, branch, or statement coverage. Similar failing traces imply the same bug [14]. Based on execution traces, researchers have designed a number of models with supervised or unsupervised learning techniques [15], [16], [17], [18], [19], [20]. These models can identify failure reports with similar causes, and the classification results can be helpful for diagnosing the frequency and severity of these reports.

1. <https://www.utest.com>, Latest access on 1st, Mar, 2020.
 2. <https://www.testin.cn>, Latest access on 1st, Mar, 2020.
 3. <https://developer.apple.com/testflight>, Latest access on 1st, Mar, 2020.
 4. <http://test.baidu.com>, Latest access on 1st, Mar, 2020.
 5. <https://mqc.aliyun.com>, Latest access on 1st, Mar, 2020.
 6. <https://test.io>, Latest access on 1st, Mar, 2020.

- Di Liu and Xiaofang Zhang are with School of Computer Science and Technology, Soochow University, Suzhou, Jiangsu 215006, China. E-mail: dliu0721@suda.edu.cn, xfzhang@suda.edu.cn.
- James A. Jones is with the Department of Informatics, University of California, Irvine, Irvine, CA 92697 USA. E-mail: jajones@uci.edu.
- Yang Feng is with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China. E-mail: charles.fengy@gmail.com.
- Zhenyu Chen is with the State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing, China, and also with the Mootest Inc., Nanjing 210000, China. E-mail: zychen@nju.edu.cn.

Manuscript received 30 Nov. 2019; revised 17 July 2020; accepted 5 Aug. 2020. Date of publication 24 Aug. 2020; date of current version 18 Apr. 2022. (Corresponding author: Xiaofang Zhang and Yang Feng.) Recommended for acceptance by D. Hao. Digital Object Identifier no. 10.1109/TSE.2020.3017514

Even though these studies have significantly improved the efficiency of dealing with test reports, they are often difficult to apply to the specific domain of mobile application testing. Crowdsourced workers often prefer to take a screenshot of the problematic activity view, which is relatively easy on most mobile devices, rather than type a long bug description, which is more difficult using the restricted small keyboard of most mobile devices. Due to these factors, the textual information of crowdsourced mobile test reports often lack sufficient details and accuracy, and execution traces are also difficult to collect due to the limited processing capability, storage, battery power, and communication of many mobile devices. While text-analysis-based methods become less effective because of short and inaccurate text descriptions, automatically identifying information from screenshots becomes critical for developers to understand reports. Images are considered to be one of the most essential and convenient communication carriers on the mobile platform [21], [22]. Moreover, in comparison with desktop applications, screenshots of the mobile application are often well-defined and describe the activity views. The resolutions, layouts, and even features of these images are limited in a given scope, unlike desktop applications, where windows can often be reshaped and occluded.

Inspired by these factors, we proposed an approach of test-report prioritization that utilizes both textual and image information, which was presented at the 31st IEEE/ACM International Conference on Automated Software Engineering (ASE'2016) [23]. Yang *et al.* proposed the basic idea of clustering test reports by leveraging both textual and image information [24]. However, their work did not introduce the method to further process the clustered test reports in practical scenarios thus failed to provide developers a complete test report processing technique. Therefore, in this paper, we extend our conference paper and provide a fully automatic test-report-clustering technique that utilizes image understanding. In addition, to help users apply this technique in practice, we not only study the accuracy of clustering but also consider the practical usefulness and parameter selection in real scenarios.

To achieve this goal, we capture information from text descriptions and screenshots, then calculate the distance between each pair of reports for clustering. For the image analysis, we employ the Spatial Pyramid Matching (SPM) [25] technique and chi-square metric to measure the similarity among screenshots. For the text analysis, we use classic natural-language-processing techniques to measure the similarity of textual description between reports. Then, we use the balanced formula of the multi-object optimization algorithm to generate a hybrid distance matrix for all test reports. Finally, based on the distance matrix, we use hierarchical agglomerative clustering for report aggregation. As a result, each cluster contains representative information, and requesting testers can sample from these clusters to quickly obtain unique bugs and reduce the time of report inspection.

To validate our clustering technique, we conducted experiments on 6 industrial mobile crowdsourced projects, which constitute more than 1600 test reports and 1400 screenshots. The experimental results show that our hybrid technique achieves an average accuracy of more than 80 percent, which is superior to both only-text-based and only-

image-based methods. Based on the result of clustering, we conducted an empirical study on random sampling, and employ hybrid test-report prioritization technique [23] as another baseline. The results also show that our technique is the closest to the *Ideal* strategy and achieves the most significant improvement over *Random* strategy using the Average Percentage of Faults Detected (APFD) metric [26].

The main contributions of this paper are as follows:

- We proposed a technique to cluster crowdsourced test reports of mobile applications based on both the textual descriptions and screenshots by leveraging the image understanding techniques.
- We collaborated with six companies and collected six industrial projects that contain more than 1600 tests reports and 1400 screenshots. We outsourced this dataset as a benchmark to broaden the research of this topic.⁷
- We empirically evaluated our technique based on this dataset and compare it with the existing methods. The experimental results show that our technique can effectively shorten the inspection procedure, and outperforms existing approaches.
- To help users adapt our technique into practice, we comprehensively analyze our technique regarding different parameter settings. Based on the empirical results, we present guidance and suggestions for applying our technique under different scenarios.

This paper extends our prior conference publication [23], new materials in comparison with the conference version include:

- We present a new image-understanding-based clustering technique to improve the efficiency of inspecting crowdsourced test reports. In contrast to our prior work [23], which requires developers to inspect *all* crowdsourced test report, this new technique enables crowdsourced requesters to identify *a representative portion* out of the whole set.
- We collaborate with six software testing companies to conduct an empirical study to validate the crowdsourced test report clustering technique. Also, we outsource this dataset to assist researchers in reproducing our technique and further improving it. In addition, four hypothesis, statistical testings, and effect size analysis are conducted to measure and demonstrate the significance of the outperformance of the new technique.
- New experiments on large-scale crowdsourced testing projects and the corresponding results are provided. In comparison with our conference paper, we evaluate the impacts of parameters on the performance of our technique. The details are described in Section 5.

The remainder of this paper is organized as follows: In Section 2, we introduce the technical background knowledge and discuss the challenges. In Section 3, we present the details of our technique framework. In Section 4, we evaluate our technical framework and introduce the experimental

7. <https://bitbucket.org/TSE2020/dataset-of-cst2016>

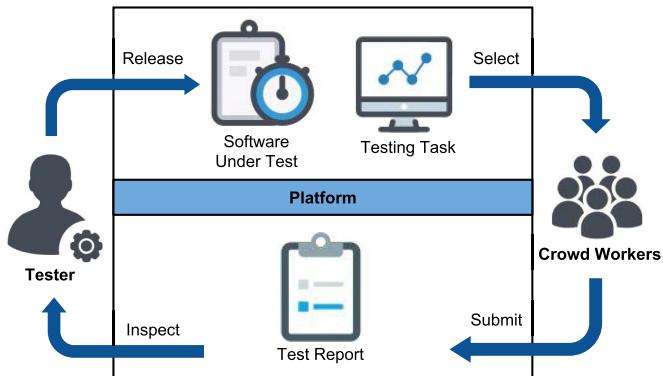


Fig. 1. Procedure of crowdsourced testing.

setup. In Section 5, we discuss the experiment results and analyze the answers to research questions. In Section 6, we discuss threats to validity of our technical framework and validation. In Section 7, we review prior related research on this topic. In Section 8, we draw conclusions and outline some directions for future work.

2 CROWDSOURCED TESTING FOR MOBILE APPLICATION

Crowdsourced testing involves three different related stakeholders [1]. Fig. 1 shows the practical procedure of crowdsourced testing.

Companies and organizations often play the role of requesters. They release testing tasks and software under test on the crowdsourced testing platform. Also, they set up constraints regarding testing resources, environments, and technical requirements. Based on these constraints, platforms can match proper crowd workers for these tasks, and conversely, crowd workers can also find tasks of interests. In recent years, many crowdsourcing platforms have been built for software testing tasks. Table 1 gives an overview of some popular industrial crowdsourced software testing platforms. By inspecting this table, we can observe the testing targets of these platforms vary widely, including performance testing, security testing, functional testing and usability testing, and some of these platforms provide the functionality to facilitate the testing process and manage the test reports. They not only enable the communication between requesters and crowd workers but also significantly improve the efficiency of each part of the whole process.

Even though these platforms lay the infrastructure for flourishing crowdsourced software testing, some features of

test reports bring challenges into the inspection procedure. Crowdsourced testing is widespread in mobile application testing because it enables developers to evaluate the performance of their software products under real usage scenarios, which includes a diverse set of mobile devices and OS versions. However, in practice, crowd workers are often required to finish crowdsourcing tasks in a given short time, and the number of complete tasks influences the rewards for the crowd workers. As such, crowd workers are less apt to filter out duplicates actively, and they are incentivized to submit as many reports as possible. These factors contribute to crowdsourced testing to contain a higher duplicate ratio than conventional testing [27], [28].

Further, on almost all mobile devices, images have played a crucial role in sharing, expressing, and exchanging information. Zhang *et al.*'s research [27] finds test reports of mobile applications contain much shorter text descriptions and more screenshots in comparison with the test reports of desktop applications. To illustrate, we present four raw crowdsourced test reports of three different mobile applications in Table 2. In this table, we can observe reporters present the reports with a brief textual description and several screenshots to describe the bug. Consider the test report in the second row: The crowdsourced tester was testing the course-evaluation view of the application "HJ Normandy." She reported that the application crashed after she entered a large number of emoji characters in the textbox and clicked send button. Even though the tester briefly describes the situation with text in the description section, the critical information for reproducing the bug, such as which emoji characters are used, how many times they were used, and what the context is, can only be obtained from the screenshots.

For testing mobile applications, testers tend to describe bugs with a direct screenshot and short descriptions rather than tedious and complicated text descriptions, mainly due to the ease of taking screenshots and the relative difficulty in typing longer descriptions on mobile virtual keyboards [27]. The shortage of the textual description hinders applying the text-analysis-based test-report-processing techniques for mobile crowdsourced testing. While the different knowledge background and software configurations of crowd workers make it difficult for developers to fully understand the textual content of test reports, compared with the text description, screenshots can objectively describe the operation steps and GUI exceptions, which makes the test reports more readable. These screenshots of mobile applications are usually well-defined application views, and do not suffer as many of the difficulties of desktop application screenshots, such as varying resolutions, scaling, occlusion, and window sizes. In

TABLE 1
Popular Crowdsourcing Platforms

Name	Site	Testing Domain
uTest	utest.com	performance testing, security testing, test case management, test case recording
Testin	testin.cn	functional testing, performance testing, intelligent hardware testing
TestIO	test.io	functional testing, exploratory testing, wearables testing, IoT testing
Testflight	developer.apple.com/testflight/	functional testing, usability testing, performance testing
Bugcrowd	bugcrowd.com	security testing
Baidu Crowd Test	test.baidu.com	functional testing, usability testing, performance testing, text&image annotation
Alibaba Crowd Test	mqc.aliyun.com	performance testing, compatibility testing, test case recording
Tencent Crowd Test	task.qq.com	functional testing, usability testing, performance testing

TABLE 2
Example Test Reports

Project	Bug Description	Bug Screenshots
HJ Normandy	When entering the user name during registration, the application can't jump normally after clicking confirm. It stays on the loading page.	
HJ Normandy	Entering a large number of emoji on the course evaluation interface, then click the send button. The program crashed directly without any prompt for success or failure.	
HW Health	When setting the nickname, the nickname can be set as a space character, but after clicking the confirm button, the nickname is still not set.	
Wonderland	Select "my itinerary" → select "shopping cart", add a itinerary → set the hotel price to over 600 CNY → the program will jump back to the interface shown in the second screenshot.	

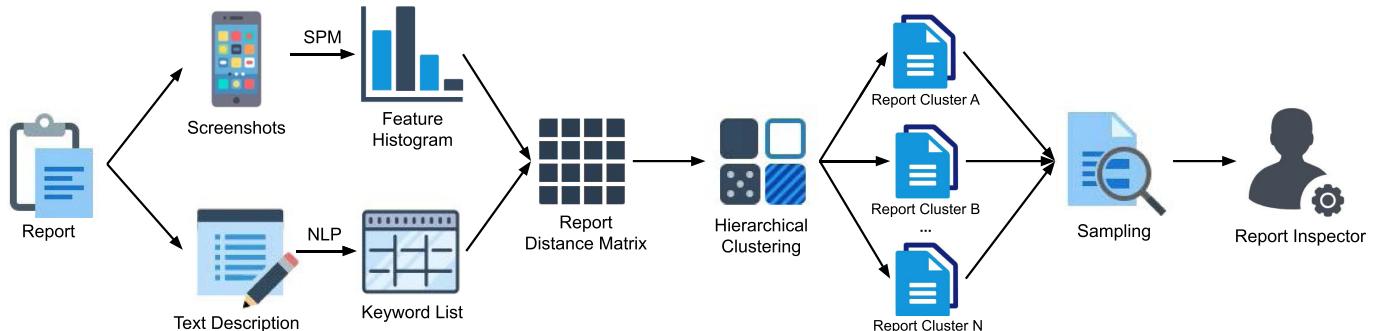


Fig. 2. Test-report clustering processing.

this context, the above issues are ameliorated, and the main problems are the prompting of the error-message dialogs, shifting of GUI elements or the fact that some elements are not displayed at all. These facts motivate us to propose an automated technique to extract the information from the screenshot of crowdsourced test reports of mobile applications to assist the procedure of test report inspection for developers.

3 APPROACH

This section details the design of our technique. We present the framework of our technique in Fig. 2. The framework consists of four major phases: (1) text processing, (2) screenshot processing, (3) balanced distance calculation, and (4) test report clustering. Each of these major phases contains several steps.

3.1 Preliminary

In the crowdsourced testing of mobile applications, test reports may contain various multimedia information, such

as voice messages and video operations. Among this information, text descriptions and screenshots are the most widely used and informative for fault diagnosis and debugging. Therefore, we mainly focus on the analysis of these two kinds of information. We assume that each test report only consists of two parts: a text description and a collection of screenshots. Thus, we can denote the test report set as $R = \{r_i(T_i, S_i) | i = 0 \dots N\}$. In a test report r_i , T_i denotes the raw description of bug performance and operation steps, S_i denotes the screenshots capturing the views of bug details. In practice, each test report often contains more than one screenshot, i.e., for the screenshot set S_i of test report r_i , we have $S_i = s_{i1}, s_{i2}, \dots, s_{im}$, in which, s_{ij} denotes the j th screenshot in test report r_i .

3.2 Text Processing

Natural-language-processing techniques have been widely used to assist various tasks in the field of software testing [2], [13], [17], [29], [30]. In text processing, we focus on applying existing mature natural-language-processing tools

to extract textual features of the bug descriptions from test reports. This process is composed of two steps: (1) keyword vector building, and (2) text distance calculation.

3.2.1 Keyword Vector Building

To extract critical information from the textual description, we model the textual description into keyword vectors. This modeling procedure consists of segmentation, stop word removal, and part-of-speech filtering.

In the first step, we employ Jieba,⁸ a lightweight Python-based word segmentation system to tokenize the text description. Jieba uses dynamic programming to find the most probable combination based on the word frequency, and tokenize each word with its part-of-speech (POS). After the word segmentation, the raw words stream still contains noise, such as spelling mistakes and uncommon words. The existing studies have shown that this phenomenon can be ameliorated by extracting only verbs and nouns [31], [32]. Hence, we retain only the verbs and nouns and filter out the other words. Similarly, we also filter out the stop words based on the ICTCLAS stop word list⁹ to obtain keywords. Then, we can build up the corpus V of test report set R by applying this method to each text description T_i of r_i and adding the keywords into it. Assuming we have q keywords in total, then we can denote the frequency of j th keyword appears in test report r_i as v_{ij} , and the corresponding keyword vector can be denoted as $KV_i = \{v_{ij} | j = 0 \dots q\}$.

Because the text analysis of mobile crowdsourcing test reports is not the focus of this research, we adopt the common word segmentation and stop word removal steps that are widely adopted in many research of software engineering, such as [2], [13], [17], [29], [30]. This method is not limited to processing Chinese. By applying other NLP tools, such as the Stanford NLP toolkit¹⁰, NLTK,¹¹ it can build up keyword vectors for other kinds of natural languages, such as French, Arabic or German.

3.2.2 Text Distance Calculation

In computing the distance between these textual descriptions, we employ the *TF-IDF* to weight keywords. We treat each $KV_i(v)$ as the *document* and keywords $\{v_{ij} | j = 0 \dots q\}$ as the *terms*. Given this context, we adopt *log normalization* to calculate the term-frequency weight (*tf*). For the j th keyword of test report r_i , the term-frequency weight can be computed based on

$$tf_{ij} = \log(1 + v_{ij}). \quad (1)$$

Regarding the inverse-document frequency(*idf*), we adopt the inverse-document-frequency-smooth weighting scheme. Similarly, for the j th keyword of test report r_i , the inverse-document frequency can be computed based on Equation (2). In Equation (2), df_{ij} denotes the number of keyword vector that contains the term v_{ij} and N is the number of keyword vectors.

$$idf_{ij} = \log \left(1 + \frac{N}{df_{ij}} \right). \quad (2)$$

Then, the final *TF-IDF* weight for the j th keyword of test report KV_i is shown in

$$w_{ij} = \text{tf-idf}_{ij} = \log(1 + v_{ij}) \cdot \log \left(1 + \frac{N}{df_{ij}} \right). \quad (3)$$

In the implementation, for each pair of test reports (r_i, r_j) , we employ the euclidean distance between their *weighted keyword vectors* as the textual distance $DT(r_i, r_j)$. Then, we can have Equation (4) to compute the textual distance between each pair of test reports $DT(r_i, r_j)$.

$$DT(r_i, r_j) = \sqrt{\sum_{k=1}^q (w_{ik} - w_{jk})^2}. \quad (4)$$

3.3 Screenshot Processing

Zhang *et al.* found that one primary feature of mobile testing is that the test reports for mobile applications contain much shorter textual description yet plentiful screenshots in comparison with conventional desktop applications [27]. Considering this, we adopt an image-understanding technique to analyze features within screenshots. For mobile applications, screenshots of an activity view can reflect the functionality because the functional requirements of mobile applications are often implemented with one or multiple activity views. Our approach seeks to assess the similarity of such screenshots and leverage this information to measure the distance between test reports. In this paper, the screenshot processing consists of two fundamental sub-steps: (1) feature histogram building, and (2) screenshot distance calculation.

3.3.1 Feature Histogram Building

In crowdsourced testing, the target application can be tested on different devices and under various software circumstance settings. Although developers well design the layouts, widgets, corresponding events, and actions for each activity view of mobile applications, these activity views can be shown with various resolutions, color contrast and customized appearance settings.

However, conventional image recognition techniques, such as Bag Of Features (BOF) [33], focuses on the distribution characteristics of global features in the whole image, while neglecting the position information of global features. Thus, the feature modeling methods that are designed merely based on naive RGB values would suffer from noises. To overcome this challenge, we employ the Spatial Pyramid Matching (SPM) [25] technique to extract the descriptors of scale-invariant feature transform (SIFT) from screenshots.

SPM takes a set of reference images as inputs, and it partitions each reference image into a number of sub-regions in a pyramid fashion. It computes an orderless histogram of low-level features in each sub-regions of each pyramid level. With this decomposition, SPM is capable of concatenating statistics of local features and summarizing this information into descriptors. The descriptors of SIFT present the *local* key

8. <https://github.com/fxsjy/jieba>, Latest access on 1st, Mar, 2020.

9. <http://ictclas.nlpir.org>, Latest access on 1st, Mar, 2020.

10. <http://nlp.stanford.edu/software>, Latest access on 1st, Mar, 2020.

11. <https://www.nltk.org>, Latest access on 1st, Mar, 2020.

points of reference images. They are stored into a database after being extracted and then are used to find candidate matching feature descriptors from new images. Because the descriptor of SIFT is invariant against various transformations, such as uniform scaling, orientation, illumination changes, and partially invariant to affine distortion, it well fits for measuring the difference between screenshots of mobile applications. SIFT features can reflect the GUI structure information that is often more useful than the RGB distribution in representing the characteristics of operations. The spatial pyramid is a simple and computationally efficient extension of the orderless BOF image representation to include spatial orientation [25].

Histograms are introduced as a practical and reliable means for image description in computer vision [34]. At each pyramid level, the technique computes an orderless histogram of low-level features in each sub-region. The histograms generated by spatial pyramid approach can be regarded as an alternative formulation of a locally orderless image, which defines a fixed hierarchy of rectangular windows.

Note that the SPM algorithm is designed for recognizing natural scenes. In the original paper [25], its inventors input a number of images describing the natural scenes as the reference images. The algorithm then extracts local features from these images and clusters these local features to group the features describing the same scene together. For a new coming image, it can recognize the scene by comparing its features with each feature cluster. However, our goal is to group the screenshots. Thus, in our technique, the inputs for the SPM algorithm are all screenshots of each project. It can group the screenshots that have similar SIFT features into the same cluster.

3.3.2 Screenshot Distance Calculation

After obtaining the feature histogram of the screenshots, we can compute the distance between each pair of screenshots. SPM can be configured to output equal number of feature statistics for each screenshot (partially assisted by the resizing normalization described in Section 3.3.1), and thus these output histograms for each screenshot contains the same number of bins. As such, we adopt the chi-square distance metric, a generally used method to compute the distance between two normalized histograms [35]. In our implementation, the equation of chi-square distance is as follows:

$$\begin{aligned} DS(s_i, s_j) &= \chi^2(H_i, H_j) \\ &= \frac{1}{2} \sum_{k=1}^n \frac{(H_i(f_k) - H_j(f_k))^2}{H_i(f_k) + H_j(f_k)}. \end{aligned} \quad (5)$$

In Equation (5), $DS(s_i, s_j)$ denotes the distance between screenshot s_i and s_j , where $H_i(f_1, f_2, \dots, f_n)$ denotes the feature histogram of screenshot s_i , and $H_i(f_k)$ denotes the k th feature of s_i .

However, each test report may contain more than one screenshot. For each pair of test reports (r_i, r_j) , we need to measure the distance between two sets of screenshots.

Note that even though two screenshots are taken from the same activity view, the distance between them may not be zero. This is because mobile applications are designed for running on diverse devices. With various resolutions, brightness,

contrast, and user settings, some minor differences could exist on the screenshot. We set a threshold γ to identify screenshots that are taken from the same activity view. For a pair of screenshots (s_i, s_j) , if $DS(s_i, s_j) \leq \gamma$, we consider they are taken from the same activity view of the application, and they would fit in the intersection of two sets of screenshots. Because the SPM algorithm is designed to capture the local feature that reflects the layout information, the distance between screenshots taken from the same view should be very small. Considering this reason, we recommend the users to set γ into a relatively small value, e.g., 0.1, when applying our technique in practice. By this means, we can obtain the intersection between the screenshot sets, and define the union as the sum of two sets and subtract the intersection.

Given this definition, we can employ Jaccard distance to measure the distance between two sets of screenshots. The distance between screenshots of two reports, reports r_i and r_j , is defined as following:

$$DS(r_i, r_j) = 1 - \frac{|S_i \cap S_j|}{|S_i \cup S_j|}. \quad (6)$$

In Equation (6), S_i and S_j represent the screenshot sets of reports r_i and r_j respectively. A particular case is when either S_i or S_j is an empty set. For this case, we assess $DS(r_i, r_j) = 1$ to maximize the distance.

3.4 Balance Distance Calculation

Based on the above distance calculations, we combine textual distance DT and screenshot distance DS to a balanced distance BD . The balanced distance BD defined in Equation (7) represents the overall distance between test report r_i and r_j .

$$BD(r_i, r_j) = \begin{cases} 0, & \text{if } DT(r_i, r_j) = 0 \\ \alpha \times DT(r_i, r_j), & \text{if } DT(r_i, r_j) \neq 0 \text{ and } DS(r_i, r_j) = 0. \\ (1 + \beta^2) \frac{DS(r_i, r_j) \times DT(r_i, r_j)}{\beta^2 DS(r_i, r_j) + DT(r_i, r_j)}, & \text{if } DT(r_i, r_j) \neq 0 \text{ and } DS(r_i, r_j) \neq 0 \end{cases} \quad (7)$$

The first branch of Equation (7) shows the condition when $DT(r_i, r_j)$ equals 0, which means the two test reports r_i and r_j contain exact same weighted keywords. Given the fact that inputting textual descriptions on mobile devices often requires more efforts in comparison with the desktop devices, these texts are usually short in length yet more meaningful in describing the buggy situation. Thus, if two reports contain the identical keyword sets, they likely describe the same bug. We define $BD(r_i, r_j) = 0$ if $DT(r_i, r_j) = 0$, no matter whether $DS(r_i, r_j) = 0$ or not.

The second branch of Equation (7) shows the condition when $DS(r_i, r_j) = 0$ and $DT(r_i, r_j) \neq 0$, which means the two test reports r_i and r_j contain exact same screenshots but different keywords. Considering screenshots within a test report represent the operations contained in the testing, $DS(r_i, r_j) = 0$ indicates the two reporters have conducted the same operations in the testing. Thus, we presume the text in these two reports may describe the same bug, and

define a factor α to scale the weight of $DT(r_i, r_j)$ in case the distance becomes a large number when reporters express their situation in totally different words. However, because these screenshots are often taken on various devices and platforms, it is very rare that the screenshot distance equates zero. This means α maybe not capable of fundamentally influencing the performance of our technique. Considering this, we recommend end users to set α into a fixed value, e.g., 0.8, when applying our technique in practice.

The last branch of Equation (7) is designed for the most common case, i.e., when $DS(r_i, r_j) \neq 0$ and $DT(r_i, r_j) \neq 0$ at the same time. We introduce the harmonic mean to calculate the BD based on textual distance DT and screenshot set distance DS . Another parameter β is designed to control the impact of the text and screenshots on the balanced distance. When β is set to 1, the weight of text distance and image distance is equal; When β is greater than 1, the image distance has a greater weight; When β is smaller than 1, the text distance has a greater weight.

The balanced distance BD quantitatively assesses the difference between two test reports and takes both text and image features into consideration. Similar to $DT(r_i, r_j)$ and $DS(r_i, r_j)$, the smaller the value of $BD(r_i, r_j)$ is, the more similar the report r_i and r_j are.

3.5 Test-Report Clustering and Sampling

Using the previous methods of obtaining a balanced distance matrix for all test reports, we can cluster the reports into groups. Note that the number of both reported bugs and received test reports is often unpredictable in practice, we utilize hierarchical agglomerative clustering(HAC) [36], which does not require the user to specify the number of clusters, to group these reports. Agglomerative hierarchical clustering can be considered as a bottom-top process of establishing a tree diagram of data instances. HAC starts with grouping each data instance into a cluster. And then, it agglomerates the closest pair of clusters based on the distance measurement. The whole aggregation process repeats until the minimum distance between the clusters reaches the user-defined threshold ε . If the minimal distance between these pairs is larger than the threshold ε , the clustering procedure terminates; otherwise, the algorithm agglomerates the two closest clusters.

Besides the threshold ε controlling the stop point of clustering, the linkage type, which defines the method of calculating the distance between clusters, influences the clustering result. There are three types of linkage in agglomerative hierarchical clustering, i.e., single-linkage, complete-linkage, and average-linkage [37].

The single-linkage measures the distance between two clusters as the distance of their closest members. It focuses on the area where the two clusters lay closest to each other instead of overall structure. In complete-linkage, the distance between two clusters is measured based on their farthest members. It presents the global, and the entire structure of the data set can influence aggregation procedure. Average-linkage measures the distance between two clusters based on the average distance between each member in one cluster to the members in the other cluster. In our approach, we adopt the average-linkage type because it properly takes the dissimilarity of all members within clusters into consideration.

HAC is capable of grouping the test reports describing similar bugs. Based on the grouped results, the users of our technique can reach an overview understanding of all reports by sampling from each cluster and thus save the cost of processing them. We first sort clusters into a list based on their size ascendingly. And then we adopt an iterative sampling strategy to capture representatives from each cluster. In each iteration, a few reports are randomly sampled from each cluster within the cluster list. The sample size is determined by the size of the cluster, and the user-defined ratio parameter ρ . In the implementation it is round up to nearest integer value. After obtained samples, the users of our technique can conduct a manual inspection. This operation of sorting clusters enables users to inspect representative reports from smallest clusters that report the unique bug, and thus help them to identify as many bugs as possible early. The whole iteration is terminated until all clusters become empty or resources are exhausted.

4 EXPERIMENT

To validate our technique, we conduct a comprehensive experiment on the industrial data. In this section, we first raise research questions of this experiment, and then we detail the dataset, baselines, and evaluation metrics. We evaluate our technique through three aspects: effectiveness, usefulness, and potential. In addition, we analyze the parameter sensitivity for helping users to apply our technique in different settings. Finally, we introduce the experiment setup, which includes the parameter settings and the hypotheses setup to answer research questions.

4.1 Research Questions

Identifying the test reports that describe the same bug or present similar topics is critical for improving the efficiency of processing the overwhelming number of crowdsourced test reports. For our technique, one of the critical steps is to group these test reports into clusters. Given the fact that clustering result fundamentally determines the effectiveness of our technique, we design the *RQ1* to investigate whether our image-understanding-based clustering technique is effective for grouping the test reports. *RQ1* is formulated as follows:

[RQ1. Effectiveness:] To what extent can cluster with image features accurately group the crowdsourced mobile test reports?

On the other hand, even though we investigate the effectiveness of our technique in clustering the test reports in the *RQ1*, it is essential for validating its usefulness for the practical test report inspection task. To understand its practical usefulness, we compare our technique with the existing state-of-the-art crowdsourced test report processing techniques. In addition, considering investigating the potential of our technique could be helpful for engineers to optimize it in the application and inform the future research in this field, we design the following two research questions:

TABLE 3
Summary of Experimental Subjects

	Name	Main Function	$ R $	$ S $	$ R_s $	$ F $
p_1	Wonderland	Travel Guidelines	191	116	93	23
p_2	Game-2048	Puzzle Game	210	174	154	12
p_3	TravelDiary	Travel Notes	240	170	142	14
p_4	HW Health	Sports & Health	262	274	201	33
p_5	HJ Normandy	English Education	269	381	241	22
p_6	MyListening	Listening Training	432	348	288	15
<i>Total</i>			1604	1463	1119	119

[RQ2.1 Usefulness:] To what extent can our approach substantially improve test-report inspection and find more unique buggy reports earlier?

[RQ2.2 Potential:] How large is the gap between our clustering method and IDEAL strategies?

Finally, because the performance of our technique is influenced by several key parameters, we analyze the performance of our approach under different parameter settings to present users an extensive understanding of our technique. Note that, as we discussed in the Section 3, we only focus on the three parameters, i.e., balanced factor β , clustering threshold ε and sampling percent ρ , which can have significant impacts on the performance of our approach, while setting $\alpha = 0.8$ and $\gamma = 0.1$ in the experiment. The RQ3 is designed as follows:

[RQ3. Parameter Sensitivity:] How does the experimental parameter influence the performance of our approach?

4.2 Data Collection

To investigate the performance of our technique under real crowdsourced testing settings, we collaborated with six industrial companies. Based on the mobile application and testing requirement provided by these companies, we host a national software-testing contest¹² to simulate the crowdsourced testing. In total, more than 100 third-year undergraduates came from 27 prestigious universities to participate in this contest. The contestants were required to test the subject applications and report bugs within four hours. They could write descriptions and/or take screenshots to document their testing procedures and the behavior of applications. More than 10 professional testers and developers from our collaborators manually labeled and evaluated the quality of these reports. The detailed information of the dataset is shown in Table 3, in which, $|R|$ denotes the number of reports, $|S|$ denotes the number of screenshots, $|R_s|$ denotes the number of reports that contain at least one screenshot, and $|F|$ denotes the number of faults revealed by the reports. Based on this figure, we can observe that 1119 out of 1604 crowdsourced test reports of mobile applications contain screenshots, and these reports detected 119 bugs.

12. http://www.mooctest.org/cst2016/index_en.html, Latest access on 1st, Mar, 2020.

4.3 RQ1. Effectiveness

4.3.1 Baselines

Because RQ1 is designed for evaluating the quality of clustering results, we adopted TXT, which clusters test reports based on only the text distance, and IMG, which clusters test reports based on only the image distance to reveal the performance of image information, and the clustering method proposed by Yang *et al.* in DMBD19 [24], as baselines. Thus, we have the following four techniques:

- *TXT&IMG*: The clustering is conducted based on both textual information and screenshots. In this method, the distance between the two reports is calculated based on the balanced distance Equation (7).
- *TXT*: The clustering is conducted based on only the text distance between test reports, which was presented as DT in Equation (4).
- *IMG*: The clustering strategy based on the screenshot distance between test reports, which was presented as DS in Equation (6).
- *DMBD19*: The test reports clustering technique using multi-source heterogeneous information proposed by Yang *et al.* in DMBD19.

4.3.2 Evaluation Metrics

We employ widely-used metrics to analyze the result of clustering, including Homogeneity, Completeness, V-measure [38], and Silhouette Coefficient [39].

To ease the explanation, we use the class set C and cluster set K to denote ground truth and clustering results. Then we define n as the total number of reports, n_c and n_k as the number of reports respectively belonging to class c and cluster k , and $n_{c,k}$ denotes the number of reports from class c assigned to cluster k . Based on these annotations, we can formulate these metrics.

Homogeneity. A clustering result satisfies homogeneity if all of its clusters contain only data points which are members of a single class. It reflects the extent to which each cluster contains only members of a single class. Homogeneity scores are formally given by:

$$h = 1 - \frac{H(C|K)}{H(C)}.$$

where

$$H(C|K) = - \sum_{c=1}^{|C|} \sum_{k=1}^{|K|} \frac{n_{c,k}}{n} \cdot \log \left(\frac{n_{c,k}}{n_k} \right)$$

$$H(C) = - \sum_{c=1}^{|C|} \frac{n_c}{n} \cdot \log \left(\frac{n_c}{n} \right).$$

Completeness. A clustering result satisfies completeness if all the data points that are members of a given class are elements of the same cluster. It measures the extent to which all members of a given class are assigned to the same cluster. Completeness scores are formally given by:

$$c = 1 - \frac{H(K|C)}{H(K)},$$

where

$$H(K|C) = - \sum_{k=1}^{|K|} \sum_{c=1}^{|C|} \frac{n_{c,k}}{n} \cdot \log \left(\frac{n_{c,k}}{n_c} \right)$$

$$H(K) = - \sum_{k=1}^{|K|} \frac{n_k}{n} \cdot \log \left(\frac{n_k}{n} \right).$$

V-measure. The V-measure is a harmonic mean between homogeneity and completeness, which is widely used as a metric to evaluate the performance of clustering. In our experiment, a higher V-measure score means better clustering performance. The V-measure score is calculated as the following formula:

$$v = 2 \cdot \frac{(h \cdot c)}{h + c}. \quad (8)$$

The value range of all three above metrics is [0,1], and they are the higher the better.

Silhouette Coefficient. Different from the metrics like H, C, and V, the Silhouette Coefficient does not require the ground truth tables. It combines the degree of cohesion and the degree of separation to evaluate the clustering results. For each report, the Silhouette Coefficient is calculated using the mean intra-cluster distance a and the mean nearest-cluster distance b . To clarify, b is the distance between a report and the nearest cluster that the report is not a part of. Then the Silhouette Coefficient is formally given by:

$$SC = \frac{b - a}{\max(a, b)}. \quad (9)$$

The value range of SC scores is [-1, 1], and when the value is closer to 1, it represents a better clustering result.

4.4 RQ2. Usefulness and Potential

4.4.1 Baselines

Note that the method in DMBD19 is designed for clustering test reports, we can only evaluate its performance based on the evaluation metrics of clustering. Thus, we have employed four techniques, i.e., TXT, IMG, TXT&IMG, DMBD19, for addressing the *RQ1*, whereas we keep TXT, IMG, TXT&IMG as baselines in the study of *RQ2*. In addition, we simulate the ideal inspection process, which can only be achieved theoretically, to investigate the usefulness and potential of our technique. Further, our prior work presented the multi-objective technique to prioritize test reports based on the distance of both text descriptions and images [23]. We also introduce this method as one of the baselines. Also, we simulate the situation that developers have no any ancillary techniques for inspecting the reports. Under that situation, they may randomly inspect test reports, i.e., in a non-systematic order. Thus, for *RQ2*, we have six techniques as follows:

- *TXT*: The sampled report clusters are derived from the results of only text-distance-based clustering.
- *IMG*: The sampled report clusters are derived from the results of only screenshot-distance-based clustering.
- *BDDiv*: A multi-objective test-report prioritization technique proposed in our prior publication (ASE'16 conference paper) [23].

- *TXT&IMG*: The sampled report clusters are derived from the results of hybrid-distance-based clustering.
- *RANDOM*: The randomly inspection strategy, which is used to simulate the situation without ancillary techniques.
- *IDEAL*: The theoretically ideal inspection orders. For test reports with M faults, all errors can be found by reviewing only M reports.

4.4.2 Evaluation Metrics

We adopted the APFD (Average Percentage of Fault Detected) [40], a widely-used evaluation metric of the classical test case prioritization, to measure the performance of our technique. For each fault, APFD marks the index of first test report which reveals it. Based on the order of test reports and fault information they revealed, we can calculate APFD scores to evaluate the usefulness of our technique for report inspection. We present the formula of computing the APFD in the following equation:

$$APFD = 1 - \frac{T_{f1} + T_{f2} + \dots + T_{fM}}{n \times M} + \frac{1}{2 \times n}. \quad (10)$$

In Equation (10), n denotes the number of test reports, M denotes the number of faults revealed by all test reports. T_{fi} is the index of the first test report that reveals fault i . In our experiment, a higher APFD value indicates a better inspection procedure. That is, the method with higher APFD value can reveal more faults earlier. In addition, we employ *Gap*, which reflects the difference of our technique compared with the *IDEAL* strategy, to evaluate the potential of our technique. In the experiment, the *Gap* between technique X and the *IDEAL* can be calculated as $G = (Best - X)/X$. This metric indicates the potential of techniques, which is helpful for researchers to design and improve crowdsourced test report processing techniques.

4.5 RQ3. Parameter Sensitivity

In *RQ3*, we analyze the impact of parameters on the performance. We focus on the following parameters: balanced factor β , which controls the weight of text distance and image distance; clustering threshold ε , which determines the stop point of hierarchical clustering; and sampling ratio ρ , which determines the number of test reports to be sampled from each cluster.

The parameter β is designed for controlling the process of clustering, thus, we employ the evaluation metrics of *RQ1*, i.e., homogeneity, completeness, and v-measures, to analyze its impact. Similarly, because of the clustering threshold ε and sampling ratio ρ influences the practical performance of our technique, we employ the evaluation metrics of *RQ2*, i.e., APFD, to analyze their impacts.

4.6 Experiment Setup

In this section, we detail the parameter settings and the hypotheses setup for research questions.

4.6.1 Parameter Settings

One of the features of crowdsourced testing is that it can provide the testing results of diverse devices. Thus, given

TABLE 4
HCV Scores of Test Report Clustering With Different Distance Metrics

	Method	<i>p</i> 1	<i>p</i> 2	<i>p</i> 3	<i>p</i> 4	<i>p</i> 5	<i>p</i> 6	<i>avg</i>
H	TXT&IMG	0.900	0.514	0.890	0.914	0.807	0.800	0.804
	TXT	0.751	0.857	0.746	0.879	0.800	0.669	0.784
	IMG	0.444	0.456	0.884	0.859	0.789	0.791	0.703
	DMBD19	0.810	0.486	0.485	0.383	0.363	0.753	0.547
C	TXT&IMG	0.694	0.440	0.537	0.643	0.546	0.390	0.542
	TXT	0.660	0.386	0.513	0.592	0.444	0.316	0.485
	IMG	0.683	0.393	0.526	0.633	0.533	0.386	0.527
	DMBD19	0.488	0.348	0.274	0.441	0.342	0.350	0.374
V	TXT&IMG	0.783	0.474	0.670	0.755	0.651	0.525	0.643
	TXT	0.702	0.532	0.608	0.707	0.572	0.429	0.592
	IMG	0.778	0.422	0.659	0.728	0.626	0.519	0.622
	DMBD19	0.609	0.405	0.350	0.409	0.351	0.477	0.434

the screenshots submitted by crowd workers are potentially of different resolutions, in our experiment, we resize all the screenshots to 480×480 pixels. Further, as we discussed in the Section 3, some fundamental parameters can influence the performance of our experiment. We set the threshold of identifying the similar screenshots $\gamma = 0.1$, the factor of scaling the weight of text similarity $\alpha = 0.8$, and the threshold of balanced factor $\beta = 1$. Also, in agglomerative hierarchical clustering, we set the threshold of determining the stop point of clustering $\varepsilon = 0.8$.

In the sampling procedure, we set the sampling ratio parameter $\rho = 5\%$. This means, we randomly sample 5 percent reports from clusters in each iteration. Especially, for the clusters that contain only one test report, we take all singular test reports in the first round. Note that, except the experiment of *RQ3* that is designed for investigating the parameter sensitivity, we did not change these settings in the whole experiment to ensure the consistency of the results.

4.6.2 Hypotheses Setup

To answer *RQ1*, we set up the following null hypothesis H_{10} and the alternative hypothesis H_{1A} for *RQ1*:

- H_{10} : The silhouette coefficient of clustering results with text and image features is not significantly higher than the baselines (TXT, IMG, DMBD19).
- H_{1A} : The silhouette coefficient of clustering results with text and image features is significantly higher than the baselines (TXT, IMG, DMBD19).

To answer questions in *RQ2*, we set up the null hypothesis H_{20} and the alternative hypothesis H_{2A} for *RQ2* as follows:

- H_{20} : The APFD scores of our technique (TXT&IMG) is not significantly higher than the baselines (TXT, IMG, BDDiv, RANDOM).
- H_{2A} : The APFD scores of our technique (TXT&IMG) is significantly higher than the baselines (TXT, IMG, BDDiv, RANDOM).

To comprehensively answer *RQ3*, we set up two null hypotheses H_{30} and H_{40} for the two primary parameters ε and ρ respectively.

- H_{30} : The minimum merging distance threshold ε of our technique (TXT&IMG) cannot significantly influence the performance.
- H_{3A} : The minimum merging distance threshold ε of our technique (TXT&IMG) can significantly influence the performance.
- H_{40} : The sampling ratio ρ of our technique (TXT&IMG) cannot significantly influence the performance.
- H_{4A} : The sampling ratio ρ of our technique (TXT&IMG) can significantly influence the performance.

5 RESULT DISCUSSION

In this section, we present the experimental results to answer the three research questions.

5.1 Answering Research Question 1: Effectiveness

[*RQ1. Effectiveness*:] To what extent can cluster with image features accurately group the crowdsourced mobile test reports?

We present the homogeneity(H), completeness(C) and V-Measure(V) results of these four techniques in Table 4. Regarding the V-Measure score, TXT&IMG, i.e., clustering based on balanced-distance, achieves 0.643 on average, while the other three baselines, i.e., TXT, IMG, and DMBD19, obtain only 0.592, 0.622, and 0.434 respectively. We can observe that TXT&IMG outperforms the TXT and IMG over five subject projects, except the *p2(Game-2048)*.

We investigate the *p2(Game-2048)* to identify the reason for this distinct result. We notice that while the TXT&IMG obtains a relatively high homogeneity score of more than 0.8 and outperforms the three baselines on *p1, p3, p4, p5, p6*, TXT reaches 0.857 when TXT&IMG achieves only 0.514 on *p2(Game-2048)*. When the completeness scores of these four techniques are close to each other, the high homogeneity score of TXT naturally leads to a high V-measure score. We further analyze the raw reports of *p2(Game-2048)*. We found that within the test reports of *p2(Game-2048)* almost all screenshots submitted by crowd workers are the activity views of the game content panel. Even though these screenshots are used to describe different bugs, they are often similar to each other because all of them are captured from the same activity view. This effect misleads the two image-involved techniques, i.e., TXT&IMG

TABLE 5
Silhouette Coefficient of Test Report Clustering with Different Distance Metrics

Method	<i>p</i> 1	<i>p</i> 2	<i>p</i> 3	<i>p</i> 4	<i>p</i> 5	<i>p</i> 6
SC scores	TXT&IMG	0.344	0.277	0.271	0.313	0.330
	TXT	0.083	0.100	0.104	0.097	0.088
	IMG	0.119	0.306	0.086	0.188	0.284
	DMBD19	0.093	0.069	0.072	0.050	0.053
p-value	TXT&IMG-TXT	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
	TXT&IMG-IMG	< 0.01	0.763	< 0.01	< 0.01	< 0.01
	TXT&IMG-DMBD19	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
effect size	TXT&IMG-TXT	0.620	0.605	0.620	0.604	0.620
	TXT&IMG-IMG	0.622	-	0.626	0.622	0.605
	TXT&IMG-DMBD19	0.619	0.621	0.621	0.620	0.620

and IMG, to group the test reports that are describing different bugs into the same cluster.

In order to evaluate the performance of our clustering technique in the absence of ground-truth tables, we further compute the Silhouette Coefficient of each project and present the results in Table 5. Similar to results in Table 4, TXT&IMG outperforms the other three techniques on all projects except the *p*2(*Game-2048*). To investigate whether TXT&IMG significantly improves the technique which only based on TXT, IMG, or DMBD19, we conduct Wilcoxon signed-rank tests between our technique (TXT&IMG) and the other three techniques. Since HAC algorithm could not specify the number of clusters during clustering, we slightly raised and lowered the current clustering threshold by 0.05 with 0.005 as a step, and obtained 20 groups of clustering results for each method to support our tests.

We present the p-value of tests and their corresponding effect size (*Cohen's d*) in Table 5. The results show that the null hypothesis H_0 will be rejected, expect the test in *p*2(*Game-2048*: TXT&IMG-IMG). On the other hand, the relatively high effect size means the difference of Silhouette Coefficient between the techniques is distinct.

Summary. The high V-measure scores and SC scores indicate that our image-understanding-based test report clustering technique is capable of improving the test reports describing similar bugs together. On the other hand, we found that the information of screenshots may negatively influence the test report clustering of these applications that contain limited number of activity views.

5.2 Answering Research Question 2

To reduce the bias that is introduced by the randomness in the iterative sampling process, we conducted the experiment 30 times and present the result in Figs. 3 and 4.

Fig. 3 shows boxplots of the APFD results for the six projects and Fig. 4 presents the average fault detection curves with the increasing number of inspected reports. In addition, we present the mean value of APFD of the 30 runs, the improvement over RANDOM, and the gap between our technique and IDEAL. Further, we conduct Wilcoxon signed-rank tests between our technique (TXT&IMG) and the other four techniques (TXT, IMG, BDDiv, RANDOM) and present the results in Table 6.

[RQ2.1. *Usefulness*]: To what extent can our technique substantially improve test-report inspection and find more unique buggy reports earlier?

Note that our prioritization technique in the conference paper [23], which is denoted as *BDDiv*, is employed as a baseline. Based on the boxplots of APFD values shown in Fig. 3 and the third column of Table 6, we observe that, to different extents, all of these clustering techniques outperform the RANDOM inspection on all projects except *p*2(*Game-2048*). Similarly, the curves in Fig. 4 show that the TXT&IMG is able to detect all faults earlier on these projects. Especially, for the BDDiv method, which also employed the information of screenshots to analyze the test reports, we observed that the TXT&IMG technique consistently outperforms it on all projects except *p*2(*Game-2048*).

Further, considering we adopted the random strategy to sample test reports from the clustering results, we repeat the experiment 30 times and conduct the Wilcoxon signed-rank tests based on the APFD scores to analyze the differences between TXT&IMG and the other four techniques. We present the test results and the average improvement over RANDOM in Table 6. Based on Table 6, for all projects except *p*2(*Game-2048*), we can observe that the improvement of TXT&IMG ranges 21.3-37.17 percent in comparison with the RANDOM, while TXT improves only 0-26.02 percent.

Given the fact that in our tests all *p*-values less than 0.01 except *p*2(*Game-2048*: TXT&IMG-IMG, TXT&IMG-RANDOM), which means for other five projects the null hypothesis H_0 will be rejected. On the other hand, their corresponding effect size (*Cohen's d*) fluctuates between 0.37 and 0.619, and we can conclude that the improvements coming from these clustering techniques are statistically significant.

Also, on all projects, we observe that the lengths of the boxplots of our clustering-sampling techniques, i.e., TXT&IMG, TXT, and IMG, are smaller than the boxplot of RANDOM. Because the box length indicates the data variability, this observation indicates the performance of these four techniques is more stable than RANDOM.

[RQ2.2. *Potential*]: How large is the gap between our clustering method and IDEAL strategies?

The fourth column of Table 6 shows the gap between our strategies and the theoretical IDEAL. Over the six subject programs, we found the gap between TXT&IMG and IDEAL varies from 17.56 to 41.42 percent while the gap between BDDiv and IDEAL ranges from 29.12 to 64.16 percent. In Fig. 4, which visualizes the growth rate of APFD value, the curves of IDEAL grow at a fast rate, and the best situation reached the top while the TXT&IMG stayed around 35 percent.

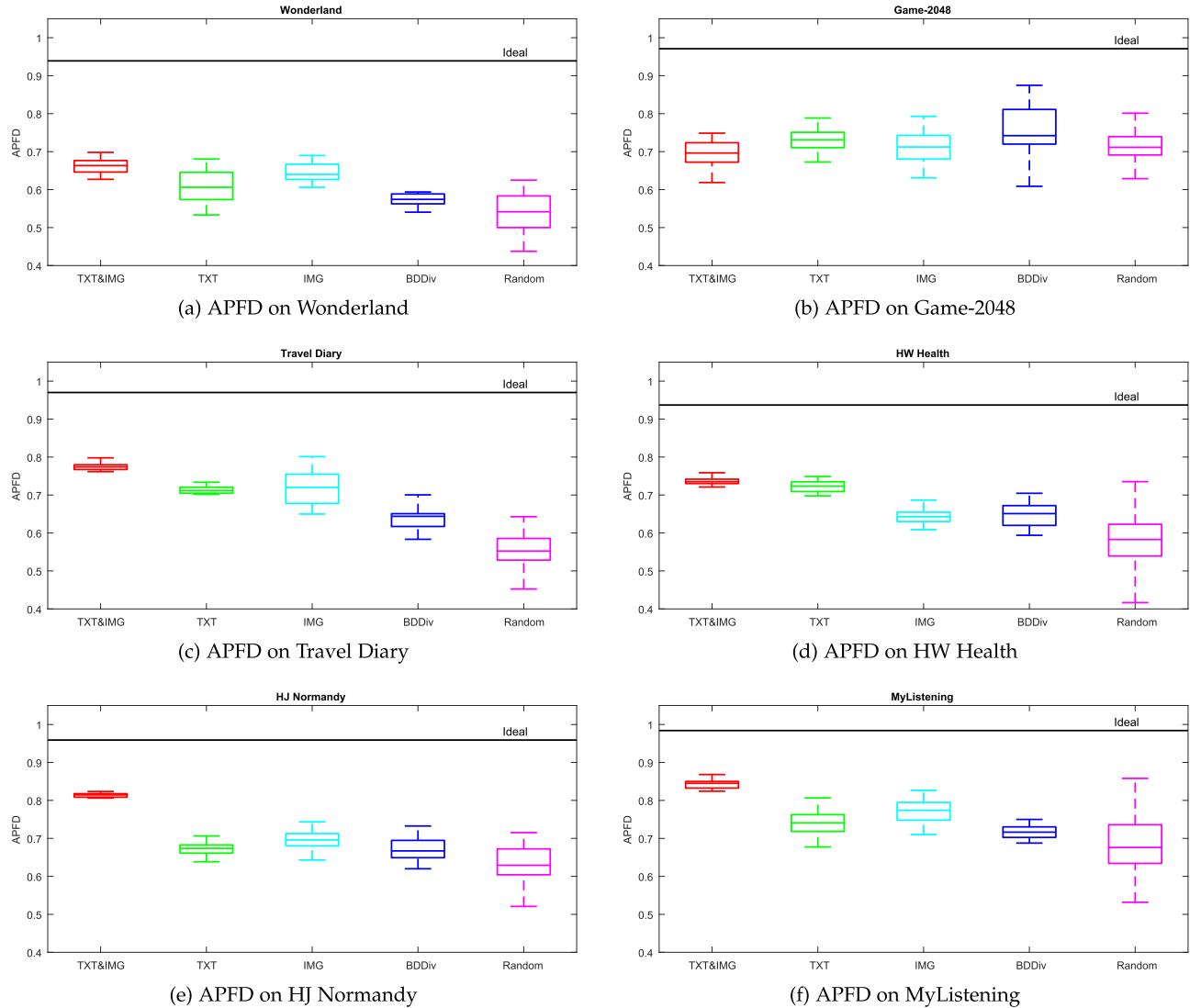


Fig. 3. APFD of experimental subjects (averaged over 30 runs).

Summary. To answer the *RQ2*, we conducted Wilcoxon signed-rank tests over the results of 30 executions. Based on the test result, we can draw the following summaries: 1. all of these clustering techniques can improve the efficiency of the test reports inspection in comparison with the RANDOM method. 2. the image-based approaches distinctly improved the performance of conventional text-based clustering techniques. 3. on the projects with ample app-specific views, clustering techniques are more appropriate for report inspection than the prioritization techniques. Compared with other strategies, the TXT&IMG shows a smaller gap for the theoretical IDEAL result. However, there is room for future work to improve the clustering-sampling techniques for test report inspection.

5.3 Answering Research Question 3

[*RQ3. Parameter Sensitivity*]: How does the experimental parameter influence the performance of our approach?

In this subsection, we further discuss the impact of parameter settings on the performance. This study is helpful for users of our approach to set proper parameters for different usage scenarios. We analyze the parameter sensitivity tests based on three key parameters: balanced factor β ,

clustering threshold ε and sampling percent ρ , which influence the three fundamental steps of the clustering-sampling process respectively:

- The parameter β controls the balance distance calculation as a harmonic weight. We analyze the clustering results when the value of β ranges from 0.5 to 1.5 with the increment of 0.1.
- The parameter ε is employed to control the stop point of hierarchical clustering. The clustering procedure terminates when the distance between the closest cluster pair is larger than the value of ε . In this study, we discuss the APFD scores when the value of ε ranges from 0.5 to 0.9 with the increment of 0.1.
- The parameter ρ controls the number of reports sampling from each cluster, it influences the efficiency of test report inspection. We analyze the trends of APFD scores when the value of ρ ranges from 5 to 30 percent with the increment of 5 percent.

Fig. 5 shows the sensitivity of clustering results to the parameter β , given the $\varepsilon = 0.8$ and $\rho = 0.1$. And we present the average value of homogeneity, completeness, and v-measure in the same setting in Table 7 . From the table we

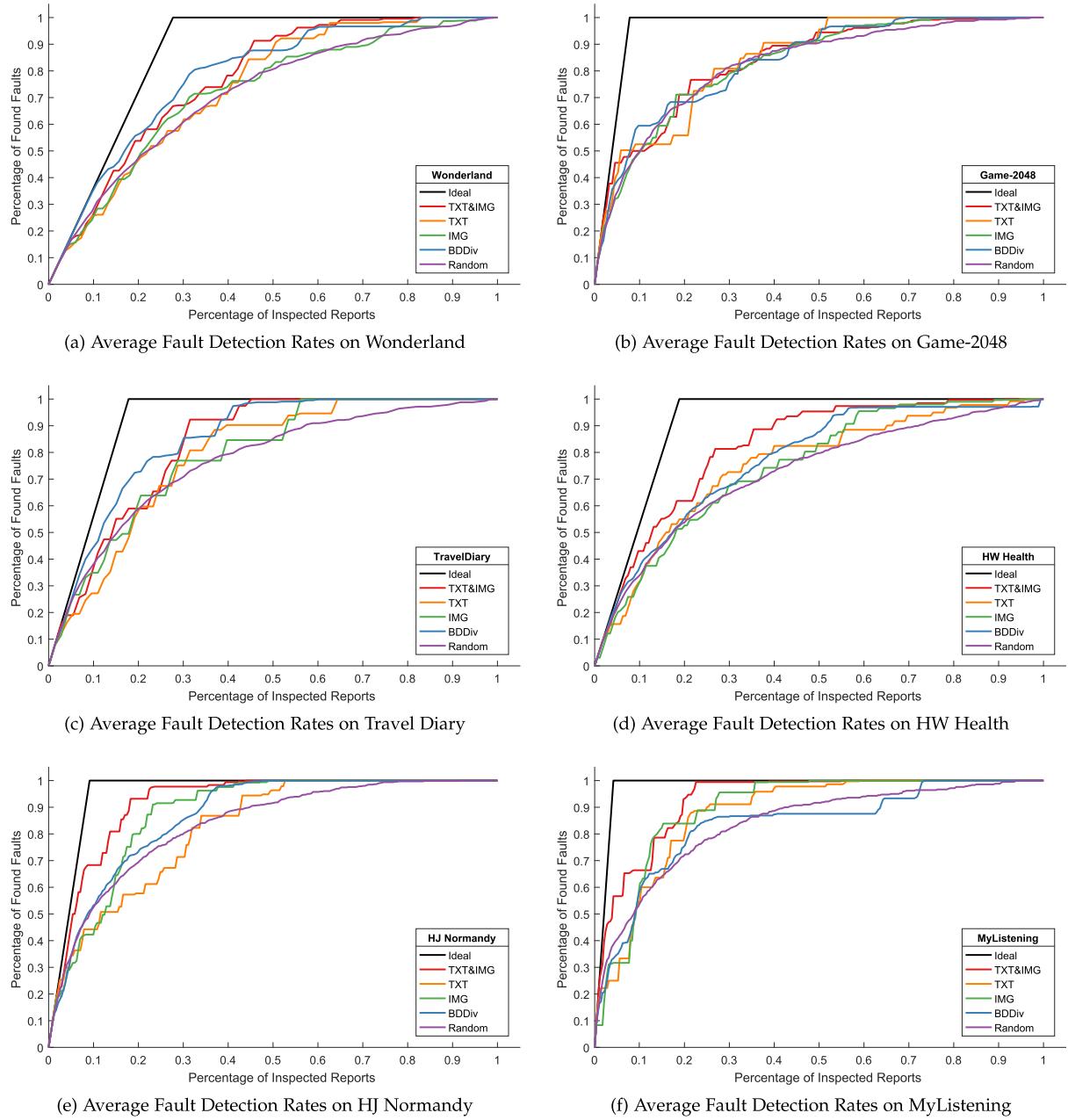


Fig. 4. Average fault detection rates on experimental subjects (averaged over 30 runs).

observe that when the value of β reaches 1.0, four projects, i.e., Wonderland, Game-2048, Travel Diary, and HJ Normandy, obtain the highest v-measure score. The other two projects, i.e., HW Health and MyListening, reach the highest v-measure score while $\beta = 0.9$, and the difference of v-measure between $\beta = 1.0$ and $\beta = 0.9$ is less than 0.01. This indicates that setting the weight of the distance of textual description as well as screenshots closely is helpful for optimizing the balanced-distance-based clustering. Also, we present the average value and corresponding standard deviation for each of these projects in the last two columns of the Table 7. We observe that the standard deviation value of homogeneity, completeness, and v-measure stays in a range of small number, i.e., from 0.005 to 0.051, which indicates that the clustering result is relatively stable to the change of β .

Similarly, Table 8 shows the average APFD score with the changes of parameter ε , given $\beta = 1.0$ and $\rho = 0.1$. From the

table we observe that APFD of five projects (p_1, p_3, p_4, p_5, p_6) reaches highest values when $\varepsilon = 0.8$. In this table we also present the average value and standard deviation. The results show that, for all the six subject programs, the standard deviation of the APFD values is marginal in comparison with the average value, which indicates the performance of our technique is stable under the setting of different ε value. Meanwhile, in order to answer the hypothesis test of RQ3, we conduct Friedman's rank tests over APFD scores of all parameter ε on each project, and present the p-value in the sixth row of Table 8. The test results show that the null hypothesis H_{30} will be rejected, the parameter ε influences the APFD scores in our technique (TXT&IMG). However, considering the change in values of APFD scores, such an influence is acceptable.

Further, the sample percent ρ influences the efficiency of test report inspection. We present the average APFD score

TABLE 6
Wilcoxon Signed Rank Tests of APFD Scores

Method	APFD Means	Improvement: $\frac{X - \text{Random}}{\text{Random}}$	Gap: $\frac{\text{Best} - X}{X}$
p1: $p - \text{value} < 0.01$, $0.370 \leq \text{effect-size} \leq 0.617$			
IDEAL	0.939	73.89%	/
TXT&IMG	0.664	22.96%	41.42%
TXT	0.607	12.41%	54.7%
IMG	0.643	19.07%	46.03%
BDDiv	0.572	5.93%	64.16%
RANDOM	0.54	/	73.89%
p2: $p - \text{value}^{***} < 0.01$, $0.471 \leq \text{effect-size} \leq 0.506$			
IDEAL	0.971	36.95%	/
TXT&IMG	0.695	-1.97%	39.71%
TXT	0.709	0%	36.95%
IMG	0.708	-0.14%	37.15%
BDDiv	0.752	6.06%	29.12%
RANDOM	0.709	/	36.95%
p3: $p - \text{value} < 0.01$, $0.546 \leq \text{effect-size} \leq 0.618$			
IDEAL	0.97	71.68%	/
TXT&IMG	0.775	37.17%	25.16%
TXT	0.712	26.02%	36.24%
IMG	0.722	27.79%	34.35%
BDDiv	0.64	13.27%	51.56%
RANDOM	0.565	/	71.68%
p4: $p - \text{value} < 0.01$, $0.434 \leq \text{effect-size} \leq 0.617$			
IDEAL	0.937	60.17%	/
TXT&IMG	0.735	25.64%	27.48%
TXT	0.722	23.42%	29.78%
IMG	0.641	9.57%	46.18%
BDDiv	0.645	10.26%	45.27%
RANDOM	0.585	/	60.17%
p5: $p - \text{value} < 0.01$, $0.617 \leq \text{effect-size} \leq 0.619$			
IDEAL	0.959	51.74%	/
TXT&IMG	0.808	27.85%	18.69%
TXT	0.672	6.33%	42.71%
IMG	0.695	9.97%	37.99%
BDDiv	0.669	5.85%	43.35%
RANDOM	0.632	/	51.74%
p6: $p - \text{value} < 0.01$, $0.609 \leq \text{effect-size} \leq 0.617$			
IDEAL	0.984	42.61%	/
TXT&IMG	0.837	21.3%	17.56%
TXT	0.74	7.25%	32.97%
IMG	0.77	11.59%	27.79%
BDDiv	0.718	4.06%	37.05%
RANDOM	0.69	/	42.61%

***TXT&IMG-IMG: $p - \text{value} = 0.245$

***TXT&IMG-RANDOM: $p - \text{value} = 0.111$

with the changes of parameter ρ in Table 9 and Fig. 6, given $\beta = 1.0$ and $\varepsilon = 0.8$. In Table 9, we observe that the APFD scores of these six subject programs reach the highest value

under different ρ value. And the p-value in the sixth row also rejects the null hypothesis H_{40} , which proves the parameter ρ will influence our technique (TXT&IMG). However, the standard deviation of the APFD value presented in Table 9 varies in 0.004~0.012, and the curves shown in Fig. 6 are relatively smooth. This fact proves that our technique is relatively stable under the different settings of sample percent ρ .

Summary. While all three parameters influence the performance of our technique to a different extent, the performance of our technique is generally stable to their changes. Because the experiment result indicates setting the weight of textual description and screenshots equally can make our technique perform well, we suggest the users of our technique adjust to β starts from 1. Similarly, we suggest the users of our technique to set the default value of ε into 0.8 and set the default value of ρ into 0.1.

6 THREATS TO VALIDITY

Subject Program Selection. Although crowdsourced testing has covered a wide range of mobile platforms (e.g., Android, IOS, WP), the limitations of data sources have allowed us to experiment with only six Android applications. We cannot guarantee the similar good results could generalize beyond the platforms. Nevertheless, this risk could be reduced because our subject applications vary different categories that diversifies the functionalities including health assistant, entertainment, travel assistant, diary editor, and language learning tools. Thus, we believe these applications can indicate the effectiveness and applicability of our methods.

Natural Language Selection. In this experiment, all the crowdsourced test reports are written in Chinese, which implies the similar results may not be observed based on the test reports written in other languages. However, the natural language processing technique is not the focus of our research. Instead, it works as the ancillary technique to generate the intermediate outputs of our techniques. Even though our technique involves natural language processing, this part focuses on building the keyword vector models to compute the text distance between reports. To build keyword vector models from different languages, many sophisticated methods and NLP tools are available, such as the

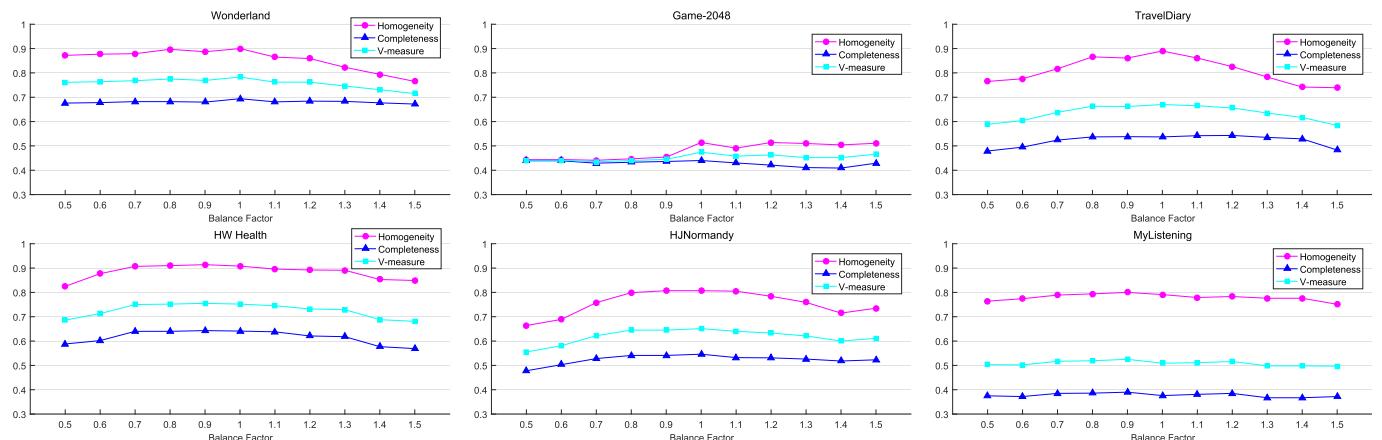


Fig. 5. The sensitivity of clustering results to the parameter β ($\varepsilon = 0.8$ and $\rho = 0.1$).

TABLE 7
The Comparison of Clustering Results Over 30 Executions Under Different Settings of β ($\varepsilon = 0.8$ and $\rho = 0.1$)

Project	Metric	Balance Factor β										<i>avg</i>	<i>std</i>	
		0.5	0.6	0.7	0.8	0.9	1	1.1	1.2	1.3	1.4			
Wonderland	H	0.872	0.877	0.879	0.897	0.887	0.9	0.865	0.859	0.823	0.794	0.765	0.856	0.042
	C	0.676	0.678	0.682	0.682	0.68	0.694	0.681	0.684	0.683	0.677	0.672	0.681	0.005
	V	0.761	0.764	0.768	0.775	0.769	0.783	0.762	0.762	0.746	0.731	0.715	0.758	0.019
Game-2048	H	0.444	0.444	0.441	0.447	0.454	0.514	0.491	0.514	0.51	0.504	0.511	0.479	0.031
	C	0.439	0.439	0.429	0.433	0.436	0.44	0.43	0.421	0.411	0.409	0.429	0.429	0.01
	V	0.441	0.441	0.435	0.440	0.445	0.474	0.458	0.463	0.452	0.452	0.466	0.452	0.012
Travel Diary	H	0.765	0.776	0.817	0.866	0.861	0.89	0.861	0.826	0.783	0.742	0.739	0.811	0.051
	C	0.479	0.495	0.524	0.537	0.538	0.537	0.542	0.543	0.535	0.529	0.483	0.522	0.023
	V	0.589	0.604	0.638	0.663	0.662	0.67	0.665	0.656	0.635	0.617	0.584	0.635	0.03
HW Health	H	0.825	0.877	0.907	0.91	0.914	0.908	0.896	0.892	0.891	0.853	0.849	0.884	0.028
	C	0.588	0.602	0.64	0.64	0.643	0.641	0.638	0.621	0.618	0.577	0.569	0.616	0.027
	V	0.686	0.713	0.75	0.752	0.755	0.752	0.745	0.732	0.729	0.688	0.681	0.726	0.028
HJ Normandy	H	0.664	0.689	0.758	0.799	0.807	0.807	0.805	0.784	0.759	0.715	0.735	0.757	0.048
	C	0.478	0.503	0.528	0.541	0.541	0.546	0.532	0.531	0.526	0.518	0.523	0.524	0.019
	V	0.555	0.581	0.622	0.645	0.645	0.651	0.64	0.633	0.621	0.6	0.611	0.619	0.029
MyListening	H	0.764	0.774	0.789	0.794	0.8	0.791	0.779	0.783	0.776	0.776	0.751	0.78	0.013
	C	0.375	0.372	0.385	0.386	0.39	0.376	0.381	0.385	0.367	0.367	0.372	0.378	0.008
	V	0.503	0.502	0.517	0.519	0.525	0.509	0.511	0.516	0.498	0.498	0.497	0.509	0.009

CoreNLP, WordNet, NLTK. This fact illustrates the transplantable potential of our technique.

Crowd Workers. To collect the experimental data and validate our technique, we collaborated with several mobile application development companies and hosted a national contest. In this contest, students play the role of crowd workers. This compromising choice means that the population of our crowd workers may be less diverse than the population from the general populace. In theory, crowdsourcing techniques require workers to come from a large workforce pool. In this pool, individuals often have no relationship with each others [1]. Thus, this requirement implies that our result may be different if the crowd workers were from the internet with open calls.

However, according to the study of Salman *et al.* [41], if a technique or task is new to both students and professionals, similar performance can be expected to be observed. In our experiment, we control that all crowd workers have no experience in developing or using these subject applications. All testing tasks are new to these crowd workers. Thus, We believe this threat may not be a critical problem for our validation procedure.

TABLE 8
The Comparison of APFD Mean Value Over 30 Executions
Under Different Settings of ε ($\beta = 1.0$ and $\rho = 0.1$)

ε	p_1	p_2	p_3	p_4	p_5	p_6
0.5	0.626	0.634	0.684	0.677	0.781	0.808
0.6	0.642	0.675	0.696	0.701	0.802	0.806
0.7	0.649	0.711	0.686	0.681	0.779	0.820
0.8	0.659	0.695	0.769	0.735	0.808	0.831
0.9	0.655	0.701	0.724	0.721	0.661	0.809
$p - value$	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
<i>avg</i>	0.646	0.683	0.712	0.703	0.766	0.815
<i>std</i>	0.012	0.027	0.032	0.022	0.054	0.009

7 RELATED WORK

7.1 Crowdsourced Software Testing

Mao *et al.* provide a comprehensive survey on the crowdsourced software engineering [1], in which, they defined the crowdsourced software engineering as “*the act of undertaking any external software engineering tasks by an undefined, potentially large group of online workers in an open call format.*” Crowd-sourced software testing has become a reasonably popular research topic in the software engineering research community. In the industry, crowdsourcing technique has been widely used in testing areas such as QoE testing, usability testing, GUI testing and performance testing [42], [43], [44], [45], [46], [47], [48], [49]. Chen *et al.* [42] developed a crowdsourcing platform for QoE assessment in network and multimedia studies(Quadrant of Euphoria), which features low cost and participant diversity. Liu *et al.* [44] discussed both methodological differences and empirical contrasts between crowdsourced usability testing and traditional face-to-face usability testing. To prove the feasibility of crowdsourced GUI testing, Komarov *et al.* [46] conducted an experiment both in a lab

TABLE 9
The Comparison of APFD Mean Value Over 30 Executions
Under Different Settings of ρ ($\beta = 1.0$ and $\varepsilon = 0.8$)

ρ	p_1	p_2	p_3	p_4	p_5	p_6
0.05	0.652	0.704	0.739	0.729	0.793	0.828
0.1	0.659	0.695	0.769	0.735	0.808	0.831
0.15	0.663	0.711	0.769	0.732	0.796	0.831
0.2	0.664	0.687	0.771	0.728	0.806	0.837
0.25	0.659	0.719	0.771	0.735	0.799	0.825
0.3	0.659	0.691	0.774	0.722	0.801	0.824
$p - value$	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01	< 0.01
<i>avg</i>	0.659	0.701	0.766	0.73	0.8	0.829
<i>std</i>	0.004	0.011	0.012	0.004	0.005	0.004

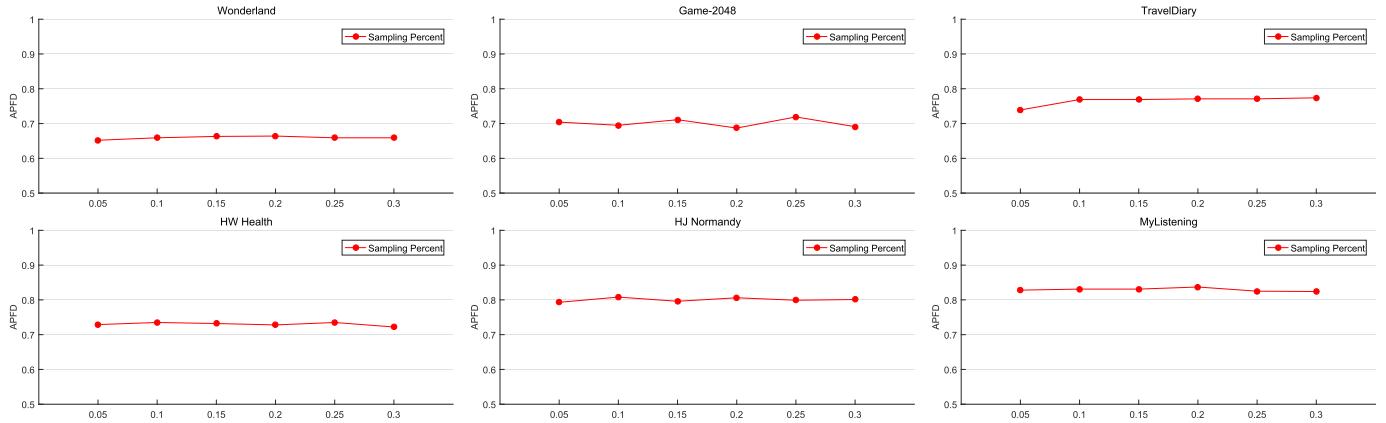


Fig. 6. The sensitivity of APFD to the parameter ρ ($\beta = 1.0$ and $\varepsilon = 0.8$).

and online with participants recruited via MTurk, and the analysis of results did not yield any evidence of significant or substantial differences. Musson *et al.* [48] helped software development teams identify and prioritize application performance issues by collecting performance data in key usage scenarios from users.

Test case generation is an indispensable part of the software testing process. By analyzing and directing the drive-by commit phenomenon on GitHub, Pham *et al.* [50] used crowdsourcing technique to recruit capable users to complete valuable test cases and maintenance tasks, giving core developers more resources to work on the more complicated issues. In program debugging, the crowd provides better solutions and more comprehensive explanations for the program issues. Chen *et al.* [51] leverage the vast mass of crowd knowledge to help developers debug their code, and even performs better than the popular static analysis tools. Crowdsourcing is particularly useful for evaluating complex and variable software systems. Sherief *et al.*'s [52] research proves that rapid feedback from crowd workers can enrich and maintain developers' timely awareness of software systems.

There are other studies on optimizing the process of crowdsourcing testing. Recruiting quality workers and effective management is the premise of carrying out crowdsourcing test tasks. Mäntylä *et al.*'s research presents that, proper time pressure and reasonable crowd worker size can achieve higher defect detection effectiveness [52]. For the decomposition of testing tasks, Tung *et al.* [53] defined the collaborative testing problem in a crowded environment as an NP-Complete job assignment problem, and solve it as an integer linear programming problem. Some researchers have discussed opportunities and challenges for crowdsourcing testing, including management of crowdsourced workers, test processes, and test techniques [54], [55], [56], [57].

All the studies mentioned above used crowdsourcing technique to solve problems in traditional software testing activities. However, in this paper, we focus on processing the overwhelming number of test reports, which is a new yet critical research topic in crowdsourced mobile software testing.

7.2 Bug Report Processing

In the development of modern software applications, rapid version updates generate a large number of bug reports. In practice, manually triage these bug reports becomes a labor-intensive and costly task for developers. To assist triagers in

improving the bug triaging efficiency, many researchers have focused on automating bug-report triaging, including bug-report prioritization, checking duplication of bug reports, and assigning bug reports to proper fixers (i.e., bug assignment) [58].

- 1) **Bug-report prioritization:** Yu *et al.* [59] employed neural network techniques to predict the priorities of bug reports, they also accelerate the training phase by reusing datasets from similar software systems. Tian *et al.* [60] proposed a machine learning framework to predict the priority levels of bug reports, their classification engine considered multiple features including temporal, textual, author, related-report, severity, and product. Kanwal *et al.* [61] developed a recommender based on Naive Bayes and SVM classifiers to automatically prioritize the new bug reports. Feng *et al.* [17] presented a prioritization strategy that combines both the risk assessment and the diversity strategy for crowdsourced test reports. This technique measured the similarity between crowdsourced test reports by leveraging natural language processing techniques. Similarly, Alenezi *et al.* [62] used different machine learning algorithms, namely Naive Bayes, Decision Trees, and Random Forest to predict the priority of bug reports. Their research also investigated the influence of different data sets on classification accuracy.
- 2) **Duplicate Report Detection:** Hiew *et al.* [63] first attempt to detect duplicate bug reports based on textual features. By transforming the textual content in bug reports into word vectors, they could calculate the similarity between reports, then ranks candidate reports to a given bug. Besides using natural language processing techniques, Jalbert *et al.* [64] proposed a system that used surface features, textual semantics, and graph clustering to identify duplicate status. Wang *et al.* [13] employed heuristics to combine the natural language information and execution information to detect duplicate bug reports.
- 3) **Bug Assignment:** Murphy *et al.* and Anvik *et al.* used machine learning techniques such as Naive Bayes and SVM to recommend bug fixers [65], [66]. Alenezi *et al.* [67] investigated the use of five-term selection methods on the accuracy of bug assignment. They

also re-balanced the load between developers based on their experience. Tamrawi *et al.* [68] used a fuzzy set to represent the developers who are capable/competent in fixing the bugs relevant to each term. Xia *et al.* [69] proposed a method for developer recommendation problem, which performed two kinds of analysis: bug reports based analysis, and developer based analysis. This method could associate the developers with the bug reports they resolved before, and measured the distance between them.

- 4) *Bug Report Clustering:* As an unsupervised-learning method, clustering techniques do not depend on the labeled data, and have been extensively employed to assist in various software engineering tasks. Jalbert *et al.* [64] used surface features, textual semantics, and graph clustering to predict duplicate status for the bug tracking system. Besides duplicate detection, their technique is also able to rank the existing reports that are more similar to the new one. Mani *et al.* [70] presented a noise reduction approach for unsupervised bug report summarization, and compared the experimental results of several unsupervised techniques including clustering technique. By using bug estimation and clustering, Nagwani *et al.* [71] proposed a data mining model to predict software bug complexity, which helps the development team to plan future software build and releases. In addition to the bug reports submitted by the testers, the crash reports reported by the operating system are equally important. To facilitate efficient handling of crashed reports, Dang *et al.* [15] proposed ReBucket, a method for clustering crash reports based on call stack matching using the Position Dependent Model (PDM). Jiang *et al.* [72] first adopted fuzzy clustering to aggregate multi-redundant test reports to reduce the inspecting time for crowdsourced test reports.

However, all of these studies focus on either text descriptions or execution traces. For crowdsourced mobile software testing, where text description is often insufficient, and execution traces are difficult to access, it is difficult to apply these techniques. The technique presented in this paper leverages the rich screenshots and short text descriptions to group crowdsourced mobile test reports. And we have conducted an experiment based on the industrial datasets to demonstrate its effectiveness in improving the efficiency of dealing with the crowdsourced mobile test reports.

7.3 Image Understanding in Software Engineering

Compared with natural language processing techniques, which have been widely used to assist textual software engineering tasks, image analysis and understanding techniques are rarely studied in the software engineering domain. In our investigation, the existing image understanding techniques mainly focus on web applications, which have clear and significant GUI structures. To detect web content structure based on visual representation, Cai *et al.* [73] proposed the VIPS algorithm, an automatic top-down, tag-tree independent approach. Choudhary *et al.* [74] proposed a testing tool called X-PERT to identify cross-browser inconsistencies in web applications automatically. Michail *et al.* [75] proposed a static approach, GUI search, to guide browsing and search of its

source code by using the GUI of applications. Such an approach would be helpful for software maintenance and reuse, particularly when the application source is unfamiliar. Liu *et al.* [76] proposed a novel technique to assist developers in understanding test reports by automatically describing the screenshots. On the other hand, to help users avoid bugs in GUI applications, Michail and Xie [77] designed a stabilizer prototype to visually describe application state at a very high level of abstraction through before/after screenshots.

In our prior work [23], we proposed the first technique to leverage image features to process test reports. It presented a multi-objective search method to prioritize the crowdsourced mobile test reports based on both the text distance and screenshot distance. Wang *et al.* [78] extracted four types of features to characterize the screenshots and the textual descriptions to detect the duplicate reports. They measured the similarity between reports based on both the structural and RBG color features. Another work presented by Yang *et al.* [24] proposed a clustering technique to group the test reports based on multi-source heterogeneous information and reported the F-measure score to evaluate their clustering technique. Yang *et al.*'s work is the closest research to this paper. Both works provide an image-understanding-based clustering technique for test reports. However, in this paper, we provide metrics for measuring the image distance between reports, as well as sampling strategies for processing the clustered reports. Further, we conduct a comprehensive analysis on the impact of the parameters to help end-users reach proper settings in the application scenarios.

8 CONCLUSION

In this paper, we proposed a novel clustering technique to alleviate the challenge of inspecting the overwhelming number of reports in crowdsourced software testing. In our preliminary investigation, mobile crowdsourced test reports usually contain shorter text descriptions and abundant screenshots. This fact motivates us to utilized image-understanding techniques to assist the traditional text-based techniques, and we proposed approaches for clustering test reports based on a hybrid information source. To the best of our knowledge, this is the first work to propose using image-understanding techniques to improve the accuracy and efficiency in test report clustering. We present the experimental results on six real industrial mobile crowdsourced projects, and evaluate the results from the standpoints of effectiveness, usefulness, and potential. We found that clustering-sampling technique, in almost all cases, is advantageous as compared to test report inspection with an orderless strategy. We also found that for most applications we studied, the practical usefulness which adopts the image-understanding technique are more promising, even if there is a minor class of applications may not be as applicable. As such, in future work, we will improve our technique to help to cluster for these classes of applications, and narrow the gap between our technique and the hypothetical ideal strategy.

ACKNOWLEDGMENTS

This work was partially supported by National Key R&D Program of China(2018YFB1403400), National Natural Science Foundation of China(61772263, 61772014), Collaborative

Innovation Center of Novel Software Technology and Industrialization, Suzhou Technology Development Plan (key industry technology innovation-prospective application research project SYG201807), and the Priority Academic Program Development of Jiangsu Higher Education Institutions.

REFERENCES

- [1] K. Mao, L. Capra, M. Harman, and Y. Jia, "A survey of the use of crowdsourcing in software engineering," *J. Syst. Softw.*, vol. 126, pp. 57–84, 2017.
- [2] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 499–510.
- [3] A. Sureka and P. Jalote, "Detecting duplicate bug report using character n-gram-based features," in *Proc. 17th Asia Pacific Softw. Eng. Conf.*, 2010, pp. 366–374.
- [4] M. Gegick, P. Rotella, and T. Xie, "Identifying security bug reports via text mining: An industrial case study," in *Proc. 7th IEEE Working Conf. Mining Softw. Repositories*, 2010, pp. 11–20.
- [5] N. Pinglasai, H. Hata, and K.-I. Matsumoto, "Classifying bug reports to bugs and other requests using topic modeling," in *Proc. 20th Asia-Pacific Softw. Eng. Conf.*, 2013, pp. 13–18.
- [6] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proc. 27th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2012, pp. 70–79.
- [7] J. Deshmukh, K. M. Annervaz, S. Podder, S. Sengupta, and N. Dubash, "Towards accurate duplicate bug retrieval using deep learning techniques," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2017, pp. 115–124.
- [8] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2011, pp. 253–262.
- [9] A. Lazar, S. Ritchey, and B. Sharif, "Improving the accuracy of duplicate bug report detection using textual similarity measures," in *Proc. 11th Working Conf. Mining Softw. Repositories*, 2014, pp. 308–311.
- [10] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 45–54.
- [11] Y. Tian, C. Sun, and D. Lo, "Improved duplicate bug report identification," in *Proc. 16th Eur. Conf. Softw. Maintenance Reeng.*, 2012, pp. 385–390.
- [12] A. Hindle, A. Alipour, and E. Stroulia, "A contextual approach towards more accurate duplicate bug report detection and ranking," *Empir. Softw. Eng.*, vol. 21, no. 2, pp. 368–410, 2016.
- [13] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proc. ACM/IEEE 30th Int. Conf. Softw. Eng.*, 2008, pp. 461–470.
- [14] C. Liu, X. Zhang, and J. Han, "A systematic study of failure proximity," *IEEE Trans. Softw. Eng.*, vol. 34, no. 6, pp. 826–843, Nov./Dec. 2008.
- [15] Y. Dang, R. Wu, H. Zhang, D. Zhang, and P. Nobel, "Rebucket: A method for clustering duplicate crash reports based on call stack similarity," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 1084–1093.
- [16] Y. Feng and Z. Chen, "Multi-label software behavior learning," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 1305–1308.
- [17] Y. Feng, Z. Chen, J. A. Jones, C. Fang, and B. Xu, "Test report prioritization to assist crowdsourced testing," in *Proc. 10th Joint Meeting Foundations Softw. Eng.*, 2015, pp. 225–236.
- [18] X. Xia, Y. Feng, D. Lo, Z. Chen, and X. Wang, "Towards more accurate multi-label software behavior learning," in *Proc. Softw. Evol. Week-IEEE Conf. Softw. Maintenance Reengineering Reverse Eng.*, 2014, pp. 134–143.
- [19] C. Fang, Z. Chen, K. Wu, and Z. Zhao, "Similarity-based test case prioritization using ordered sequences of program entities," *Softw. Quality J.*, vol. 22, no. 2, pp. 335–361, 2014.
- [20] A. Podgurski et al., "Automated support for classifying software failure reports," in *Proc. 25th Int. Conf. Softw. Eng.*, 2003, pp. 465–475.
- [21] X. Fan, X. Xie, W.-Y. Ma, H.-J. Zhang, and H.-Q. Zhou, "Visual attention based image browsing on mobile devices," in *Proc. Int. Conf. Multimedia Expo*, 2003, pp. I-53.
- [22] D. Tao, L. Jin, W. Liu, and X. Li, "Hessian regularized support vector machines for mobile image annotation on the cloud," *IEEE Trans. Multimedia*, vol. 15, no. 4, pp. 833–844, Jun. 2013.
- [23] Y. Feng, J. A. Jones, Z. Chen, and C. Fang, "Multi-objective test report prioritization using image understanding," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 202–213.
- [24] Y. Yang, X. Yao, and D. Gong, "Clustering study of crowdsourced test report with multi-source heterogeneous information," in *Proc. Int. Conf. Data Mining Big Data*, 2019, pp. 135–145.
- [25] S. Lazebnik, C. Schmid, and J. Ponce, "Beyond bags of features: Spatial pyramid matching for recognizing natural scene categories," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, 2006, pp. 2169–2178.
- [26] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Trans. Softw. Eng.*, vol. 27, no. 10, pp. 929–948, Oct. 2001.
- [27] T. Zhang, J. Chen, X. Luo, and T. Li, "Bug reports for desktop software and mobile apps in github: What is the difference?" *IEEE Softw.*, vol. 36, no. 1, pp. 63–71, Jan./Feb. 2019.
- [28] J. Zhang and D. Wang, "Duplicate report detection in urban crowdsensing applications for smart city," in *Proc. IEEE Int. Conf. Smart City/SocialCom/SustainCom*, 2015, pp. 101–107.
- [29] S. Afshan, P. McMinn, and M. Stevenson, "Evolving readable string test inputs using a natural language model to reduce human oracle cost," in *Proc. IEEE 6th Int. Conf. Softw. Testing Verification Validation*, 2013, pp. 352–361.
- [30] M. Ilieva and O. Ormandjeva, "Automatic transition of natural language software requirements specification into formal presentation," in *Proc. Int. Conf. Appl. Natural Lang. Inf. Syst.*, 2005, pp. 392–397.
- [31] E. Shutova, L. Sun, and A. Korhonen, "Metaphor identification using verb and noun clustering," in *Proc. 23rd Int. Conf. Comput. Linguistics*, 2010, pp. 1002–1010.
- [32] M. T. Diab and P. Bhutada, "Verb noun construction MWE token supervised classification," in *Proc. Workshop Multiword Expressions: Identification Interpretation Disambiguation Appl.*, 2009, pp. 17–22.
- [33] E. Nowak, F. Jurie, and B. Triggs, "Sampling strategies for bag-of-features image classification," in *Proc. Eur. Conf. Comput. Vis.*, 2006, pp. 490–503.
- [34] B. Schiele and J. L. Crowley, "Recognition without correspondence using multidimensional receptive field histograms," *Int. J. Comput. Vis.*, vol. 36, no. 1, pp. 31–50, 2000.
- [35] Y. Rubner, C. Tomasi, and L. J. Guibas, "The earth mover's distance as a metric for image retrieval," *Int. J. Comput. Vis.*, vol. 40, no. 2, pp. 99–121, 2000.
- [36] F. Murtagh, "A survey of recent advances in hierarchical clustering algorithms," *Comput. J.*, vol. 26, no. 4, pp. 354–359, 1983.
- [37] O. Yim and K. T. Ramdeen, "Hierarchical cluster analysis: Comparison of three linkage measures and application to psychological data," *Quantitative Methods Psychol.*, vol. 11, no. 1, pp. 8–21, 2015.
- [38] A. Rosenberg and J. Hirschberg, "V-measure: A conditional entropy-based external cluster evaluation measure," in *Proc. Joint Conf. Empir. Methods Natural Lang. Process. Comput. Natural Lang. Learn.*, 2007, pp. 410–420.
- [39] P. J. Rousseeuw, "Silhouettes: A graphical aid to the interpretation and validation of cluster analysis," *J. Comput. Appl. Math.*, vol. 20, pp. 53–65, 1987.
- [40] P. R. Srivastava, "Test case prioritization," *J. Theor. Appl. Inf. Technol.*, vol. 4, no. 3, pp. 178–181, 2008.
- [41] I. Salman, A. T. Misirli, and N. Juristo, "Are students representatives of professionals in software engineering experiments?" in *Proc. IEEE/ACM 37th Int. Conf. Softw. Eng.*, 2015, pp. 666–676.
- [42] K.-T. Chen, C.-J. Chang, C.-C. Wu, Y.-C. Chang, and C.-L. Lei, "Quadrant of euphoria: A crowdsourcing platform for QoE assessment," *IEEE Netw.*, vol. 24, no. 2, pp. 28–35, Mar./Apr. 2010.
- [43] C.-C. Wu, K.-T. Chen, Y.-C. Chang, and C.-L. Lei, "Crowdsourcing multimedia QoE evaluation: A trusted framework," *IEEE Trans. Multimedia*, vol. 15, no. 5, pp. 1121–1137, Aug. 2013.
- [44] D. Liu, R. G. Bias, M. Lease, and R. Kuipers, "Crowdsourcing for usability testing," *Proc. Assoc. Inf. Sci. Technol.*, vol. 49, no. 1, pp. 1–10, 2012.
- [45] C. Schneider and T. Cheung, "The power of the crowd: Performing usability testing using an on-demand workforce," in *Information Systems Development*. Berlin, Germany: Springer, 2013, pp. 551–560.
- [46] S. Komarov, K. Reinecke, and K. Z. Gajos, "Crowdsourcing performance evaluations of user interfaces," in *Proc. SIGCHI Conf. Human Factors Comput. Syst.*, 2013, pp. 207–216.
- [47] E. Dolstra, R. Vliegendhart, and J. Pouwelse, "Crowdsourcing GUI tests," in *Proc. IEEE 6th Int. Conf. Softw. Testing Verification Validation*, 2013, pp. 332–341.

- [48] R. Musson, J. Richards, D. Fisher, C. Bird, B. Bussone, and S. Ganguly, "Leveraging the crowd: How 48,000 users helped improve lync performance," *IEEE Softw.*, vol. 30, no. 4, pp. 38–45, Jul./Aug. 2013.
- [49] Y. Feng, Q. Liu, M. Dou, J. Liu, and Z. Chen, "Mubug: A mobile service for rapid bug tracking," *Sci. China Inf. Sci.*, vol. 59, no. 1, pp. 1–5, 2016.
- [50] R. Pham, L. Singer, and K. Schneider, "Building test suites in social coding sites by leveraging drive-by commits," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 1209–1212.
- [51] F. Chen and S. Kim, "Crowd debugging," in *Proc. 10th Joint Meeting Foundations Softw. Eng.*, 2015, pp. 320–332.
- [52] N. Sherief, N. Jiang, M. Hosseini, K. Phalp, and R. Ali, "Crowdsourcing software evaluation," in *Proc. 18th Int. Conf. Eval. Assessment Softw. Eng.*, 2013, Art. no. 19.
- [53] Y.-H. Tung and S.-S. Tseng, "A novel approach to collaborative testing in a crowdsourcing environment," *J. Syst. Softw.*, vol. 86, no. 8, pp. 2143–2153, 2013.
- [54] R. Gao, Y. Wang, Y. Feng, Z. Chen, and W. E. Wong, "Successes, challenges, and rethinking—an industrial investigation on crowdsourced mobile application testing," *Emp. Softw. Eng.*, vol. 24, no. 2, pp. 537–561, 2019.
- [55] Y. Feng, Y. Wang, C. Fang, N. Guo, and Z. Chen, "An approach for developing a highly trustworthy crowd-sourced workforce," *Scientia Sinica Informationis*, vol. 49, no. 11, pp. 1412–1427, 2019.
- [56] S. Zogaj, U. Bretschneider, and J. M. Leimeister, "Managing crowdsourced software testing: A case study based insight on the challenges of a crowdsourcing intermediary," *J. Bus. Econ.*, vol. 84, no. 3, pp. 375–405, 2014.
- [57] F. Guaiani and H. Muccini, "Crowd and laboratory testing can they co-exist?: An exploratory study," in *Proc. 2nd Int. Workshop CrowdSourcing Softw. Eng.*, 2015, pp. 32–37.
- [58] J. Zhang, X. Y. Wang, D. Hao, B. Xie, L. Zhang, and H. Mei, "A survey on bug-report analysis," *Sci. China Inf. Sci.*, vol. 58, no. 2, pp. 1–24, 2015.
- [59] L. Yu, W.-T. Tsai, W. Zhao, and F. Wu, "Predicting defect priority based on neural networks," in *Proc. Int. Conf. Advanced Data Mining Appl.*, 2010, pp. 356–367.
- [60] Y. Tian, D. Lo, and C. Sun, "Drone: Predicting priority of reported bugs by multi-factor analysis," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2013, pp. 200–209.
- [61] J. Kanwal and O. Maqbool, "Bug prioritization to facilitate bug report triage," *J. Comput. Sci. Technol.*, vol. 27, no. 2, pp. 397–412, 2012.
- [62] M. Alenezi and S. Banitaan, "Bug reports prioritization: Which features and classifier to use?" in *Proc. 12th Int. Conf. Mach. Learn. Appl.*, 2013, pp. 112–116.
- [63] L. Hiew, "Assisted detection of duplicate bug reports," Ph.D. dissertation, Dept. Comput. Sci., Univ. British Columbia, Vancouver, BC, 2006.
- [64] N. Jalbert and W. Weimer, "Automated duplicate detection for bug tracking systems," in *Proc. IEEE Int. Conf. Dependable Syst. Netw. With FTCS, DCC*, 2008, pp. 52–61.
- [65] G. Murphy and D. Cubranic, "Automatic bug triage using text categorization," in *Proc. 16th Int. Conf. Softw. Eng. Knowl. Eng.*, 2004, pp. 92–97.
- [66] J. Anvik, "Automating bug report assignment," in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 937–940.
- [67] M. Alenezi, K. Magel, and S. Banitaan, "Efficient bug triaging using text mining," *J. Softw.*, vol. 8, no. 9, pp. 2185–2190, 2013.
- [68] A. Tamrawi, T. T. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Fuzzy set-based automatic bug triaging (tier track)," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 884–887.
- [69] X. Xia, D. Lo, X. Wang, and B. Zhou, "Accurate developer recommendation for bug resolution," in *Proc. 20th Working Conf. Reverse Eng.*, 2013, pp. 72–81.
- [70] S. Mani, R. Catherine, V. S. Sinha, and A. Dubey, "Ausum: Approach for unsupervised bug report summarization," in *Proc. ACM SIGSOFT 20th Int. Symp. Foundations Softw. Eng.*, 2012, Art. no. 11.
- [71] N. K. Nagwani and A. Bhansali, "A data mining model to predict software bug complexity using bug estimation and clustering," in *Proc. Int. Conf. Recent Trends Inf. Telecommun. Comput.*, 2010, pp. 13–17.
- [72] H. Jiang, X. Chen, T. He, Z. Chen, and X. Li, "Fuzzy clustering of crowdsourced test reports for apps," *ACM Trans. Internet Technol.*, vol. 18, no. 2, 2018, Art. no. 18.
- [73] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma, "VIPS: A vision-based page segmentation algorithm," Tech. Rep. MSR-TR-2003-79, 2003, pp. 1–29, [Online]. Available: <https://www.microsoft.com/en-us/research/publication/vips-a-vision-based-page-segmentation-algorithm/>
- [74] S. Roy Choudhary, M. R. Prasad, and A. Orso, "X-pert: A web application testing tool for cross-browser inconsistency detection," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 417–420.
- [75] A. Michail, "Browsing and searching source code of applications written using a GUI framework," in *Proc. 24th Int. Conf. Softw. Eng.*, 2002, pp. 327–337.
- [76] D. Liu, X. Zhang, Y. Feng, and J. A. Jones, "Generating descriptions for screenshots to assist crowdsourced testing," in *Proc. IEEE 25th Int. Conf. Softw. Anal. Evol. Reeng.*, 2018, pp. 492–496.
- [77] A. Michail and T. Xie, "Helping users avoid bugs in GUI applications," in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 107–116.
- [78] J. Wang, M. Li, S. Wang, T. Menzies, and Q. Wang, "Images dont lie: Duplicate crowdtesting reports detection with screenshot information," *Inf. Softw. Technol.*, vol. 110, pp. 139–155, 2019.



Di Liu received the ME degree from Soochow University. He is currently working toward the PhD degree in the Department of Computer Science and Technology, Nanjing University, China. His research interests include crowdsourced software engineering and program analysis.



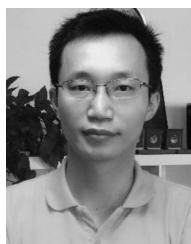
Yang Feng received the BE and ME degrees from Nanjing University, and the PhD degree in software engineering from the University of California, Irvine. He is now an assistant researcher with the State Key Laboratory for Novel Software Technology, Nanjing University. His research interests include the areas of the program comprehension, testing, debugging, program analysis, and crowdsourced software engineering.



Xiaofang Zhang is an associate professor with the School of Computer Science and Technology, Soochow University, China. Her research interests include the intersection of Software Engineering and Artificial Intelligence, including intelligent software engineering, software testing, and software defect prediction.



James A. Jones received the PhD degree from the Georgia Institute of Technology. He is an associate professor with the Department of Informatics, Donald Bren School of Informatics and Computer Science, University of California, Irvine. His research interests include the area of software testing, software analysis (run-time and compile-time), and debugging. He is particularly interested in supporting the creative and intellectual process of developing and maintaining software.



Zhenyu Chen is a full professor with the Software Institute of Nanjing University. His research interesting is in intelligent software engineering. He is the founder of MoocTest, the initiator of the *IEEE International Software Testing Competition*. He has served on ICSE, FSE, ASE, the *IEEE Transactions on Reliability* and so on. He has some efforts in MoocTest, BigCode and AIEngin. Some of them have been transferred into well-known software companies such as Baidu, Alibaba and Huawei.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

Redundancy, Context, and Preference: An Empirical Study of Duplicate Pull Requests in OSS Projects

Zhixing Li¹, Yue Yu¹, Minghui Zhou¹, Tao Wang, Gang Yin, Long Lan, and Huaimin Wang

Abstract—OSS projects are being developed by globally distributed contributors, who often collaborate through the pull-based model today. While this model lowers the barrier to entry for OSS developers by synthesizing, automating and optimizing the contribution process, coordination among an increasing number of contributors remains as a challenge due to the asynchronous and self-organized nature of distributed development. In particular, duplicate contributions, where multiple different contributors unintentionally submit duplicate pull requests to achieve the same goal, are an elusive problem that may waste effort in automated testing, code review and software maintenance. While the issue of duplicate pull requests has been highlighted, to what extent duplicate pull requests affect the development in OSS communities has not been well investigated. In this paper, we conduct a mixed-approach study to bridge this gap. Based on a comprehensive dataset constructed from 26 popular GitHub projects, we obtain the following findings: (a) Duplicate pull requests result in redundant human and computing resources, exerting a significant impact on the contribution and evaluation process. (b) Contributors' inappropriate working patterns and the drawbacks of their collaborating environment might result in duplicate pull requests. (c) Compared to non-duplicate pull requests, duplicate pull requests have significantly different features, e.g., being submitted by inexperienced contributors, being fixing bugs, touching cold files, and solving tracked issues. (d) Integrators choosing between duplicate pull requests prefer to accept those with early submission time, accurate and high-quality implementation, broad coverage, test code, high maturity, deep discussion, and active response. Finally, actionable suggestions and implications are proposed for OSS practitioners.

Index Terms—Duplicate pull requests, pull-based development model, distributed collaboration, social coding

1 INTRODUCTION

THE success of many community-based Open Source Software (OSS) projects relies heavily on a large number of volunteer developers [35], [64], [84], [86], who are geographically distributed and collaborate online with others from all over the world [43], [58]. Compared to the traditional email-based contribution submission [25], the pull-based model [40] on modern collaborative coding platforms (e.g., GitHub [9] and GitLab [10]) supports a more efficient collaboration process [115], by coupling code repository with issue tracking, review discussion and continuous integration, delivery and deployment [79], [110]. Consequently, an increasing number of OSS projects are adopting the synthesized pull-based

mechanism, which helps them improve their productivity [98] and attract more contributors [108].

However, while the increased number of contributors in large-scale software development leads to more innovations (e.g., unique ideas and inspiring solutions), it also results in severe coordination challenges [103]. Currently, one of the typical coordination problems in pull-based development is duplicate work [85], [114], due to the asynchronous nature of loosely self-organized collaboration [26], [84] in OSS communities. On the one hand, it is unreasonable for a core team to arrange and assign external contributors to carry out every specific task under the open source model [53], [54] (i.e., external contributors are mainly motivated by interest and intellectual stimulation derived from writing code, rather than requirements or assignments). On the other hand, it is impractical to expect external developers (especially newcomers and occasional contributors) to deeply understand the development progress of the OSS projects [41], [56], [83] before submitting patches. Thus, OSS developers involved in the pull-based model submit *duplicate pull requests* (akin to duplicate bug reports [24]), even though they collaborate on modern social coding platforms (e.g., GitHub) with relatively transparent [37], [94] and centralized [40] working environments. The recent study by Zhou *et al.* [114] has showed that complete or partial duplication is pervasive in OSS projects and particularly severe in some large projects (max 51 percent, mean 3.4 percent).

Notably, a large part of duplicates are not submitted intentionally to provide different or better solutions. Instead, contributors submit duplicates unintentionally because of

• Zhixing Li, Tao Wang, Gang Yin, and Huaimin Wang are with the Key Laboratory of Parallel and Distributed Computing, College of Computer, National University of Defense Technology, Changsha 410073, China.

E-mail: {lizhixing15, taowang2005, yingang, hmwang}@nudt.edu.cn.

• Yue Yu is with the Key Laboratory of Parallel and Distributed Computing, College of Computer, National University of Defense Technology, Changsha, China, and also with Peng Cheng Laboratory, Shenzhen 518066, China.

E-mail: yuyue@nudt.edu.cn.

• Minghui Zhou is with the School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China, and also with Peng Cheng Laboratory, Shenzhen 518066, China. E-mail: zhmh@pku.edu.cn.

• Long Lan is with the Peng Cheng Laboratory, Shenzhen 518066, China.

E-mail: long.lan@pcl.ac.cn.

Manuscript received 21 Dec. 2019; revised 18 Aug. 2020; accepted 19 Aug. 2020.

Date of publication 24 Aug. 2020; date of current version 18 Apr. 2022.

(Corresponding author: Yue Yu.)

Recommended for acceptance by E. Murphy-Hill.

Digital Object Identifier no. 10.1109/TSE.2020.3018726

misinformation and unawareness of a project's status [41], [114]. In practice, duplicate pull requests may cause substantial friction among external contributors and core integrators; these duplicates are a common reason for direct rejection [41], [85] without any chance for improvement, which frustrates contributors and discourages them from contributing further. Moreover, redundant work is more likely to increase costs during the evaluation and maintenance stages assembled with DevOps tools compared to traditional development models. For example, continuous integration tools (e.g., Travis-CI [98], [104]) automatically merge every newly received pull request into a testing branch, build the project and run existing test suites, so computing resources are wasted if integrators do not discover the duplicates and stop the automation process in time. Therefore, avoiding duplicate pull requests is becoming a realistic demand for OSS management, e.g., scikit-learn provides a special note in the contributing guideline "*To avoid duplicating work, it is highly advised that you search through the issue tracker and the PR list. If in doubt about duplicated work, or if you want to work on a non-trivial feature, it's recommended to first open an issue in the issue tracker to get some feedbacks from core developers.*" [5]

Existing work has highlighted the problems of duplicate pull requests [40], [85], [114] (e.g., inefficiency and redundant development), and proposed ways to detect duplicates [57], [73]. However, the nature of duplicate pull requests, particularly the fine-grained resources that are wasted by the duplicates, the context in which duplicates occur, and the features that distinguish merged duplicates from their counterparts, have rarely been investigated. Understanding these questions would help mitigate the threats brought by duplicate pull requests and improve software productivity.

Therefore, we bridge the gap on the investigation of duplicate pull requests in this study. We extend our previously collected duplicate pull request dataset [106] by adding change details, review history, and integrators' choice. Based on the dataset, we analyze the redundancies of duplicate pull requests in the development and evaluation stages, explore the context in which duplicates occur and examine the difference between duplicate and non-duplicate pull requests. We further investigate the reasons why among a group of duplicates, a pull request is more likely to be accepted by an integrator. Finally, we propose actionable suggestions for OSS communities.

The main contributions of this paper are summarized as follows:

- It presents empirical evidence on the impact of duplicate pull requests on development effort and review process. The findings will help software engineering researchers and practitioners better understand the threats of duplicate pull requests.
- It reveals the context of duplicate pull requests, highlighting the inappropriateness of OSS contributors' work patterns and the shortcomings of the current OSS collaboration environment. These findings can guide developers to avoid redundant effort on the same task.
- It provides quantitative insights into the difference between duplicate and non-duplicate pull requests,

which can offer useful guidance for automatic duplicate detection.

- It summarizes the characteristics of the accepted pull requests compared to those of their duplicate counterparts, which will provide actionable suggestions for inexperienced integrators in duplicate selection.

The rest of the paper is organized as follows: Section 2 introduces the background and research questions. Section 3 presents the dataset used in this study. Sections 4, 5 and 6 report the experimental results and findings. Section 7 provides further discussion and proposes actionable suggestions and implications for OSS practitioners. Section 8 discusses the threats to the validity of the study. Finally, we draw conclusions in Section 9.

2 BACKGROUND AND RESEARCH QUESTIONS

2.1 Pull-Based Development

In the global collaboration of OSS projects, a variety of tools [115], including mailing lists, bug trackers (e.g., Bugzilla [3]), and source code version control systems (e.g., SVN and Git), have been widely used to facilitate collaboration processes. The pull-based development model is the latest paradigm [40] for distributed development; it integrates code base with task management, code review and DevOps toolset. Compared with the traditional patch-based model, the pull-based model provides OSS developers with centralization of information, integration of tools, and process automation, thus simplifying the participation process and lowering the entry barrier for contributors [40], [98]. In addition, the pull-based model separates the developers into two teams, i.e., the external contributor team, which does not have the write access to the repository and submits contributions via pull requests, and the core integrator team, which is responsible for assessing and integrating the pull requests sent from external contributors. This decoupling of effort stimulates and enhances the parallel and distributed collaboration among OSS developers [115]. As shown in Fig. 1, the pull-based development workflow [31] includes the following steps.

- a) *Fork*: For a contributor (*Bob* or *alice*) in GitHub, the first step to contribute to a project is forking its original repository. As a result, the contributor owns a copy of the repository containing all the source code and commit histories under her/his GitHub account. Both the original repository and the forked repository are hosted on the servers of GitHub.
- b) *Clone*: Before the contributor engages in actual work, s/he must clone the forked repository to her/his local computer and makes code changes based on the local repository.
- c) *Edit*: The contributor can then fix bugs or add new features by editing the local repositories. Moreover, the contributor is recommended to always create a *topic branch* separated from the *master branch* and to commit local changes to that topic branch.
- d) *Sync*: It is possible that the local repository becomes out of date compared with the original repository. To make it easy for project integrators to merge the local changes cleanly, the contributor is expected

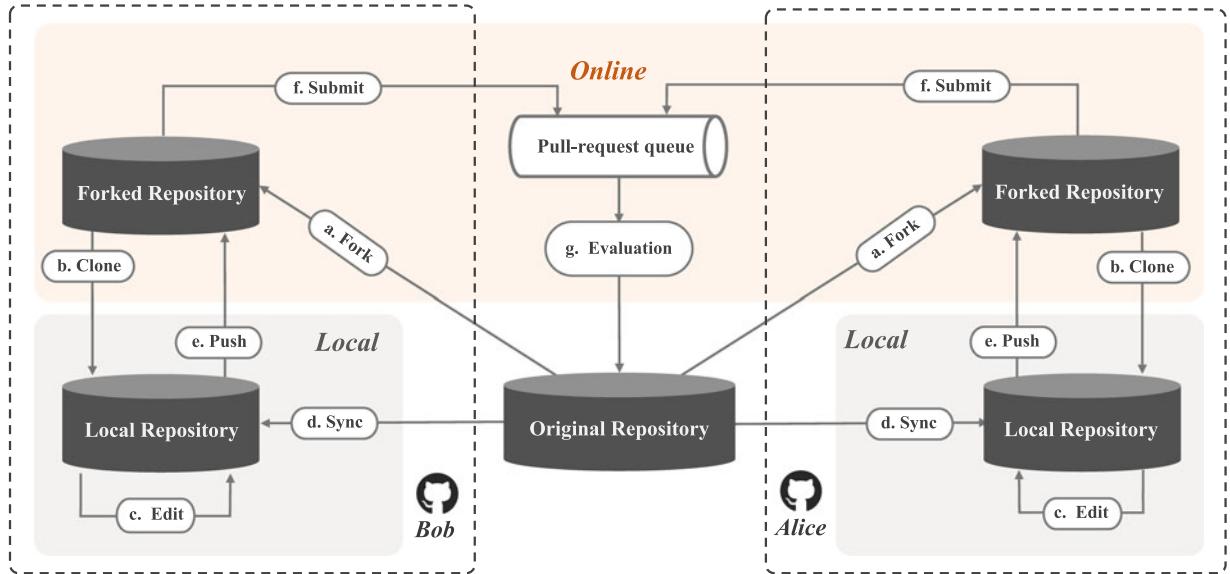


Fig. 1. The workflow of the pull-based development model.

first to sync the latest commits from the original repository and handle the possible merge conflicts.

- e) *Push*: The contributor then pushes the local changes to the forked repository. The forked repository acts as a transfer station of the local changes from the local repository to the original repository.
- f) *Submit*: Based on the forked repository, the contributor issues a pull request to notify the project integrators to merge (i.e., pull) the pushed commits. The pull request consists of a title and a description, which are used as a straightforward elaboration of the contained commits. All pull requests submitted to a specific project are maintained in a queue (i.e., the issue tracker), and each developer can check the status and review histories of the pull requests.
- g) *Evaluation*: To ensure that the submitted changes do not contain defects and adhere to project conventions, the project integrators and other community developers who are interested in the project discuss the appropriateness and quality of the pushed changes. Finally, the integrators reach an agreement on whether to accept the changes after several rounds of discussion.

2.2 Evaluation and Decision of OSS Code Contributions

The evaluation and decision of OSS contributions is a comprehensive process [93], [98], [107], in which the code reviewers would consider various factors. Previous studies have attempted to use quantitative methods to uncover the characteristics of accepted contributions or use qualitative methods to explore the factors that integrators examine when making decisions. Those studies provide a good guidance on which factors should be considered as controls when we analyze the characteristics of and integrators' preference among duplicate pull requests.

Characteristics of Accepted Contributions. Rigby *et al.* [76] analyzed the patches submitted to the Apache server project, and found that patches of small size are more likely to be accepted than large ones in that project. The similar

finding was also reported by Weissgerber *et al.* [101] in their study on two other OSS projects. Jiang *et al.* [47] conducted a case study on the Linux kernel project, which showed that patches submitted by experienced developers, patches of high maturity, and patches changing popular subsystems are more likely to be accepted. Baysal *et al.* [20] also found that developer's experience has a positive effect on patch acceptance in the WebKit and Google Blink projects. In the pull request model specifically, Gousios *et al.* [40] found that the hotness of project area is the dominating factor affecting pull request acceptance. Tsay *et al.* [93] investigated the effects of both technical and social factors on pull request acceptance. Their findings showed that stronger social connection between the pull request author and integrators can increase the likelihood of pull request acceptance. Yu *et al.* [108] investigated pull request evaluation in the context of continuous integration (CI). They found that CI testing results significantly influence the outcome of pull request review and the pull requests failing CI tests have a high likelihood of rejection. Kononenko *et al.* [52] studied the pull requests submitted to a successful commercial project. Their analysis results presented that patch size, discussion length, and authors' experience and affiliation are important factors affecting pull request acceptance in that project. A recent study conducted by Zou *et al.* [116] showed that pull requests with larger code style inconsistency are more likely to be rejected.

Factors That Integrators Examine When Making Decisions. Rigby *et al.* [77] interviewed with nine core developers from the Apache server project on why they rejected a patch. Although technical issue is the dominating reason, the reported reasons also include project scope and other political issues. Pham *et al.* [68] interviewed with project owners from GitHub, and found that many factors are considered by project owners when evaluating contributions, e.g., the trustworthiness of contributor and the size, type and target of changes. Tsay *et al.* [94] analyzed integrators' comments and decisions on highly discussed pull requests. They found that integrators are usually polite to new contributors for social encouragement, and integrators' decisions can be

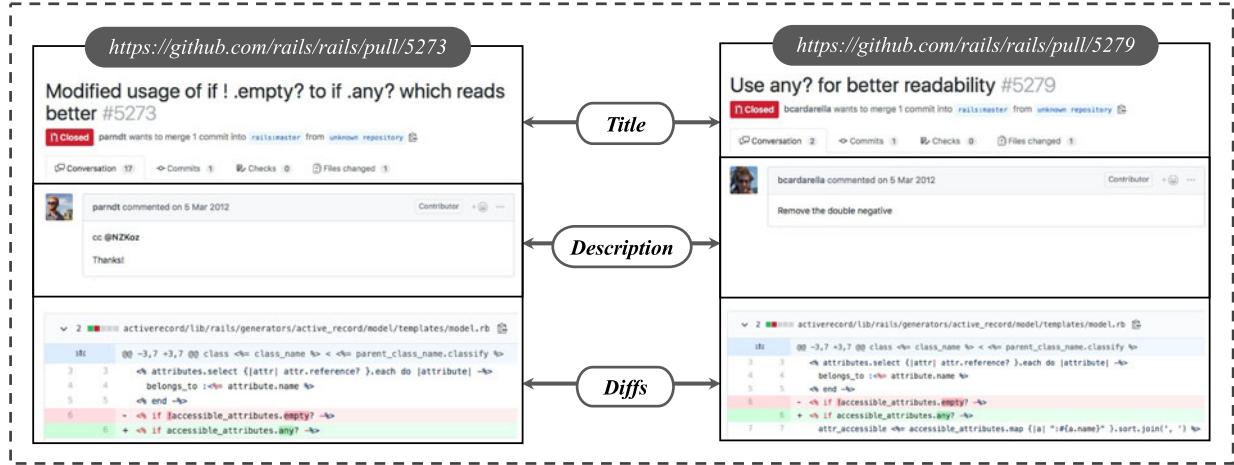


Fig. 2. An example of a pair of duplicate pull requests.

affected by community support. Gousios *et al.* [42] surveyed hundreds of integrators from GitHub on how they decide whether to accept a pull request. Their survey results showed that the most frequently mentioned factors are contribution quality, adherence to project norm, and testing results. Tao *et al.* [89] analyzed the rejected patches from Eclipse and Mozilla, and derived a list of reasons for patch rejection, e.g., compilation errors, test failures, and incomplete fix. Kononenko *et al.* [51] surveyed core developers from the project Mozilla and asked them about the top factors affecting the decision of code review. They found that the experience of developers receives the overwhelming number of positive answers. In a recent study, Ford *et al.* [39] investigated the pull request review process on the basis of an eye-tracker dataset collected from direct observation of reviewers' decision making. Interestingly, they found that developers reviewing code contributions in GitHub actually examined the social signals from profile pages (e.g., avatar image) more than they reported.

2.3 Duplicates in Community-Based Collaboration

Duplicate efforts, including duplicate bug reports, duplicate questions, and, recently, duplicate pull requests have been studied in the literature. Studies have mainly focused on revealing the threats in duplicates, and proposed methods to detect and remove duplicates.

Duplicate Bug Reports. Many OSS projects incorporate bug trackers so that developers and users can report the bugs they have encountered [78], [100]. From Bugzilla to GitHub issue system [2], bug tracking systems have become more lightweight, which makes it easier to submit bug reports. Consequently, popular OSS projects can receive hundreds of bug reports from the community every day [87]. However, because the reporting process is uncoordinated, some bugs might be reported multiple times by different developers. For example, duplicate issues account for 36 and 24 percent of all reported issues in Gnome and Mozilla [112], respectively, and consume considerable effort from developers to confirm. Meanwhile, researchers have proposed various methods to automatically detect duplicate bug reports. Most of them used similarity-based methods that compute the similarity between a given bug report and previously submitted reports based on various information,

including natural language text [78], [100], execution trace [100], categorical features of bugs [87]. Other work has used a machine learning-based classifier [55], [88] and a topic-based model [65] to improve detection performance. It is also interesting to observe that duplicates may attract less attention and consume less effort. For example, Zhou and Mockus found that the issues resolved with FIXED tend to have more comments than other issues, and issues with resolution DUPLICATE tend to have the least comments [112].

Duplicate Questions. Stack Overflow [13] is currently the most popular programming Q&A site where developers ask and answer questions related to software development and maintenance [91]. In Stack Overflow, developers can vote on the quality of any question and answer, and they can gain reputation for valued contributions. Since its founding in 2008, Stack Overflow has accumulated 20 million questions and answers and attracted millions of visitors each month. Because of the large user base, Stack Overflow also faces the challenge of receiving duplicate questions posted by different developers, despite of its explicit suggestion that developers conduct a search first before posting a question. Prior studies have investigated the detection of duplicate questions in Stack Overflow. Zhang *et al.* [111] computed the similarity between two questions based on the titles, descriptions and tags of the questions and latent topic distributions, and they recommended the most similar questions for a given question. To improve detection accuracy, Mizobuchi *et al.* [62] used word embedding to overcome the problem of word ambiguities and catch up new technical words. Zhang *et al.* [109] leveraged continuous word vectors, topic model features and frequent phrases pairs to capture semantic similarities between questions. Moreover, Mizobuchi *et al.* [17] investigated why duplicate questions are submitted and found that not searching for the questions is the most frequent reason. However, submitting questions is significantly different from submitting pull requests in both the form and the process.

Duplicate Pull Requests. Fig. 2 shows an example of a pair of duplicate pull requests, both of which replace function `empty?` with `any?` for better readability. Duplicate pull requests both go through the normal evaluation process until their duplicate relation is identified by the reviewers. Prior studies have reported that duplicate pull requests are

a pervasive and severe problem that affects development efficiency [85], [114]. Gousios *et al.* [40] found that more than half of sampling pull requests were rejected due to non-technical reasons and duplication is one of the major problems. Steinmacher *et al.* [85] conducted a survey with the quasi-contributors to obtain their perspectives on reasons for pull request nonacceptance, and duplication was the most common reason mentioned by the quasi-contributors. Furthermore, to help reviewers find duplicate pull requests in a timely manner, researchers [57], [73] have proposed methods to automatically recommend similar pull requests. In addition, Zhou *et al.* [114] explored the weak evidence that discussing or claiming an issue before submitting a pull request correlates with a lower risk of duplicate work. In brief, the above studies have revealed the threats in duplicate pull requests and proposed automatic detection methods for duplicates, but to what extent duplicate pull requests affect the OSS development, the context in which duplicates occur and integrators' choice between duplicates remain unclear.

2.4 Goals and Research Questions

In this study, we aim to better understand the mechanism of distributed collaboration with pull requests, and to avoid duplication and redundancy in an actionable and effective way. In particular, the main goals of the paper are as follows.

Reveal the Impact of Duplicates. We aim to obtain quantitative evidence of the impacts of duplicate pull requests on the contribution and evaluation process to more clearly reveal the inefficiency of redundant development.

Guide OSS Practitioners. We hope our study can guide OSS contributors to improve their work patterns and avoid unintentional redundant efforts on the same task. Moreover, we expect to help integrators learn from the practices in dealing with duplicates and make more informed decisions about choosing between duplicates.

Inspire Tool Design. We also hope our findings can inspire the OSS community and researchers and provide some insight into how to design and develop mechanisms and tools to assist developers in avoiding, detecting, managing, and handling duplicate pull requests more effectively and efficiently.

To achieve our goals, we address the following research questions.

RQ1: How much effort do duplicate pull requests consume, and to what extent do they delay the review process?

Motivation: Duplicate pull requests submitted by multiple different developers are usually evaluated through the same rigorous review process as original ones. As a result, duplicate pull requests waste resources spent on separate and redundant programming and evaluation efforts. We attempt to quantify the redundant effort spent on duplicate pull requests.

RQ2: What is the context in which duplicate pull requests occur?

Motivation: As reported in prior studies [40], [98], [115], the pull-based development model is associated with higher contribution effectiveness than traditional patch-based model in terms of activity transparency and information centralization. Nevertheless, contributors are still at risk of conducting redundant development. Hence, we aim to reveal the practical factors resulting in duplicate pull requests.

RQ3: Which duplicate pull requests are more likely than their counterparts to be accepted?

Motivation: Prior research [42], [94] has studied the factors that should be examined when integrators decide whether to accept an individual pull request. However, little is known about integrators' preference for what kind of duplicate pull requests should be accepted. We hope to determine the characteristics of accepted duplicates compared to those of their counterparts and summarize the common practices of duplicate selection for integrators.

3 DATASET

In this study, we leverage our previous dataset *DupPR* [106], which contains the duplicate relations among pull requests and the profiles and review comments of pull requests from 26 popular OSS projects hosted on GitHub. We also extend the dataset by adding complementary data, including code commits, check statuses of DevOps tools and contribution histories of developers.

3.1 DupPR Basic Dataset

In our prior work [106], we have built an unique dataset of more than 2,000 pairs of duplicate pull requests (called *DupPR* [7]) by analyzing the review comments from 26 popular OSS projects hosted on GitHub. Each pair of duplicates in *DupPR* is represented in a quaternion as $\langle \text{proj}, \text{pr}_1, \text{pr}_2, \text{idn_cmt} \rangle$ (pr_1 was submitted before pr_2). Item proj indicates the project (e.g., *rails/rails*) that the duplicate pull requests belong to. Items pr_1 and pr_2 are the tracking numbers of the two pull requests, respectively. Item idn_cmt is a review comment of either pr_1 or pr_2 , which is used by reviewers to state the duplicate relation between pr_1 and pr_2 . The dataset meant to only contain the accidental duplicates of which all the authors were not aware of the other similar pull requests when creating their own. In order to increase the accuracy of this study, we recheck the dataset again and filter out the intentional duplicates that were not found before. Specifically, we omit duplicates from the dataset when they fit one of the following criteria: i) The authors' discussion on the associated issue reveals that the duplication was on purpose. A representative comment indicating intentional duplication is "*I saw your PR and there wasn't any activity or follow up in that from last 18 days, [so I create a new one.]*"¹; ii) The submitter of pr_2 has performed actions on pr_1 , including commenting, assigning reviewers and adding labels, which clearly indicates that s/he was aware of pr_1 before submitting her/his own one; and iii) The author of pr_2 immediately (< 1 min) mentioned pr_1 after creating pr_2 , which means that the author might already know that pull request before. Overall, we eliminate 330 pairs from *DupPR*.

A pull request might be duplicate of more than one pull request. Therefore, in this study, we organize a group of duplicate pull requests in a tuple structure $\langle \text{dup}_1, \text{dup}_2, \text{dup}_3, \dots, \text{dup}_n \rangle$ in which the items are sorted by their submission time. In total, we have 1,751 tuples of duplicate pull requests. Table 1 presents the quantitative overview of the

1. The sources of pull requests, issues and comments cited in this paper can be found online at <https://github.com/whystar/DupPR-cited>

TABLE 1
The Quantitative Overview of the Dataset

Metrics	Overall pull requests	DupPR
#Pull requests	333,200	3,619
#Contributors	39,776	2,589
#Reviewers	24,071	2,830
#Comments	2,191,836	39,945
#Checks	364,646	4,413

dataset. It lists the main metrics including the number of pull requests, the number of pull request contributors, the number of pull request reviewers and their review comments, and the number of pull request checks (introduced in Section 3.2.2).

3.2 Collecting Complementary Data

3.2.1 Patch Detail

GitHub API (`/repos/:owner/:repo/pulls/:pull_number/commits`) allows us to retrieve the commits on each pull request. From the returned results, we parse the sha of each commit and request the API (`/repos/:owner/:repo/commits/:commit_sha`) to return detailed information about a commit, including author and author_date. Moreover, the API (`/repos/:owner/:repo/pulls/:pull_number/files`) returns the files changed by a pull request, from which we can parse the filename and changes (lines of code added and deleted) of each changed file.

3.2.2 Check Statuses

Various DevOps tools are seamlessly integrated and widely used in GitHub; examples are Travis-CI [14] for continuous integration and Code-Climate [4] for static analysis. When a pull request has been submitted or updated, a set of DevOps tools are automatically launched to check whether the pull request can be safely merged back to the codebase. GitHub API (`/repos/:owner/:repo/commits/:ref/status`) returns the check statuses for a specific commit. There are two different levels of statuses in the returned results. Because multiple DevOps tools can be used to check a commit, each tool is associated with a check status, which we call the *context-level check status*. For each context-level check status, we can parse the state and context fields. The state of a check can be designated *success*, *failure*, *pending*, or *error*. State *success* means a check has successfully passed, while *failure* indicates that the check has failed. If the check is still running and no result is returned, its state is *pending*. State *error* indicates a check did not successfully run and produced an error. Following the guidelines of prior work [22], [81], we treat the state *error* as the same as *failure*, which are both opposed to *success*. The context indicates which tool is used in a specific check. According to the bot classification defined in prior study [102], the checking tools can be classified into three categories: CI (report continuous integration test results, e.g., *Travis-ci*), CLA (ensure license agreement signing e.g., *cla/google*), and CR (review source code e.g., *coverage/coveralls* and *codeclimate*). Based on all context-level check statuses of a commit, the API also returns a overall check status of that commit [8], which we call the *commit-level check status*. The state of a commit-level check can be one of *success*, *failure* and *pending*.

3.2.3 Timeline Events

GitHub API (`/repos/:owner/:repo/issues/:issue_number/events`) returns the events triggered by activities (e.g., assigning a label and posting a comment) in issues and pull requests. We request this API for each pull request. From the returned result, we can parse who (actor) triggered which event (event) at what time (created_at). For close events, we can parse which commit (commit_id, aka SHA) closed the pull request. Events data are mainly used for rechecking dataset and determining pull request acceptance.

3.2.4 Contribution Histories

Rather than requesting the GitHub API, we use the GHTorrent dataset [40], which makes it easier and more efficient to obtain the entire contribution history for a specific developer in GitHub. GHTorrent stores its data in several tables and we mainly use `pull_requests` (PR), `issues`, `pull_request_history` (PRH), `pull_request_comments` (PRC), and `issue_comments` (ISC). From table PR, table PRH, and table PRC, we can parse who (PRH.actor_id) submitted which pull request (PR.pullrequest_id) to which project (PR.base_repo_id) at what time (PRH.created_at) and who (PRC.user_id) have commented on that pull request at what time (PRC.created_at). Similarly, from table issues and table ISC, we can parse who (issues.reporter_id) reported which issue (issues.issue_id) to which project (issues.repo_id) at what time (issues.created_at) and who (ISC.user_id) commented on that issue at what time (ISC.created_at). Based on this information we can acquire the whole contribution history for a specific developer.

3.2.5 Popularity and Reputation

GHTorrent also provides tables relating to project popularity and developer reputation. From table `watchers`, we can parse who (user_id) started to star which project (repo_id) at what time (created_at). From table `projects`, we can parse which project (id) was forked from which project (forked_from) at what time (created_at). From table `followers`, we can parse who (followers) started to follow whom (user_id) at what time (created_at).

4 THE IMPACT OF DUPLICATE PULL REQUESTS

Although duplicate pull requests can bring certain benefits (e.g., they could complement each other and be combined to achieve a better solution), most unintentional duplicate pull requests are likely to waste resources during the asynchronous development and review process. In this section, we report a quantitative analysis that helps to better understand the impact of duplicate pull requests on development efforts and review processes.

4.1 Redundant Effort

Duplicate pull requests are organized as tuple structure in our dataset, i.e., $\langle dup_1, dup_2, dup_3, \dots, dup_n \rangle$. For each tuple, we identify the first received pull request (i.e., dup_1) as the ‘master’, and the following ones (i.e., $dup_i, i > 1$) as its ‘duplicates’. Since we want to quantify how much extra

TABLE 2
The Statistics of Code Patch Redundancy

	Min	25%	Median	75%	Max	Mean
#Files	0	1	2	3	3824	13.79
LOCs	0	4	16	70	82973	502.68

effort would be costed if the first contribution has been qualified, we accumulate the effort spent on all duplicates (i.e., $\sum_{i=2}^n dup_i$) as the redundancy. In this section, we analyze the redundant effort caused by duplicate pull requests from three perspectives, i.e., code patch, code review, and DevOps checking.

Code Patch Redundancy. A group of pull requests being duplicate means that multiple contributors have spent unnecessary redundant effort on implementing similar functionalities. We measure contribution effort for the code patch of a pull request with the number of changed files and LOCs (i.e., lines of code). The statistics is summarized in Table 2. We can see that each group of duplicates, on average, result in redundant contribution effort of changing more than 13 files (median of 2) and 502 LOCs (median of 16).

Code-Review Redundancy. For a group of duplicate pull requests, each one is reviewed separately until the reviewers detect the duplicate relation among them. That is, there is a detection latency of duplicate pull requests, and the review activities of the duplicates during that latency period are redundant. We define detection latency as the time period from the submission time of a pull request to the creation time of the first comment revealing the duplicate relation between it and other pull requests. Fig. 3 shows the distribution of detection latency. We find that almost half of the duplicates are detected after one day, and nearly 20 percent of them are detected after more than one week. The later the duplicate relation is detected, the more redundant review effort would be wasted.

Next, we compute the redundant review effort wasted during the detection latency, which is measured with the number of involved reviewers and the number of comments they have made. As shown in Table 3, there are, on average, more than 2 reviewers (median of 2) participating in the redundant review discussions and making more than 5 review comments (median of 3) before the duplicate relation is identified. This considerable redundancy cost reaches the standard number of contemporary peer review practices [74],

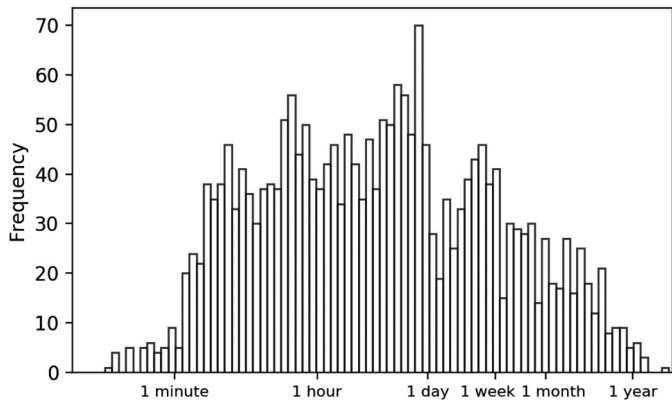


Fig. 3. Detection latency of duplicate pull requests.

TABLE 3
The Statistics of Code-Review Redundancy

	Min	25%	Median	75%	Max	Mean
#Reviewers	1	1	2	3	35	2.30
#Comments	1	2	3	6	148	5.68

[75] (i.e., median of 2 reviewers and 2-5 comments). Considering that the availability of reviewers has been discussed as one of the bottlenecks in large OSS projects' review process [107], code review redundancy can be avoided through automatic detection tools [57], [73] at submission time.

DevOps Redundancy. DevOps techniques, e.g., continuous integration, are widely used to improve code quality in GitHub. However, running DevOps services to check pull requests consumes a certain amount of resources and time. Moreover, both newly submitted pull requests and updates on existing pull requests trigger the launch of DevOps services. Therefore, duplicate pull requests waste valuable DevOps resources on redundant checking effort. In this paper, we measure the DevOps effort on a pull request by counting the total number of commit-level checks and context-level checks on this pull request. Table 4 lists the statistics for DevOps redundancy related to duplicates. We find that each group of duplicate pull requests, on average, cause 1.34 redundant commit-level checks and 2.87 redundant context-level checks. Thus, it is possible to improve DevOps efficiency by stopping or postponing the unnecessary checks of duplicates from the scheduling queue.

4.2 Delayed Review Process

The review duration, the number of reviewers and the number of comments made by these reviewers are important metrics for the efficiency of the pull request review process. Based on these metrics, we study whether the review process of duplicate pull requests differs from that of non-duplicate pull requests. We classify the duplicate pull requests into two groups: *MST* including the 'master' in each tuple, *DUP* including the 'duplicates' in each tuple. For comparison, we also create the *NON* group, which includes all non-duplicate pull requests from the corresponding projects. Figs. 4, 5, and 6 plot the statistics of the review duration, the number of reviewers and the number of review comments of pull request in each group, respectively. We observe that *MST* has notably longer review duration (median: 291.07 hours), compared with *NON* (median: 22.83 hours) and *DUP* (median: 21.75 hours). Moreover, compared with *NON*, both *MST* and *DUP* have more reviewers (medians: 3 versus 2, 3 versus 2) and more review comments (medians: 6 versus 3, 4 versus 3). Furthermore, we use the \tilde{T} -procedure [50] to compare their distributions pairwise. We opt for \tilde{T} because it is robust against unequal population variances and does not

TABLE 4
The Statistics of DevOps Redundancy

	Min	25%	Median	75%	Max	Mean
#CMT-Check	0	1	1	1	30	1.34
#CTT-Check	0	1	1	3	66	2.87

CMT-Check: commit-level checks; CTT-Check: context-level checks.

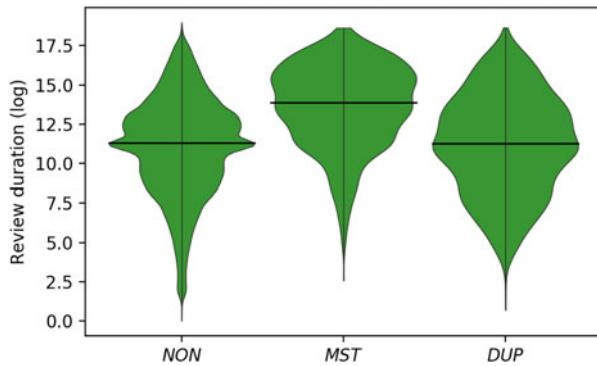


Fig. 4. The review duration of pull requests.

have the drawbacks of two-steps methods [95]. As shown in Table 5, all distributions are statistically different (p -value < 0.05), except for *NON-DUP* of review duration. And the signs of estimators also coincide with the observations from the figures, e.g., the estimator of *NON-MST* in terms of review duration comparison is less than 0 (-0.232) which indicates that *MST* has longer review time than *NON*. This suggests that the review process of duplicate pull requests is significantly delayed and involves more reviewers for extended discussions.

RQ1: *Duplicate pull requests result in considerable redundancy in writing code and evaluation. On average, each group of duplicate pull requests would result in code patch redundancy of more than 13 files and 500 lines of code, code-review redundancy of more than 5 review comments created by more than 2 reviewers, and DevOps redundancy of more than 1 commit-level check and more than 2 context-level checks. Moreover, duplicate pull requests significantly slow down the review process, requiring more reviewers for extended discussions.*

5 CONTEXT WHERE DUPLICATE PULL REQUESTS ARE PRODUCED

Despite of the increased activity transparency and information centralization in the pull-based development, developers still submitted duplicates. Thus, we further investigate the context in which duplicates occur and the factors leading pull requests to be duplicates.

First, we investigate the context of pull requests when duplicate occurs, as described in Section 5.1. In particular,

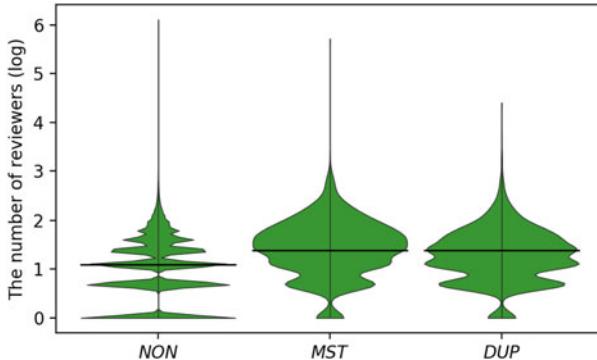


Fig. 5. The number of pull request reviewers.

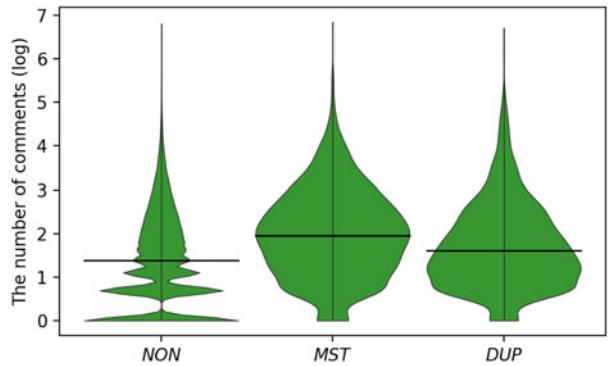


Fig. 6. The number of review comments on pull requests.

we examine the lifecycle of pull requests and discover three types of sequential relationship between two duplicate pull requests. For each relationship, we investigate whether contributors' work patterns and their collaborating environment have any flaw that may produce duplicate pull requests.

Second, we investigate the differences between duplicate and non-duplicate pull requests, as described in Section 5.2. We identify a set of metrics from prior studies to characterize pull requests. We then conduct comparative exploration and regression analysis to examine the characteristics that can distinguish duplicate from non-duplicate pull requests.

5.1 The Context of Duplicate Pull Requests

The entire lifecycle of a pull request consists of two stages: *local creation* and *online evaluation*. In the local creation stage, contributors edit the files and commit changes to their local repositories. In the online evaluation stage, contributors submit a pull request to notify the integrators of the original repository to review the committed changes online. These two stages are separated by the submission time of a pull request. For each pair of pull requests, there are only three types of sequential relationships in logic when comparing the order in which they enter each stage. We manually analyze contributors' work patterns, discussion and the collaborating environment to explore the possible context of duplicates in different relationships. In the following sections, we first elaborate on the three types of sequential

TABLE 5
Results of Multiple Contrast Test Procedure
for Review Process Measures

Pair	Estimator	Lower	Upper	Statistic	p-value
Review duration					
NON - MST	-0.232	-0.246	-0.218	-38.822	0.000 ***
NON - DUP	-0.012	-0.028	0.004	-1.771	0.172
MST - DUP	0.220	0.200	0.240	25.400	0.000 ***
Number of reviewers					
NON - MST	-0.191	-0.206	-0.176	-30.065	0.000 ***
NON - DUP	-0.114	-0.127	-0.100	-19.549	0.000 ***
MST - DUP	0.077	0.056	0.098	8.606	0.000 ***
Number of review comments					
NON - MST	-0.168	-0.182	-0.154	-27.077	0.000 ***
NON - DUP	-0.085	-0.098	-0.071	-14.338	0.000 ***
MST - DUP	0.083	0.063	0.104	9.305	0.000 ***

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$.

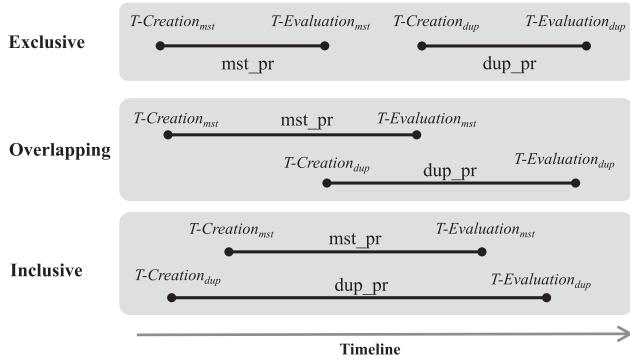


Fig. 7. The sequential relationship between two pull requests *mst_pr* and *dup_pr*.

relationships and then present the identified context of duplicate pull requests demonstrated by statistics and representative cases.

5.1.1 Types of Sequential Relationship

We first introduce two critical time points, *T – Creation* and *T – Evaluation*, in the lifecycle of pull requests; these time points are defined as follows.

- *T – Creation* indicates the start of pull request local creation. It is impossible to know the exact time at which a developer begins to work since developers only git commit when their work is finished rather than when the work is launched. However, we can still get an approximate start time. We set *T – Creation* as the *author_date* of the first commit packaged in a pull request, which is the earliest timestamp contained in the commit history of a pull request.
- *T – Evaluation* indicates the start of pull request online evaluation, i.e., the submission time of a pull request. This value is the *created_at* value of a pull request.

For a pair of duplicate pull requests $\langle mst_pr, dup_pr \rangle$ (*mst_pr* is submitted earlier than *dup_pr*), we suppose that the contributor of *mst_pr* begins to work at *T – Creation_{mst}* and submits *mst_pr* at *T – Evaluation_{mst}*, and the contributor of *dup_pr* starts to work at *T – Creation_{dup}* and submits *dup_pr* at *T – Evaluation_{dup}*. We discover three possible sequential relationships between *mst_pr* and *dup_pr*, as shown in Fig. 7.

- **Exclusive.** $T – Creation_{mst} < T – Evaluation_{mst} < T – Creation_{dup} < T – Evaluation_{dup}$, i.e., the author of *dup_pr* begins to work after the author of *mst_pr* has already finished the local work and submitted the pull request.
- **Overlapping.** $T – Creation_{mst} < T – Creation_{dup} \leq T – Evaluation_{mst} < T – Evaluation_{dup}$, i.e., the author of *dup_pr* starts working after the author of *mst_pr* starts working and before the author of *mst_pr* finishes working.
- **Inclusive.** $T – Creation_{dup} \leq T – Creation_{mst} < T – Evaluation_{mst} < T – Evaluation_{dup}$, i.e., although the author of *dup_pr* starts to work earlier than the author of *mst_pr* does, s/he submits the pull request later.

TABLE 6
Three Cases Involving the *STC* Pull Requests

Case	<i>mst_pr</i>	<i>dup_pr</i>
1	CTS	STC
2	STC	STC
3	STC	CTS

In the above, we discuss the common cases where developers first commit changed code and then submit a pull request. However, we also find rare cases where the pull request submission time is earlier than the first code-committing time. This might be due to the permission to submit ‘empty’ pull requests in the early stage of GitHub as indicated in its official document [15], or the incomplete record of developers’ updates to the pull request, e.g., force-push actions. The aforementioned definition of sequential relationship between *CTS* (commit-then-submit) pull requests does not apply to these *STC* (submit-then-commit) pull requests. To demonstrate this kind of situation, we define *T – Exposure* to indicate the exposure time of developer’s ideas for *STC* pull requests. *T – Exposure* is also set to be the *created_at* of pull requests. Next, we discuss the sequential relationship between two pull requests, of which at least one is a *STC* pull request.

As shown in Table 6, there are three specific cases involving *STC* pull requests. In *case 1* ($T – Creation_{mst} < T – Evaluation_{mst} < T – Exposure_{dup}$) and *case 2* ($T – Exposure_{mst} < T – Exposure_{dup}$), the local work or the idea of *mst_pr* has been exposed to the community before *dup_pr* is submitted. Therefore, we treat the sequential relationship between a pair of pull requests in these two cases as exclusive. In *case 3*, there are two possible situations: a) $T – Exposure_{mst} < T – Creation_{dup} < T – Evaluation_{dup}$ means that the idea of *mst_pr* has been exposed before the author of *dup_pr* starts to work and their sequential relationship can be seen as exclusive. b) $T – Creation_{dup} < T – Exposure_{mst} < T – Evaluation_{dup}$ means the author of *dup_pr* starts to work before the idea of *mst_pr* is exposed and finishes the work after that; therefore, their sequential relationship can be seen as inclusive.

Finally, to explore the distribution of the three types of relationships in our dataset, we convert each tuple of duplicate pull requests ($\langle dup_1, dup_2, dup_3, \dots, dup_n \rangle$) to pairs: (dup_i, dup_j) , where $1 \leq i, j \leq n$ and $i < j$. Table 7 shows the distribution, and we can see that the majority of duplicate pull request pairs have an exclusive sequential relationship (i.e., the duplicate contribution begins to work after the original pull request has been visible), which suggests that it is still a great challenge of awareness and transparency [37] during collaboration process.

5.1.2 Context of Exclusive Duplicate Pull Requests

Not Searching for Existing Work. For a pair of exclusive duplicate pull requests, there is a time window during which the author of *dup_pr* had a chance to figure out the existence of *mst_pr*. However, the author failed to do so and finally submitted a duplicate pull request. For example, contributors did not search the existing pull requests for similar work (e.g., the typical responses in duplicates: “Oh, Sorry I did not

TABLE 7
The Distribution of Different Sequential Relationships in the Dataset

	Exclusive	Overlapping	Inclusive
Count	1,924	17	81

search for a previous PR before submitting a PR” and “Ah should have searched first, thanks”). In some cases, developers’ search was not complete because they only searched the open pull requests and missed the closed ones (e.g., “Ah, my bad. I thought I searched, but I must have only been looking at open”). The survey conducted by Gousios [41] also showed that 45 percent contributors occasionally or never check whether similar pull requests already exist before coding.

Diversity of Natural Language Usages. Some developers tried to search for existing duplicates, but they ultimately found nothing (e.g., “Sorry, I searched before pushing but did not find your PR...”). One challenge is the diversity of natural language usages. For a pair of duplicate pull requests, we compute the common words ratio based on their titles, which is calculated by the following formula.

$$CWR(mst_pr, dup_pr) = \frac{|WS_{mst_pr} \cap WS_{dup_pr}|}{|WS_{mst_pr}|}. \quad (1)$$

WS_{mst_pr} and WS_{dup_pr} represent the set of words extracted from the titles of mst_pr and dup_pr , respectively, after necessary preprocessing like tokenizing, stemming [61], and removing common stop words. Fig. 8 shows the statistics of common words ratio. Approximately half of them have a value less than 0.25, which means a pair of duplicates tend to share a small proportion of common words. That is to say, a keyword-based query cannot always successfully detect existing duplicate pull requests due to the difference in wording for the same concept. For example, the title of *angular/angular.js*#4916 is “Fixed step12 correct file reference” and the title of *angular/angular.js*#4860 is “Changed from phone-list.html to index.html”. We can see that the two titles share no common word although the two pull requests have edited the same file and changed the code in the same line.

Disappointing Search Functionality in GitHub. Another challenge that can cause ineffective searching for duplicates is that GitHub’s search functionality might be disappointing in retrieving similar pull requests even though they share common words. For example, the titles of *angular/angular.js*#5063 and *angular/angular.js*#7846 are “fix(copy): preserve prototype chain when copying object” and “Use source object prototype in object copy”, respectively, which share three common critical words, i.e., *prototype*, *copy*, and *object*. For testing purpose, we launch a query in GitHub using the keywords *prototype copy object*. We retrieve 9 pages (each page containing

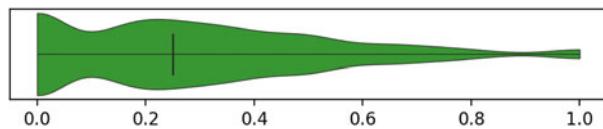


Fig. 8. The statistics of common words ratio between duplicates.

TABLE 8
The Statistics of Exclusive Intervals (in hour)

Interval	Min	25%	Median	75%	Maxs	Mean
	0.004	23.81	212.59	1276.82	29377.12	1397.59

10 items) of issues and pull requests in the search results and we finally find *angular/angular.js*#5063 in the 7th page. It is unlikely that developers have the willingness and patience to look through 7 pages of search results to figure out the existence of duplicates, since people tend to focus on the first few pages [49]. Perhaps that is exactly what leads the author of *angular/angular.js*#7846 to submit a duplicate, although he blamed the failed retrieval on himself (“apologies, I did search before posting (forget the search term I used) but clearly my search was bad... Thanks for finding the dup”).

Large Searching Space. Developers might manually look through the issue tracker to search for duplicates rather than retrieving through a query interface. Sometimes it is hard to find out the existing duplicates due to large searching space. The statistics of exclusive intervals between duplicates is listed in Table 8. On average, the local work of *dup_pr* is started approximately 1,400 hours (i.e., more than 58 days) after *mst_pr* has been submitted. During that long period, many new pull requests have been submitted in popular projects. For example, 307 pull requests were submitted between *pandas-dev/pandas*#9350 and *pandas-dev/pandas*#10074. These pull requests can occupy more than 10 pages in the issue tracker, which makes it rather hard and ineffective to review historical pull requests page by page, as a developer stated “...This is a dup of that PR. I should have looked harder as I didn’t see that one when I created this one...”.

Overlooking Linked Pull Requests. When developers submit a pull request to solve an exiting GitHub issue, they can build a link between the pull request and the issue by referencing the issue in the pull request description. The cross-reference is also displayed in the discussion timeline of the issue. Links not only allow pull request reviewers to find out the issue to be solved by a pull request but also help developers who are concerned with an issue to discover which pull requests have been submitted for that issue. In some cases, contributors did not examine or did not notice the linked pull requests to an issue (e.g., “Uhm yeah, didn’t spot the reference in #21967” and “Argh, didn’t see it in the original issue. Need more coffee I guess”) to make sure that no work had already been submitted for that issue, and consequently submitted a duplicate pull request.

Lack of Links. If a developer does not link her/his pull request to the associated issue, other developers might asynchronously do the duplicate work to fix the same one. For example, a developer *Dev2* submitted a pull request *facebook/react*#6135 trying to address the issue *facebook/react*#6114. However, *Dev2* was told that a duplicate pull request *facebook/react*#6121 was already submitted by *Dev1* before him. The conversation between *Dev1* and *Dev2* (*Dev2* said “I’m glad to hear that. But please link your future PRs to the issues”, and *Dev1* replied “Yeah I will, that’s on me!”) revealed that the lack of the link accounted for the duplication.

Missing Notifications. If developers have watched [37], [80] a project, they receive notifications about events that

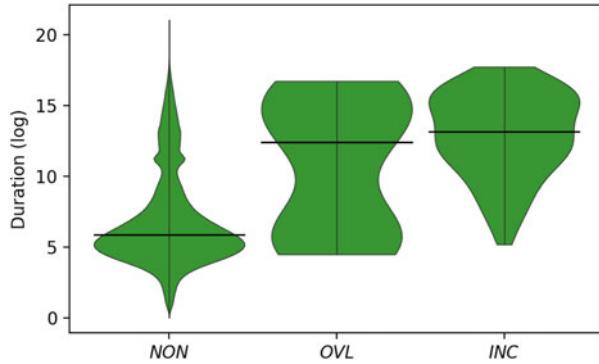


Fig. 9. Duration of local work in each group.

occur in the project, e.g., new commits and pull requests. The notifications are displayed in developers' GitHub dashboard and, if configured, sent to developers via email. However, developers might miss the important notifications due to information overload [36], and eventually submit a duplicate. For example, *kubernetes/kubernetes/#43902* was duplicate of *kubernetes/kubernetes/#43871* because the author of *kubernetes/kubernetes/#43902* missed the creation notification of the early one (as the author said "*missed the mail for the PR it seems :-/*").

5.1.3 Context of Overlapping and Inclusive Duplicates

Unawareness of Parallel Work. Developers who encounter a problem might prefer to fix the problem by themselves and submit a pull request, instead of reporting the problem in the issue tracker and waiting for a fix. When a problem is encountered by two developers at the same time, regardless of which developer is the first to work on the problem, the other developer might also start to work on the problem before the first developer submits a pull request. In such cases, both developers are unaware of concurrent activities of each other, because their local work is conducted offline and is not publicly visible. For example, the authors of *emberjs/ember.js/#4214* and *emberjs/ember.js/#4223* individually fixed the same typos in parallel without being aware of each other, and finally submitted two duplicate pull requests.

Implementing Without Claiming First. Sometimes, developers directly start to implement a patch for a GitHub issue without claiming (e.g., leaving a comment on the corresponding issue like "*I'm on it*"). This can introduce a risk that other interested developers might also start to work on the same issue without awareness of that there is already a developer working on that issue. For example, although two developers were both trying to solve the issue *facebook/react/#3948*, neither of them claimed the issue before coding their patch. Finally, they submitted two duplicate pull requests *facebook/react/#3949* and *facebook/react/#3950*. The phenomenon that developers are not used to claim issues was also reported in previous research [114].

Missing Existing Claims. Although a public issue has been claimed by a developer, other OSS contributors still have a chance of missing the claim comments among issue discussions. For example, a developer *Dev1* first claimed the issue *scikit-learn/scikit-learn/#8503* by leaving a comment "*We are working on this*". However, another developer *Dev2* did not notice this claim as explained by herself: "*Ah, nope. I just*

TABLE 9
Results of Multiple Contrast Test Procedure
for Local Work Durations

Pair	Estimator	Lower	Upper	Statistic	p-value
OVL - NON	0.256	0.109	0.402	4.188	0.001 ***
NON - INC	-0.385	-0.427	-0.344	-22.251	0.000 ***
OVL - INC	-0.130	-0.294	0.035	-1.891	0.138

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$.

realized someone was also working on it after I committed". Consequently, *Dev2* and *Dev1* conducted duplicate development in parallel and submitted two duplicate pull requests *scikit-learn/scikit-learn/#8517* and *scikit-learn/scikit-learn/#8518*, respectively.

Overlong Local Work. For overlapping and inclusive duplicate pull request pairs, we calculate the local duration of the work started earlier. Specifically, we collect two groups of pull requests: (a) *OVL*, which includes *mst_pr* of each pair of overlapping duplicates, and (b) *INC*, which includes *dup_pr* of each pair of inclusive duplicates. Fig. 9 plots the duration statistics of each group together along the group *NON*, which includes all non-duplicate pull requests. We observe that compared with the pull requests in *NON*, the pull requests in *OVL* and *INC* have longer local durations regarding median measures. We also test the difference using the \tilde{T} -procedure test. As shown in Table 9, the difference is significant ($p\text{-value} < 0.05$), and the signs of estimators present consistent difference directions. This reveals that overlong local work delays the exposure time of work and thereby hinders late contributors from realizing in a timely fashion that someone has already done the same work. As discussed in [41], developers rarely recheck the existence of similar pull request after they have finished the local work.

RQ2-1: We identified 11 contexts where duplicate pull requests occur, which are mainly relating to developers' behaviors, e.g., not checking for existing work, not claiming before coding, not providing links, and overlong local work, and their collaborating environment, e.g., unawareness of parallel work, missing notifications, lack of effective tools for checking for duplicates.

5.2 The Difference Between Duplicate and Non-Duplicate Pull Requests

Although some specific cases could be effectively avoided if developers pay attention to their work patterns, duplicates are difficult to eradicate completely considering the distributed and spontaneous nature of OSS development. Therefore, automatic detection of duplicates is still needed to help reviewers dispose of duplicates faster and in a timely fashion. Given that prior studies have mainly used a similarity-based method to detect duplicate pull requests [57], [73], we are interested in exploring the difference between duplicate and non-duplicate pull requests from a comparative perspective, which could offer useful guidance to optimize detection performance. In particular, we want to observe what distinguishing characteristics of duplicate pull requests are leading them to be duplicates. First, we identify metrics that are used in prior research, as shown in Section 5.2.1.

Then, we compare duplicate and non-duplicate pull requests in terms of each metric, and check whether significant difference could be observed between them through statistical test, as described in Section 5.2.2. Furthermore, in Section 5.2.3, we apply a regression analysis to model the correlations between the collected metrics and pull requests' likelihood of being duplicates.

5.2.1 Metrics

As the difference between duplicates and non-duplicates has not been studied in past studies, our study is an exploratory analysis. Therefore, we identified highly related metrics which have been studied in the previous research in the area of OSS contribution including traditional patch-based development [20], [44], [47], [68], [101] and modern pull request development [27], [40], [52], [71], [93], [95], [96], [108], [114]. The selected metrics are classified into the following three categories.

A). Project-Level Characteristics.

Maturity. Previous studies [71], [93], [108] used the metric `proj_age`, i.e., the period of time from the time the project was hosted on GitHub to the pull request submission time, as an indicator of the project maturity.

Workload. Prior studies have characterized project workload using two metrics: `open_tasks` [108] and `team_size` [40], [93], [108], which are the number of open issues and open pull requests at the pull request submission time and the number of active core team members during the last three months, respectively.

Popularity. In measuring project popularity, the metrics `stars` and `forks`, i.e., the total number of stars and the total number of forks the project has got prior to the pull request submission, were commonly used in previous studies [27], [93].

Hotness. This metric is the number of total changes on files touched by the pull request three months before the pull request creation time [40], [108].

B). Submitter-Level Characteristics.

Experience. Developers' experience before they submit the pull request has been analyzed in prior studies [40], [47]. This measure can be computed from two perspectives: project-level experience and community-level experience. The former measures the number of previous pull requests that the developer have submitted to a specific project (`prev_pullreqs_proj`) and their acceptance rate (`prev_prs_acc_proj`). The latter measures the number of previous pull requests that the developer have submitted to GitHub (`prev_pullreqs`) and their acceptance rate (`prev_prs_acc`). When calculating acceptance rate, the determination of whether the pull request was integrated through other mechanisms than GitHub's merge button follows the heuristics defined in previous studies [40], [114]. We also use two metrics `first_pr_proj` and `first_pr` to represent whether the pull request is the first one submitted by the developer to a specific project and GitHub, respectively.

Standing. A dichotomous metric `core_team`, which indicates whether the pull request submitter is the core team member of the project, was commonly used as a signal of the developer's standing within the project [93], [108]. Furthermore, a continuous metric `followers`, i.e., the number of GitHub users that are following the pull request submitter,

was used to represent the developers' standing in the community [40], [93], [108].

Social Connection. The metric `prior_interaction`, which is the total number of events (e.g., such as commenting on issues and pull requests) prior to the pull request submission that the developer has participated in within the project, was usually used to measure the social connection between the developer and the project [93], [108].

C). Patch-Level Characteristics

Patch Size. Prior studies [40], [93], [95] quantified the size of a patch, i.e., the changes contained in the pull request, in different granularity. The commonly used metrics are the number of changed files (`files_changed`) and the number of changed lines of code added and deleted (`loc`).

Textual Length. This metric is computed by counting the number of characters in the pull request title and description [108].

Issue Tag. This metric indicates whether the pull request description contains links to other GitHub issues or pull requests [40], [108], such as "fix issue #1011". We determine this metric by automatically checking the presence of cross-references in the pull request description based on regular expression technique.

Type. Prior studies [44], [63] summarized that developers can make three primary types of changes: fault repairing (*FR*), feature introduction (*FI*), and general maintenance (*GM*). The change type (`change_type`) of the pull request is identified by analyzing its title and commit messages based on a set of manually verified keywords [63]. Prior studies [45], [97] also identified the types of developer activities on the basis of the types of changed files. We follow the classification by Hindle *et al.* [45], which includes four types: changing source code files (*Code*), changing test files (*Test*), changing build files (*Build*), and changing documentation files (*Doc*). This metric (`activity_type`) is determined by checking the names and extensions of the files changed by the pull request.

5.2.2 Comparative Exploration

In order to explore the difference between duplicate and non-duplicate pull requests, we compare them in terms of each of the collected metrics and study to what extent a metric varies across duplicate and non-duplicate pull requests. Specifically, we formulate a non-directional hypothesis which can be used when there is insufficient theory basis for the exact prediction (i.e., we do not predict the exact direction of the difference). The null hypothesis and the alternative hypothesis are defined as follows:

H_0 : duplicate and non-duplicate pull requests exhibit the same value of metric m .

H_1 : duplicate and non-duplicate pull requests exhibit different values of metric m .

$\forall m \in \{\text{proj_age}, \text{open_tasks}, \text{team_size}, \text{forks}, \text{stars}, \text{hotness}, \text{prev_pullreqs}, \text{prev_prs_acc}, \text{first_pr}, \text{first_pr_proj}, \text{prev_pullreqs_porj}, \text{prev_prs_acc_proj}, \text{core_team}, \text{followers}, \text{prior_interaction}, \text{loc}, \text{files_changed}, \text{text_len}, \text{issue_tag}, \text{change_type}, \text{activity_type}\}$

H_0 is tested with *Mann-Whitney-Wilcoxon* test [105] on continuous metrics and *Chi-square* test [72] on categorical metrics. The test results are listed in Table 10 which reports the p-value and effect size of each test. The p-values are

TABLE 10
Results of Hypothesis Test

Metric	Adjusted p-value	Effect size
Project-level characteristics		
proj_age	4.1e-21 ***	0.091
open_tasks	0.791	0.003
team_size	3.8e-41 ***	0.131
stars	2.1e-46 ***	0.139
forks	8e-61 ***	0.159
hotness	4.5e-36 ***	0.122
Submitter-level characteristics		
first_pr	9.9e-33 ***	0.045
prev_pullreqs	7.7e-94 ***	0.199
prev_prs_acc	8.5e-90 ***	0.205
first_pr_proj	6.4e-148 ***	0.148
prev_pullreqs_proj	1.1e-190 ***	0.285
prev_prs_acc_proj	8.5e-52 ***	0.173
core_team	2.5e-116 ***	0.192
followers	1.2e-20 ***	0.090
prior_interaction	5.1e-107 ***	0.212
Patch-level characteristics		
files_changed	4.6e-08 ***	0.050
loc	3e-16 ***	0.079
text_len	9.1e-66 ***	0.166
issue_tag	0.001 **	0.027
change_type	8.5e-26 ***	0.095
activity_type	3.4e-07 ***	0.003

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$.

adjusted using the *Benjamini-Hochberg (BH)* method [23] to control the false discovery rate. To measure the effect size, we use *Cliff's delta* (d) [59] as it is a non-parametric approach which does not require the normality assumption of a distribution.

We reject H0 and accept H1 when p-value is less than 0.05. We can see that the null hypothesis is rejected on all metrics except for open_tasks. This means that duplicate and non-duplicate pull requests are significantly different in terms of all metrics except for open_tasks. Following the previous guidelines [70], [90] on interpreting the effect size (trivial: $|d| \leq 0.147$; small: $0.147 < |d| < 0.33$; medium: $0.33 \leq |d| < 0.474$; large: $|d| \geq 0.474$), we find that the effect size of difference is generally small with a maximum of 0.285.

5.2.3 Regression Analysis

The comparative exploration does not consider the correlations between metrics. As a refinement, we apply a regression analysis to model the effect of the selected metrics on pull requests' likelihood of being duplicates.

Regression Modeling. We build a mixed effect logistic regression model which is fit to capture the relationship between the explanatory variables, i.e., the metrics discussed in Section 5.2.1, and a response variable, i.e., *is_dup*, which indicates whether a pull request is a duplicate. Since our dataset is nested in the aspect of project (i.e., the pull requests collected from 26 different projects), the selected metrics are modeled as fixed effects, and a new variable proj_id is modeled as a random effect, to mitigate the over-represented phenomena present in some of the projects. Instead of building one model with all metrics at once, we add one level metrics at a time and build a model, which

can check whether the addition of the new metrics can significantly improve the model. As a result, we compare the fit of three models: a) *Model 1*, which includes only project-level variables, b) *Model 2*, which adds the submitter-level variables, and c) *Model 3*, which adds patch-level variables. In the models, all numeric factors are log transformed (plus 0.5 if necessary) to stabilize variance and reduce heteroscedasticity [60]. We manually check the distributions of all variables, and conservatively remove not more than 3 percent of values as outliers with exponential distributions. This reduces slightly the size of our dataset onto which we build the regression models, but ensures that our models are robust against outliers [67]. In addition, we check for the correlation of coefficients and the Variance Inflation Factors (VIF below 5 as recommended [32]) among variables to overcome the effect of multicollinearity. Specifically, four metrics (forks, prev_pullreqs_proj, prev_prs_acc_proj, and first_pr) are removed due to multicollinearity. This process leaves us with 17 features, which can be seen in Table 11.

Analysis Results. The analysis results are shown in Table 11. In addition to the coefficient, standard error, and significance level for each variable, the table reports the area under the ROC curve (AUC), the marginal R-squared (R_m^2) and conditional R-squared (R_c^2) to quantify the goodness-of-fit of each model. We can see that the models can explain more variability in the data when considering both the fixed and random effects ($R_c^2 > R_m^2$). Overall, Model 3 performs better than the other two Models (AUC: 0.729 versus 0.700/0.719) and they have obtained consistent variable effects (i.e., there is no significant effect flipping from positive to negative and vice versa), therefore we discuss their effects based on Model 3.

With regard to project-level predictors, open_tasks and team_size have significant, positive effects. This suggests that the more open tasks (pull requests and issues) and active core team members at the submission time of a new coming pull request, the more likely the new pull request is a duplicate. Especially, open_tasks does not show a significant difference by making the comparison with a single hypothesis testing, but exhibits a strong positive effect when controlled for other confounds. The predictor stars has a strong, negative effect, which means that the more popular the project becomes the less likely the submitted pull request is a duplicate. Our explanation is that the popular projects have well-established codebase, contribution guidelines and collaborating process. Hot files tend to attract more contributions from the community, but surprisingly, there is a negative effect of the hotness metric. We assume that pull requests changing hot files are more likely to be reviewed faster and get accepted in a timely fashion. A quick review can allow the target issue to be solved in short time, which prevents others from encountering the same issue and submitting duplicate pull requests. We leave a deep investigation in the future work.

As for submitter-level predictors, prev_prs_acc has a negative effect and first_pr_proj has a positive effect when its value is TRUE. This suggests that pull requests submitted by inexperienced developers and newcomers are more likely to be duplicates. On the contrary, the predictors prior_interaction and core_team (TRUE) have significant, negative effects, which indicates that pull requests from core

TABLE 11
Statistical Models for the Likelihood of Duplicate Pull Requests

	Model 1			Model 2			Model 3		
	response: <i>is_dup</i> = 1			response: <i>is_dup</i> = 1			response: <i>is_dup</i> = 1		
	Coeffs.	Errors	Signif.	Coeffs.	Errors	Signif.	Coeffs.	Errors	Signif.
log(proj_age)	0.005	0.064		0.127	0.065	*	0.068	0.061	
log(open_tasks + 0.5)	0.247	0.043	***	0.199	0.042	***	0.192	0.042	***
log(team_size + 0.5)	0.152	0.080	.	0.209	0.079	**	0.256	0.080	**
log(stars + 0.5)	-0.046	0.013	***	-0.044	0.013	***	-0.049	0.013	***
log(hotness + 0.5)	-0.049	0.013	***	-0.010	0.013		-0.049	0.015	***
log(prev_pullreqs + 0.5)	-	-	-	-0.034	0.014	*	-0.020	0.014	
log(prev_prs_acc + 0.5)	-	-	-	-0.207	0.064	**	-0.221	0.064	***
first_pr_proj TRUE	-	-	-	0.254	0.055	***	0.231	0.055	***
log(followers + 0.5)	-	-	-	0.005	0.013		0.002	0.013	
core_team TRUE	-	-	-	-0.222	0.056	***	-0.207	0.056	***
log(prior_interaction + 0.5)	-	-	-	-0.035	0.011	**	-0.044	0.011	***
log(files_changed + 0.5)	-	-	-	-	-		0.098	0.030	**
log(loc + 0.5)	-	-	-	-	-		-0.049	0.015	***
log(text_len + 0.5)	-	-	-	-	-		0.120	0.017	***
issue_tag TRUE	-	-	-	-	-		0.102	0.038	**
change_type FI	-	-	-	-	-		-0.308	0.043	***
change_type GM	-	-	-	-	-		-0.368	0.060	***
change_type Other	-	-	-	-	-		-0.291	0.048	***
activity_type Test	-	-	-	-	-		-0.285	0.072	***
activity_type Build	-	-	-	-	-		0.020	0.084	
activity_type Doc	-	-	-	-	-		-0.317	0.059	***
activity_type Other	-	-	-	-	-		-0.006	0.060	
Akaike's Information Criterion (AIC):	37451.49			37059.59			36861.77		
Bayesian's Information Criteria (BIC):	37526.33			37198.58			37118.37		
Area Under the ROC Curve (AUC):	0.700			0.719			0.729		
Marginal R-squared (R_m^2):	0.03			0.05			0.08		
Conditional R-squared (R_c^2):	0.18			0.20			0.25		

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$.

team members and developers who have a stronger social connection to the project are less likely to be duplicates. This highlights that developers equipped with enough experience and those having a stronger relationship with the project do better in avoiding duplicated work.

For patch-level predictors, two size related predictors present opposite effects. The predictor *loc* has a negative effect, which indicates that pull requests changing more lines of code have a less chance of being duplicates. While the predictor *files_changed* has a positive effect, suggesting that pull requests changing more files are more likely to be duplicates. Generally speaking, the more lines of code a patch has changed, the more complicated and difficult a task it solves, which poses a barrier for potential contributors (i.e., decreasing the likelihood of duplication). While holding other variables constant, if a pull request has touched more files, it increases the probability of the patch being duplicate (or partial conflict) with others' code changes. We think this interesting result deserves a future investigation. The predictor *text_len* has a positive effect, indicating that pull requests with complex description are more likely to be duplicates. Longer description may indicate higher complexity and thus longer evaluation [108], which increases the likelihood of the same issue being encountered by more developers who might also submit a patch for the issue. The predictor *issue_tag* has a positive effect when its value is TRUE, suggesting that pull requests solving already tracked issues have greater chances of being duplicates. One possible reason is that tracked issues are already publicly visible, and they are

more likely to attract more interested developers and result in conflicts. In terms of change types (*change_type*), we can see that compared with pull requests of the type *FR*, pull requests of the type *FI*, *GM* and *Other* are less likely to be duplicates. We speculate that fixing bugs are more likely to produce duplicates because bugs tend to have general effect on a bigger developer base compared to new feature or maintenance requirements which might be specific to a certain group of developers. For activity types (*activity_type*), we notice that pull requests changing test files (*activity Test*) and documentation files (*activity Doc*) have less chances of being duplicates, compared with those changing source code files. OSS projects usually encourage newcomers to try their first contribution by writing documentation and test cases [6], [11]. We conjecture that activities changing source code files might require more effort and time to conduct the local work, which are more risky and prone to duplication.

RQ2-2: Duplicate pull requests are significantly different from non-duplicate pull requests in terms of project-level characteristics (e.g., changing cold files and submitted when the project has more active core team members), submitter-level characteristics (e.g., submitted from newcomers and developers who have weaker connection to the project), and patch-level characteristics (e.g., solving already tracked issues rather than non-tracked issues and fixing bugs rather than adding new features or refactoring).

6 INTEGRATORS' PREFERENCE BETWEEN DUPLICATE PULL REQUESTS

To investigate what kind of duplicate pull requests are more likely to be accepted than their counterparts, we first construct a dataset of the integrators' choice between duplicates, as described in Section 6.1. Then we perform two investigations.

First, as described in Section 6.2, we identify metrics from prior work in the area of patch evaluation and acceptance, and apply them in a regression model to analyze their effects on the response variable *accept*, which indicates whether a duplicate pull request has been accepted or rejected.

Second, we want to learn what exactly integrators examine when they accept a duplicate pull request rather than its counterparts. We thus manually inspect 150 randomly selected duplicate pairs and use the card sorting method [82] to analyze the integrators' explanations of their choice between duplicates, as described in Section 6.3.

6.1 The Dataset of Integrators' Choice Between Duplicate Pull Requests

We convert each duplicate pull request tuple into pairs (i.e., $\langle mst_pr, dup_pr \rangle$, mst_pr is submitted earlier than dup_pr), and collect integrators' choice on each pair of duplicate pull requests. However, GitHub does not explicitly label which duplicate has been accepted by the integrators. Therefore, we decide to determine integrators' choice based on pull request status. A pull request in GitHub may occupy a variety of status, i.e., *open*, *merged*, and *closed*. The status *open* means a pull request is still under review and the integrators have not made the final decision, while the status *closed* means that the review of the pull request has concluded and should no longer be discussed. If a pull request is in the status *merged*, the pull request has been accepted and the review is over.

The dataset construction consists of two steps: (a) filtering out non-compared duplicate pairs, and (b) comparing the statuses of duplicates, as elaborated in the following sections.

6.1.1 Filtering Out Non-Compared Duplicate Pairs

In the study, we focus on the duplicate pairs that integrators compared after detecting their duplicate relation. First, we exclude duplicate pairs in which both pull requests remain open. Then, we exclude duplicate pairs which were closed but not compared by integrators. Fig. 10 shows the different cases where the relation identification and decision-making between duplicates happened at different times. In case A, mst_pr was closed before dup_pr was submitted. In case B, although mst_pr was closed after dup_pr was submitted, mst_pr (or dup_pr) was closed before the duplicate relation between them was identified. In case C, the relation identification and decision-making between mst_pr and dup_pr happened after mst_pr and dup_pr were submitted and before they were closed. Duplicate pairs of cases A and B are excluded because integrators did not make a comparison between them before making decisions. Moreover, we also exclude from duplicate pairs of case C those in which one pull request was closed by its submitter before integrators left any comment. Finally, we exclude 875 non-compared duplicate pairs, and 1,147 duplicate pairs remain.

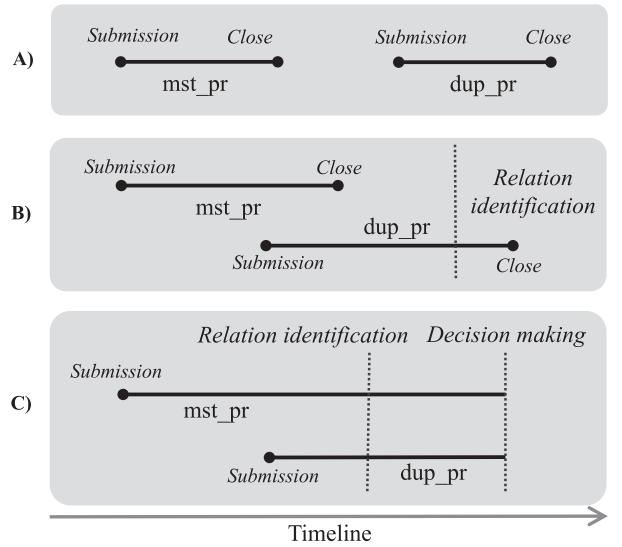


Fig. 10. Different cases where the relation identification and decision-making between duplicates happen at different times.

6.1.2 Comparing the Statuses of Duplicates

For a pair of duplicate pull requests, there are several combination of their statuses (e.g., one is merged and the other one is closed). In the following, we describe how to determine integrators' choice in different situations.

Only One is Merged. If one of the duplicate pull requests is merged and the other one is closed or open, the merged one has clearly been accepted by integrators.

Both are Merged. It is possible that both duplicates are accepted with some necessary coordination. For example, *rust-lang/rust*/#20380 was first merged, and *rust-lang/rust*/#20437 was then rebased and merged afterwards to provide an enhancement to the previous solution. In other cases, project integrators might also merge two duplicate pull requests to different branches; for example, *rails/rails*/#28068 and *rails/rails*/#28399 were merged to branches *master* and *5-0-stable*, respectively.

Both are not Merged. It is somewhat complicated to determine which one has been accepted in a pair of duplicates in which neither is merged (i.e., *(closed, open)* or *(closed, closed)* or *(open, closed)*). Unlike the status *merged*, which always means that a pull request has been accepted, the status *closed* can indicate that a pull request is accepted or rejected, which depends on the merge strategy of a specific project. In GitHub, project integrators can click the merge button on the web page to accept pull requests. However, integrators can also merge pull requests outside GitHub via several git commands. After that, integrators close the original pull requests. This means that closed pull requests might have been accepted. Prior research [40], [114] used a set of heuristics to automatically determine whether a closed pull request has been accepted. However, the heuristics are not completely reliable as some accepted pull requests might be mistakenly recognized as rejected, or vice versa. To avoid this bias, we manually examine the entire review history of a pair of duplicates to determine which one has been accepted. Additionally, it is possible that both duplicates in a pair are rejected. For example, *rails/rails*/#10737 and *rails/rails*/#10738 were both rejected by integrators because integrators think the change is not necessary.

TABLE 12
The Statistics of Integrators' Choice Between Duplicates

	Accept_One	Accept_Both	Reject_Both
Count	1,082	36	29

Finally, as shown in Table 12, we collect a total of 1,082 duplicate pairs in which only one is accepted. Subsequently, we conduct regression analysis and manual inspection based on those 1,082 duplicate pairs.

6.2 Regression Analysis

We conduct a regression analysis to investigate what factors would affect the chance that duplicate pull requests would be accepted by integrators. In the following sections, we present the selected predictors, regression models and analysis results.

6.2.1 Predictors

The predictor selection is based on prior work in the area of patch acceptance analysis. The selected predictors are split into three categories: submitter-level, patch-level, and review-level metrics. Compared with the study in Section 5.2, this study additionally includes review-level metrics because many signals in the code review process, e.g., review discussion length, are available after pull request submission, and past studies have found that review-level metrics have significant effects on pull request acceptance [40], [47], [52], [108]. However, this section does not include project-level metrics because two duplicate pull requests have the same project environment at the decision-making time. To start with, we discuss our metrics as follows.

A) Submitter-Level Metrics.

Developer experience is an important factor affecting patch acceptance. Prior studies [20], [40], [52] have shown that more experienced developers are more likely to get their patches accepted. To understand whether developer experience affects integrators' choice between duplicates, we still use six metrics to operationalize developer experience, i.e., `prev_pullreqs_proj`, `prev_prs_acc_proj`, `prev_pullreqs`, `prev_prs_acc`, `first_pr_proj` and `first_pr`, as discussed in Section 5.2. The previous studies showed that pull requests submitted by developers with higher standing are more likely to be accepted [93], [108]. To investigate the influence of developers' standing on integrators' choice, we include two metrics `core_team` and `followers` defined in Section 5.2. A developer's social relationship and interaction history with others can affect others' judgement for the developer's work [28], [42], [68]. Prior studies have found that pull requests from developers with a stronger social connection to the project have a higher chance to be accepted [93], [108]. In addition to the metric `prior_interaction` as defined earlier, we include the metric `social_strength`, which represents the proportion of decision-making group members that have co-occurred with the submitter in at least one discussion during the last three months, to examine the effects of social metrics on integrators' choice.

B) Patch-Level Metrics.

Prior studies [40], [93], [108] have found that large pull requests are less likely to be accepted. Integrators value small pull requests as they are easy to assess and integrate [42]. To investigate the effect of patch size on integrators' choice between duplicates, we include the two size-related metrics presented in Section 5.2, i.e., `files_changed` and `loc`. The study conducted by Yu *et al.* [108] showed that pull requests with longer description have higher chances to be rejected. It also revealed that pull requests containing links to issues have higher acceptance rates. For this, we include the metrics `text_len` and `issue_tag`, which are already defined in Section 5.2. From the prior work [37], [42], we learn that the existence of testing code is treated as a positive signal when integrators evaluate pull requests. Pull requests with test cases are more likely to be accepted [40], [93], [108]. To investigate this, we include a dichotomous metric `test_inclusion` to indicate whether the pull request has changed test files. Finally, in the context of selection between duplicate pull requests, we conjecture that the order that pull requests arrive might affect integrators' choice. The duplicate pull request submitted early might be more likely to be accepted than the late one, because people usually consider the recent one is redundant and should be closed in favor of the old one [12], [87], [88]. To verify our conjecture, we include a dichotomous metric `early_arrival` to indicate whether the pull request is submitted earlier than its counterpart.

C) Review-Level Metrics.

The participation metrics relating to human reviewers have been shown to affect the latency and outcome of pull request review [52], [93], [108]. For example, the study by Tsay *et al.* [93] showed that pull requests with a higher amount of comments are less likely to be accepted. To investigate the effects of human participation metrics on integrators' choice, we include two discussion-related metrics, i.e., the total number of comments on the pull request (`comments`) and the number of inline comments pointing to the source code (`comments_inline`). In addition, developers might leave tendentious comments on a pull request, according to whether they like or dislike a pull request [16], [94]. Following the philosophy of social coding [21], it is interesting to analyze whether the duplicate pull requests received more positive comments would likely win out, compared to those with more negative comments. To verify the hypothesis, we include two metrics: the proportion of the comments expressing positive sentiment (`comments_pos`) and the proportion of the comments expressing negative sentiment (`comments_neg`). To analyze the sentiment in pull request comments, we use the state-of-the-art sentiment analysis tool Senti4SD [30] retrained on the latest GitHub data [66]. Integrators also rely on automated testing tools to check pull request quality [42]. For example, prior study [108] found that the presence of CI test failure has a negative effect on pull request acceptance. To investigate this, we include three metrics `CI`, `CLA`, and `CR`, to represent the check statuses of the most recent commit in the pull request, which are returned by the three kinds of check tools discussed in Section 3. Finally, we include a metric `revisions`, i.e., how many times the pull request has been updated, to indicate the maturity of and the efforts that developers put on the pull

TABLE 13
Overview of Metrics

Metric	Mean	St.	Min	Median	Max
Submitter-level characteristics					
prev_pullreqs	1.7e2	3.1e2	0	45.00	3.5e3
prev_prs_acc	0.45	0.26	0.0	0.47	1.0
first_pr_proj	0.29	0.45	0	0.00	1
followers	3.1e2	1.5e3	0	41.00	3.3e4
core_team	0.33	0.47	0	0.00	1
social_strength	0.62	0.46	0.0	1.00	1.0
Patch-level characteristics					
early_arrival	0.50	0.50	0	0.50	1
files_changed	11.60	73.44	0	2.00	1.4e3
loc	6.0e2	5.0e3	0	17.00	9.9e4
test_inclusion	0.37	0.48	0	0.00	1
issue_tag	0.37	0.48	0	0.00	1
text_len	4.9e2	1.4e3	4	2.6e2	2.7e4
Review-level characteristics					
revisions	0.83	2.11	0	0.00	27
comments	5.18	11.84	0	2.00	3.3e2
comments_inline	1.55	5.42	0	0.00	1.1e2
comments_pos	0.17	0.27	0.0	0.00	1.0
comments_neg	0.04	0.13	0.0	0.00	1.0
CI	0.10	1.05	-1.0	0.00	3.0
CLA	-0.81	0.60	-1.0	-1.00	3.0
CR	-0.84	0.56	-1.0	-1.00	3.0

request. The prior study [47] showed that patch maturity is one of the major factors affecting patch acceptance.

6.2.2 Statistical Analysis

Our goal is to explain the relationship (if any) between the selected factors and the binary response variable *accept*, which indicates whether a duplicate pull request has been accepted over its counterpart (1 for accepted and 0 for not accepted). We use conditional logistic regression [33], implemented in the *mlogit* package of R, to model the likelihood of duplicate pull requests being accepted based on the matched pair samples in our dataset (i.e., two paired duplicate pull requests in which one is merged and the other is rejected).

Similar to the analysis in Section 5.2, we add one level metrics at a time and build a model instead of building one model with all metrics at once. As a result, we compare the fit of three models: a) *Model 1*, which includes only the submitter-level variables, b) *Model 2*, which adds patch-level variables, and c) *Model 3*, which adds review-level variables. The variable transformation and multicollinearity control in the model construction process is similar to those in Section 5.2. Specifically, we remove three predictors (*prev_pullreqs_proj*, *prev_prs_acc_proj*, and *first_pr*) due to multicollinearity, which leaves us with 20 predictors, as shown in Table 13.

6.2.3 Analysis Results

Overall, as shown in Table 14, both Model 2 and Model 3 have achieved remarkable performance given their high value of AUC [60]. Since Model 3 performs better than Model 2 (AUC: 0.890 versus 0.856) and they have obtained consistent variable effects, we discuss their effects based on Model 3.

As for the submitter-level predictors, only the predictor *prev_prs_acc* has a significant, positive effect. This means that duplicate pull requests submitted by experienced developers whose previous pull requests have a higher acceptance rate are more likely to be accepted. This is inline with the prior findings about general pull request evaluation [42], [47], [52], [116]. Perhaps surprisingly, the predictors relating to developers' standing (*core_team* and *followers*) and their social connection to the project (*social_strength*) are not significant by controlling for other confounds. Prior studies have shown that pull requests from the developers holding higher standing and having stronger social connection with the project have higher acceptance rates [93], [108]. However, our model does not achieve significant effects. Except for the bias on our dataset, we present an assumption that in the context of making a choice between duplicates, integrators' decision does not differentiate based on the identity of the submitter in order to ensure fairness within the community [19], [46]. This assumption deserves a further investigation.

For patch-level metrics, *early_arrival* is highly significant in the model. As expected, duplicate pull requests submitted earlier have a higher likelihood of being accepted. We can also observe that the predictor *loc* has a positive effect, which indicates that duplicate pull requests changing more LOCs are more likely to be accepted. This finding is opposite with the results in previous studies [93], [101] that large pull requests are less likely to be accepted. For a pair of duplicate pull requests, the large one might provide a more thorough solution or fix additional related issues compared to the small one, which increases the probability of acceptance under the same conditions. For example, there is a typical comment left by the integrator: "Since the PR also contains some test cleanup, I'll merge that instead and close this one. But thank you for the efforts, it is much appreciated!". In addition to *loc*, the predictor *test_inclusion* presents a significant, positive effect, which is similar to the effect on pull request acceptance in general [42], [93], [108].

Finally, we discuss the review-level metrics. The predictor *comments_inline* has a significant, positive effect. This indicates that duplicate pull requests receiving more inline comments have a higher chance of being accepted. Nevertheless, prior study [93] showed that pull requests with a high amount of discussion are less likely to be accepted. We argue that duplicate pull requests receiving more comments, especially inline comments, might mean that they have been reviewed and discussed more thoroughly than their counterparts. It requires less effort to be spent on the follow-up review if integrators choose the highly discussed duplicates. We notice that the predictor *revisions* has a positive effect. This indicates that duplicate pull requests revised more times are more likely to be accepted, which agrees with what was already found in prior study [47]. As for comment sentiment, unsurprisingly, duplicate pull requests receiving more positive comments than negative comments are more likely to be accepted. In terms of DevOps checking, as expected, the predictor *CI* has a strong, positive effect when its value is success or pending compared to when the value is failure. This means that duplicate pull requests have lower likelihood of being accepted if the CI test result is failure. Our result confirms the finding in the previous study [98], [108] that *CI* plays

TABLE 14
Statistical Models for the Acceptance of Duplicate Pull Requests

	Model 1			Model 2			Model 3		
	response: <i>accept</i> = 1			response: <i>accept</i> = 1			response: <i>accept</i> = 1		
	Coeffs.	Errors	Signif.	Coeffs.	Errors	Signif.	Coeffs.	Errors	Signif.
log(prev_pullreqs + 0.5)	-0.078	-1.832		-0.097	-1.987	*	-0.091	-1.738	
log(prev_prs_acc + 0.5)	1.597	7.884	***	1.729	7.363	***	1.644	6.617	***
first_pr_proj TRUE	-0.206	-1.270		-0.169	-0.911		-0.173	-0.868	
log(followers + 0.5)	0.061	1.674		0.062	1.477		0.064	1.415	
core_team TRUE	0.251	1.786		0.184	1.111		0.237	1.312	
log(social_strength + 0.5)	0.383	2.590	**	0.217	1.295		0.283	1.538	
early_arrival TRUE	-	-	-	0.534	6.865	***	0.483	5.572	***
log(loc + 0.5)	-	-	-	0.462	5.701	***	0.386	4.477	***
log(files_changed + 0.5)	-	-	-	0.388	2.341	*	0.200	1.135	
test_inclusion TRUE	-	-	-	0.239	1.034		0.203	2.805	*
issue_tag TRUE	-	-	-	0.255	1.855		0.261	1.734	
log(text_len + 0.5)	-	-	-	0.152	2.253	*	0.133	1.877	
log(revisions + 0.5)	-	-	-	-	-	-	0.275	2.563	*
log(comments + 0.5)	-	-	-	-	-	-	0.036	0.406	
log(comments_inline + 0.5)	-	-	-	-	-	-	0.208	2.138	*
comments_pos > comments_neg TRUE	-	-	-	-	-	-	0.460	3.315	***
CI Success	-	-	-	-	-	-	0.728	3.574	***
CI Pending	-	-	-	-	-	-	1.175	5.226	***
CLA Success	-	-	-	-	-	-	1.596	0.532	
CLA Pending	-	-	-	-	-	-	2.948	0.984	
CR Success	-	-	-	-	-	-	1.4e+01	0.022	
CR Pending	-	-	-	-	-	-	1.4e+01	0.022	
Akaike's Information Criterion (AIC):	1411.52			1116.52			1014.19		
Bayesian's Information Criteria (BIC):	1436.39			1171.23			1118.64		
Area Under the ROC Curve (AUC):	0.704			0.856			0.890		

*** $p < 0.001$, ** $p < 0.01$, * $p < 0.05$.

a key role in the process of pull request evaluation, even in the context of duplication. We do not achieve any result for other two DevOps tools (i.e., CLA and CR), probably due to the imbalanced data (i.e., most of pull requests are not checked by these tools). We plan to conduct further analysis focusing on these two kinds of tools in future work.

6.3 Manual Inspection

The regression analysis examines the correlation between several factors and integrators' choice between duplicate pull requests. It reveals what kind of duplicates are more or less likely to be accepted. We further investigate the exact reasons why integrators accept a duplicate pull request rather than its counterpart. This can also examine and verify the results of the regression analysis. To this end, we analyze the review comments of duplicate pull requests and perform a card sort [82] to gain insight into the common themes around integrators' choice between duplicates. The following sections present the card sorting process and the identified reasons for integrators' choice.

6.3.1 Card Sorting

The card sorting analysis is conducted on 150 randomly selected duplicate pairs. This sample yields a 90 percent confidence level with a 6.28 percent error margin. This process includes the following three steps.

Card Preparation. For each randomly selected duplicate pair, we read the dialogue and select all the comments

expressing integrators' choice preference between duplicates. The selected comments are then recorded in a card.

Pair Execution. For each card, two authors read the text and sort the card into an existing category. If the authors believe that the card does not belong to any of the existing categories, they create a new category for that card. The created category is labeled with a descriptive title to indicate its theme. For a card citing multiple themes, more copies of the card are created for each cited theme. When the two authors disagree about the category of a specific card, they invite the other authors to discuss the discrepancy and vote on the card.

Final Analysis. After all cards have been sorted, the two authors review each of the cards again to ensure the integrity of the emerged categories and resolve potential inclusive or redundant relations among categories. All categories are then further grouped into higher-level categories. Finally, to reduce the bias from the two authors, all authors of the paper review and agree on the final taxonomy of categories.

6.3.2 Reasons for Integrators' Choice

As shown in Table 15, we find 8 reasons for integrators' choice between duplicates, which can be classified into two categories, i.e., *technical assessment* and *non-technical consideration*. Note that the total frequency is greater than 150, because more than one reason might be cited in a decision-making discussion. In the following, we discuss each of reasons supported by examples.

TABLE 15
The Taxonomy of the Reasons for Integrators' Choice Between Duplicates

	Reason	Frequency
Technical assessment	Higher quality	20
	Correctness	11
	Broader coverage	7
	Pushed to proper branch	4
	Inclusion of test code	3
Non-technical consideration	First-come, First-served	30
	Active response	3
	Encouraging newcomers	1
	No explanation	76

A) Technical Assessment.

Higher Quality. If both of two duplicates have provided correct implementation, integrators are inclined to choose the one of higher quality. High quality has multiple manifestations in our analysis, such as less affected files (e.g., “*After a short discussion internally, we are going to close this one in favour of merging #7666. This is nothing more than the fact that the other PR touches much less files*”), and more outstanding performance (e.g., “*Closing this in favor of #10373, which also contains performance improvements*”). Given that higher-quality implementation tends to receive more positive review comments, this choice preference is consistent with the regression analysis result that duplicate pull requests receiving more positive comments are more likely to be accepted.

Correctness. The acceptance of duplicate pull requests fundamentally depends on the correctness of its implementation, as described by an integrator “*Looks like #2716 fixed this though the quoting on that one is not 100 percent right and I prefer your solution... Would you like to rebase yours on top of head to use the ‘+=(...’ operator?*”. Since a failed status check indicates problematic implementation, this preference supports the finding in the regression analysis that duplicate pull requests are less likely to be accepted if the status is failure.

Broader Coverage. Two pull requests might be partial duplicates in which one is the subset of the other one, as a reviewer pointed “*This PR is a subset of PR #16031*”. In such case, we observe that integrators prefer to choose the large one that has covered more issues. Considering that the broader coverage usually leads to more changed LOCs, this choice preference is reflected in the regression analysis that duplicate pull requests changing more LOCs are more likely to be accepted.

Pushed to the Proper Branch. It is common for OSS projects to follow a certain strategy on branch management. For example, the maintenance branch in many projects accepts only bug fixes, and new feature proposals are not accepted. Therefore, integrators accept duplicate pull requests pushed to the proper branch (e.g., “*closing this PR as a duplicate of #5193 (the latter is sent against 2.0 which is the branch to target for bugfix)*”).

Inclusion of Test Code. Most OSS project require contributors to provide necessary test code for their modifications. Therefore, the existence of test code in the duplicate pull requests could help win some favor from integrators (e.g., “*Thanks for*

the fix. Not sure we want to merge this or wait for #3907 which also fixes it and adds a regression test for all estimators”). This agrees with the result in the regression analysis.

B) Non-Technical Consideration.

First-Come, First-Served. Integrators may follow the “first-come, first-served” rule and accept duplicates arriving earlier than those arriving later (e.g., “*Actually, I see that #28026 is at the head of the merge queue right now, so it will probably merge before this one, in which case we can just discard this PR*” and “*Thank you very much for opening the pull request @xxx. Sadly I have to close it nonetheless because someone already opened one with the same changes before: #5761*”). This is consistent with the finding in the regression analysis that duplicates submitted earlier have a higher possibility of being accepted.

Active Response. In the review process of a pull request, contributors are usually requested to update their pull requests until integrators are satisfied. Consequently, if a duplicate pull request has not been actively updated by its submitter, integrators might turn to its counterpart from which integrators have received active responses (e.g., “*@xxx The original author of that PR hasn't responded in a week. If you want to fix the tests in your PR, we could merge this one. Your call*”).

Encouraging Newcomers. New contributors act as an innovation source for new ideas and are essential to the survival of OSS projects [84]. Integrators have always tried to retain newcomers and hope that they become long-term contributors [112]. Consequently, duplicate pull requests submitted by newcomers might be accepted by integrators to encourage newcomers to make more contributions (e.g., “*@xxx do you mind if we close this in favor of #12004 from a new contributor?*”). However, we find very few instances of this choice preference. This is in line with the fact that the factors `first_pr_proj` and `core_team` show no significant effects in the regression model. This finding can indicate that when making a choice between duplicates, integrators care more about the pull requests than the role of their submitters.

C) No Explanation.

Integrators may make a choice without leaving any further explicit explanation (e.g., “*Replaced by #5067*” and “*closing in favour of #10582*”). For these cases, we examine the submission time of the involved duplicate pull requests. We find that most accepted pull requests are those that are submitted earlier. Therefore, a possible explanation for why integrators offer no explicit explanation is that they think the “first-come, first-served” rule is the default standard in duplication choice. In addition, integrators and contributors might have discussed the matter on other communication media or face-to-face outside GitHub, and as a result, integrators simply make a choice without more explanation.

6.3.3 Complementary Investigation: Beyond Rejection

In examining integrators' comments, we also find that, sometimes, making a decision between duplicates is more complex than simple acceptance and rejection. Even though integrators give their preference to one duplicate pull request, they might not directly and completely abandon the other one. In the following, we present three instructive scenarios where integrators make a decision beyond simple close in dealing with the rejected duplicates.

Supplementary. Although it has been decided to accept one duplicate pull request instead of another one, some integrators would examine what could be reused from the rejected one to enhance the accepted one. For example, although *rails/rails/#20697* was preferred over *rails/rails/#20050*, the author of *rails/rails/#20697* was asked to reuse the test code from *rails/rails/#20050* (“*I prefer this behavior (and its simpler implementation) Can you please cherry-pick the test from #20050?*”). In such cases, integrators usually also gave credits to the author of the rejected pull request for the incorporated code (“*along with adding an changelog entry listing both authors*” and “*Can you credit him in the commit as well after you squash and rebase?*”).

Reopen and Backup. Some rejected duplicate pull requests were later reopened as useful backups, since their counterparts that were previously preferred could not be merged finally. For example, *facebook/react/#5236* was closed at first because there was already a duplicate pull request *facebook/react/#5220*. But later, *facebook/react/#5236* was reopened and merged due to the inactivity of *facebook/react/#5220*, as the integrator said: “*Other PR didn’t get updated today and the failures were annoying on new PRs*”.

Collaboration. Integrators might ask the author of the rejected duplicate pull request to improve the preferred one together (e.g., “*Could you jump on #3082 and help us review it?*” and “*Please work with @xxx to test his code once he fixes the merge conflicts*”). We also found that some of the authors were willing to offer their help (e.g., “*Made a minor note on the other PR to include one of the improvements*” and “*Looks like they took my advice and used proc_terminate() so another package was not needed. Thanks for catching duplicate, but at least it was not a waste. :)*”).

RQ3: Pull requests with accurate and high-quality implementation, broad coverage, necessary test code, high maturity, and deep discussion are more likely to be accepted. However, integrators also make a choice based on non-technical considerations, e.g., they may accept pull requests to respect the arrival order and active response. For the rejected duplicates, integrators might try to maximize their value, e.g., cherry-picking the useful code.

7 DISCUSSION

Based on our analysis results and findings, we now provide additional discussion and propose recommendations and implications for OSS practitioners.

7.1 Main Findings

7.1.1 Awareness Breakdown

Maintaining awareness in global distributed development is a significant concern [43], [92]. Developers need to pursue “*an understanding of the activities of others, which provides a context for your own activity*” [38]. As the most popular collaborative development platform, GitHub has centralized information about project statuses and developer activities and made them transparent and visible [37], which helps developers to maintain awareness with less effort [48]. However, awareness breakdown still occurs and results in duplicate work. Our

findings about the specific contexts where duplicates are produced, as shown in Section 5.1, highlight three mismatches leading to awareness breakdown.

A Mismatch Between Awareness Requirements and Actual Activities. In most community-based OSS projects, developers are self-organized [26], [34], and are allowed to work on any part of the code according to individual’s interest and time [43], [54]. Awareness requirements arise as a response to developers’ free and spontaneous activities. Whenever a developer decides to get engaged in a task, s/he should ensure that no other developers have worked on the same task. However, our findings show that some contributors lack sufficient effort investment in awareness activities (Section 5.1: *Not searching for existing work*, *Overlooking linked pull requests*, and *Missing existing claim*). We assume that this is due to the volunteer nature of OSS participation. For some developers, especially the casual contributors and one time contributors, a major motivation to make contributions is to “*scratch their own itch*” [56], [69]. When they encounter a problem, they code a patch to fix it and send the patch back to the community. Some of them even do not care about the final outcome of their pull requests [85]. It might be harder to get them to spend more time to maintain awareness of other developers. Automatic awareness tools can mitigate this problem. Prior research has proposed to automatically detect duplicates at pull request submission [57], [73] and identify ongoing features from forks [113]. Furthermore, we advocate for future research on seamlessly integrating awareness tools to developers’ development environment and designing intelligent and non-intrusive notification mechanism.

A Mismatch Between Awareness Mechanisms and Actual Demands. Currently, GitHub provides developers with a wide range of mechanisms, e.g., following developers and watching projects [37], to maintain a general awareness about project status. However, developers can be overwhelmed with a large-scale of incoming events in popular projects (Section 5.1: *Missing notifications*). It is also impractical for developers to always maintain overall awareness of a project due to multitasking [95] and turnover [58]. Usually, developers need to obtain on-demand awareness around a specific task whenever deciding to submit a pull request, i.e., gathering task-centric information to figure out people interested in the same task. Currently, the main mechanisms to meet this demand are querying issue and pull request list and reading through the discussion history. As mentioned in Section 5.1, the support by these mechanisms is not as adequate as expected due to information mixture (Section 5.1: *Overlooking linked pull requests* and *Missing existing claims*) and other technical problems (Section 5.1: *Disappointing search functionality* and *Diversity of natural language usages*). Awareness mechanisms would be most useful if they can fulfil developers’ actual demands in maintaining awareness.

A Mismatch Between Awareness Maintenance and Actual Information Exchange. Maintaining awareness is bidirectional. Intuitively, it means that developers need to gather external information to stay aware of others’ activities, with the hope that *I do not duplicate others’ work*. But from a global perspective, it also means developers should actively share their personal information that can be gathered by others, with the hope that *others do not duplicate my work*. Our

findings show that some developers do not timely announce their plans (Section 5.1: *Implementing without claiming first*) and link their work to the associated issue (Section 5.1: *Lack of links*). This hinders other developers' ability to gather adequate contextual information. Although prior work [18], [29], [92] has extensively studied on how to help developers track work and get information, more research attention should be paid to encouraging developers to share information. For example, it would be interesting to investigate whether developers' willingness to share information is affected by the characteristics of collaboration mechanisms and communication tools.

Obviously, awareness tools are important for OSS developers to stay aware of each other. However, no tool or mechanism can prevent all awareness breakdown entirely. A better understanding of the importance of group awareness and better use of available technologies can help developers ensure that their individual contributions do not cause accidental duplicates.

7.1.2 The Paradox of Duplicates

Generally speaking, both project integrators and contributors hope to prevent duplicate pull requests, because duplicates can waste their time and effort, as shown in Section 4. This is also reflected in their comments, e.g., “*it probably makes sens to just center around a single effort*”, “*No need to do the same thing in two PRs*”, and “*Oops! Sorry, did not mean to double up*”. However, when duplicates are already produced, potential value might be mined from them as shown in Section 6.3.3. From our findings, we notice two interesting paradoxes of duplicates.

Redundancy versus Alternative. In many cases, duplicate pull requests change pretty much the same code, which only bring unnecessary redundancy. While in some cases, duplicates implemented in different approaches provide alternative solutions, as a developer put it: “*The pull requests are different, so maybe it is good there are two*”. In such cases, project integrators have a higher chance to accept a better patch. However, this comes at a price. Integrators have to invest more time to compare the details of duplicates in order to clearly disclose the difference between them. Ensuring that their effort is not wasted in coping with duplicates, but maximizing the disclosure and adoption of additional value provided by each duplicate, is a trade-off integrators should be aware of.

Competition versus Collaboration. At first sight, authors of duplicate pull requests face a competition in getting their own patches accepted. For example, one contributor tried to persuade integrators to choose his pull request rather than another one: “*Pick me! Pick me! I was first! :)*”. Nevertheless, we found some cases where the authors of duplicates worked together towards a better patch by means of mutual assessment and negotiated incorporation, as shown in Section 6.3.3. According to developers, the collaboration is also an opportunity for both the authors to learn from each other's strength (“*I looked at some awesome code that @xxx wrote to fix this issue and it was so simple, I just did not fully understand the issue I was fixing*”). Standing for their own patches, but seeking for collaboration and learning, is a trade-off the authors of duplicates should be aware of.

7.1.3 Decision-Making in Either/Or Contexts

In the general context of pull request evaluation, integrators are answering a Yes/No question “*whether to accept this pull request?*”, and the considered factors are mainly relating to individual pull requests. While in the context of making a choice between duplicates, integrators are answering an Either/Or question “*whether to choose this duplicate pull request or the other one?*”, and integrators would evaluate the duplicates from a comparative perspective. Based on our findings about integrators' preference between duplicates, as presented in Section 6, we infer two prominent characteristics of integrators' decisions in the Either/Or contexts.

Choose the Patch, not the Author. We find that when making decisions between duplicates, integrators have a preference to consider patch-level metrics (e.g., arrival order) and review-level metrics (e.g., CI test results) instead of submitter-level metrics (e.g., the submitter's identity). Compared to submitter-level metrics, patch-level and review-level metrics are more objective evidence. While the submitter's identity can be used to make inferences about the quality and trustworthiness of a pull request [42], [93], decisions made on the basis of objective evidence might look more fair and rational in the context of selection between duplicates. The benefits of such decision strategy include ensuring the fairness within the communities [19], [46] and eliminating integrators' pressure of explaining the rejection of duplicates [42].

Invested Effort Matters. We find that integrators prefer duplicate pull requests that have been highly discussed and revised a couple of times. While a higher number of comments and revisions might indicate that a pull request was not perfect and integrators have requested changes to update it, it also reflects that the pull request has been thoroughly reviewed and improved, and both integrators and the submitter have invested considerable effort. The invested effort on one duplicate pull request cannot be “transferred” to its counterpart because duplicate pull requests might have different implementation details and each of them has to be carefully reviewed. Given the facts that time is the top challenge faced by integrators [42] and that asking for more work from contributors to improve their code might be difficult [42], [85], choosing the thoroughly discussed and revised duplicate pull requests might be a cost-efficient and safe decision.

7.2 Suggestions for Contributors

To avoid unintentional duplicate pull requests, contributors may follow a set of best contributing practices when they are involved in the pull-based development model.

Adequate Checking. Many duplicates were produced because contributors did not conduct adequate checking to make sure that no one else was working on the same thing (Section 5.1: *Not searching for existing work*, *Overlooking linked pull requests*, and *Missing existing claims*). We recommend that contributors should perform at least three kinds of checking before starting their work: i) reading through the whole discussion of an issue and checking whether anyone has claimed the issue; ii) examining each of the pull requests linked to an issue and checking whether any of them is an ongoing work to solve the issue; and iii) performing searches with different keywords against open and closed

pull requests and issues, and carefully checking where similar work already exists.

Timely Completion. Quite a number of OSS developers contribute to a project at their spare time, and some of them even switch between multiple tasks. As a result, it might be difficult for them to complete an individual task in a timely fashion. However, we still suggest that contributors should quickly accomplish each work in proper order, e.g., one item at a time, to shorten their local duration. This can make their work publicly visible earlier, which can, to some extent, prevent others from submitting duplicates (Section 5.1: *Overlong local work*).

Precise Context. Providing complete and clear textual information for submitted pull requests is helpful for other contributors to retrieve these pull requests and acquire an accurate and comprehensive understanding of them (Section 5.1: *Diversity of natural language usage*). In addition, if a pull request is solving a tracked issue, adding the issue reference in the pull request description, e.g., “fix #[issue_number]”, can avoid some duplicates because of the increased degree of awareness (Section 5.1: *Lack of links*).

Early Declaration. Zhou *et al.* [114] already suggested that claiming an issue upfront is associated with a lower chance of redundant work. In our study, we find several actual instances of duplicates where integrators clearly pointed out the contributors should claim the issues first and then implement the patches (e.g., “@xxx, btw, it is a good idea to comment on an issue when you start working on it, so we can coordinate better and avoid duplication of effort”). We would like to emphasize again the importance of early declaration which should become a best practice developers can follow in OSS collaborative development. Compared with late report, early declaration can timely broadcast contributors’ intention to the community to get the attention of interested parties, so that they can avoid some accidental duplicate work (Section 5.1: *Implementing without claiming first*).

Argue for Their Patches. As shown in Section 6, various factors can be examined when integrators make decisions between duplicates. The authors of duplicates should actively argue for their own pull requests by explicitly stating the strength of their patches, especially if they have proposed a different approach and provided additional benefits. They can also review each of other’s patch and discuss the difference before waiting for an official statement from integrators. This can provide a solid basis for integrators to make informed decisions about which duplicate should be accepted. Moreover, if the value of a duplicate pull request has been explicitly stated, even it is finally closed, its useful part has a higher chance to be noticed and cherry-picked by integrators, as shown in Section 6.3.3.

7.3 Suggestions for Core Team

The core team of an OSS project, acting as the integrator and maintainer of the project, is responsible for establishing contribution standards and coordinating contributors’ development. To achieve the long-term and continuous survival of the project, the core team may also follow some best practices.

Evident Guidelines. Although most projects have warned contributors not to submit duplicate issues and pull requests, the advice are usually too general. We suggest projects to make the advice more visible, specific, and easy-follow. For

example, projects can use a section to list the typical contexts where duplicates occur, as presented in Section 5.1, and itemize the specific actions should be taken to avoid duplicates, as we have suggested for contributors (Section 7.2: *Adequate checking*).

Explaining Decisions. Integrators must make a choice between duplicate pull requests, which means that they have to reject someone. For contributors whose pull requests have been rejected, they might be pleased to get feedback and explanation about why their work has been rejected rather than simply closing their pull requests. However, we observed nearly 50 percent of our qualitative samples where decisions were made without any explanation (as shown in Table 15). Even worse, we identified that the rough explanation (e.g., “*Thanks for your PR but this fix is already merged in #20610*”) would be likely to make the contributor upset (“*already implies I submitted my PR later than that, rather than nearly a year earlier ;)* But at least it’s fixed”). In that case, the integrator had to give an additional apology (“*sorry, sometimes a PR falls in the cracks and a newer one gets the attention. We have improved the process in hopes to avoid this but we still have a big backlog in which these things are present*”) to mitigate the negative effect. In the future, a careful analysis should be designed to examine the effectiveness of this suggestion based on controlled experiments.

7.4 Suggestions for Design of Platforms

Online collaborating platforms such as GitHub have designed and provided numerous mechanisms and tools to support OSS development. However, the practical problem of duplicate contributions proves that the platforms need to be improved.

Claim Button. In order to make it more efficient for developers to maintain awareness of each other, we envision a new mechanism called *Claim* which is described as follows. For a GitHub issue, each interested developer can click the *Claim* button on the issue page to claim that s/he is going to work on the issue. The usernames of all claimers are listed together below the *Claim* button. Every time the *Claim* button is clicked, an empty pull request is automatically generated and linked to the claimer’s username in the issue claimer list. Moreover, claimers have a chance to report their plans about how to fix the issue in the input box displayed when the *Claim* button is clicked. The reported plans would be used to describe the empty pull request. Subsequently, claimers perform updates of the empty pull request until they produce a complete patch. All important updates on the empty pull request, e.g., new commits pushed, would be displayed in the claimer list. On the one hand, this mechanism makes it more convenient for developers to share their intentions and activities through just clicking a button. On the other hand, developers can efficiently catch and track other developers’ intentions and activities by simply checking the issue claimer list.

Duplicate Detection. As contributors complained, e.g., “... I wish there has been some automated method to detect pending PR per file basis. This could save lot of work duplicacy ...”, or “It’s strange that GitHub isn’t complaining about this, because it’s an exact dup of #5131 which was merged already”, an automatic detection tool of duplicates is missing in GitHub. Such a tool can help integrators detect duplicates in a timely manner

and prevent them spending resources on the redundant effort of evaluating duplicates separately. Therefore, GitHub can learn from Stack Overflow and Bugzilla to recommend similar work when developers are creating pull requests by utilizing various similarity measures, e.g., title and code changes. The features discussed in Section 5.2.1 can also be integrated to enhance the recommendation system.

Reference Reminder. Since developers might overlook linked pull requests to issues (Section 5.1: *Overlooking linked pull requests*), platforms can actively remind developers of existing pull requests linked to the same issue at pull request submission time. The goal of this functionality is similar to that of the duplicate detection tool. However, it can be implemented in a more straightforward way. For example, whenever developers add an issue reference in filling a pull request, a pop-up box can be displayed next to the issue reference to list the existing pull requests linked to that issue.

Duplicate Comparison. As discussed in Section 6, when integrators make a choice between duplicate pull requests, they consider several factors. Platforms can support duplicate comparison to make the selection process more efficient. For example, platforms can automatically extract several features of compared duplicates, e.g., inclusion of test codes and the contributor's experience, and display these features in a comparison format to clearly show the difference between duplicate pull requests and speed up the selection process.

Online Incorporation. As presented in Section 6.3.3, integrators sometimes prefer to incorporate one duplicate pull request into the other one to promote patch thoroughness. Currently, the typical way to incorporate a pull request PR_i into another pull request PR_j is as follows: i) adding the head branch of PR_i as a remote branch in the corresponding local repository of PR_j , ii) fetching the remote branch to the local repository, iii) cherry-picking the needed commits from or rebase onto the remote branch, and iv) updating PR_j by synchronizing the changes from local repository to the head branch of PR_j . Developers might also need to update the commit message or project changelog to give credit for the incorporated code. The whole incorporation process can be too complex for newcomers to undertake. Moreover, this process seems to be tedious for incorporating trivial changes. GitHub can support online incorporation of duplicate pull requests. For example, it can allow developers to pick the needed code by clicking buttons in the UI, and the credit is given to the picked code by automatically updating the commit message and changelog.

8 THREATS TO VALIDITY

In this section, we discuss threats to construct validity, internal validity and external validity, which may affect the results of our study.

Construct Validity. In the definition of sequential relationships between two duplicates, two time points are critical, i.e., $T\text{-Creation}$ and $T\text{-Evaluation}$, which stand for the starting times of local work and online evaluation, respectively. In the paper, we set $T\text{-Evaluation}$ as the submission time of pull request, in accordance with its definition. Nevertheless, we cannot obtain the exact value of $T\text{-Creation}$ because there

is no information recording when a contributor starts local work. We set $T\text{-Creation}$ as the creation time of the first commit contained in a pull request. As a result, the observed value of $T\text{-Creation}$ is actually later than its real value because it can be certain that the contributor must first start the local work and then later submit the first commit. It is possible that we introduce some bias to our quantitative study when we set $T\text{-Creation}$ to the creation time of the first commit. For duplicate pairs of overlapping or inclusive relations, the bias does not matter much because they already intersect with each other. However, the bias for duplicate pairs of exclusive relations needs careful attention since inaccurate value of $T\text{-Creation}_{dup}$ may affect whether the relation is exclusive. Indeed, in our quantitative study of the exclusive interval (Table 8), we find that the majority of duplicate pairs of exclusive relations have relatively long intervals, which means that a minor shift in the value of $T\text{-Creation}_{dup}$ is unlikely to affect the original relation. Therefore, setting $T\text{-Creation}$ as the creation time of the first commit is acceptable in practice.

Internal Validity. In the manual analysis of integrators' choice between duplicate pull requests, we target a sampled subset of duplicate pull requests. It is possible that we have missed some other cases that are not in the sampled subset. However, the items in the subset are randomly selected, and the sample size is of high confidence level, as described in Section 6.3. Therefore, missed cases (if any), accounting for a very small proportion of the whole, would not significantly change our findings.

External Validity. The threat to external validity relates to the generalizability of our findings. To mitigate this threat, our empirical study was conducted on 26 projects hosted on GitHub, covering a diversity of programming languages and application domains. However, it is still a small sample given that 100 million repositories [1] have been hosted on GitHub, let alone there are other social coding sites such as GitLab and BitBucket. In the future, we plan to extend our study by including more projects from jointCloud [99] development platforms.

9 CONCLUSION

In this study, we investigated the problem of duplicate contributions in the context of pull-based distributed development. The goal of our study is to better understand the influences of duplicate pull requests during collaborative development, the context in which duplicate pull requests occur, and the alternative preference of integrator between duplicate pull requests. We conducted an empirical study on 26 GitHub projects to achieve the goal. We found that duplicate pull requests slow down the review process and require more reviewers for extended discussions. We observed that the inappropriateness of OSS contributors' work patterns (e.g., not checking for existing work) and the shortcomings of their collaboration environment (e.g., unawareness of parallel work) would result in duplicates. We also observed that duplicate pull requests are significantly different from non-duplicate pull requests in terms of project-level characteristics (e.g., area hotness and number of active core team members), submitter-level characteristics (e.g., experience and social connection to project), and

patch-level characteristics (e.g., change type and issue visibility). We found that duplicate pull requests with accurate and high-quality implementation, broad coverage, necessary test codes, high maturity, and deep discussion, are more likely to be accepted. We also found that integrators might make a choice based on non-technical considerations, e.g., they may accept pull requests to respect arrival order and active response.

Based on the findings we recommend that OSS contributors should always perform sufficient verification against existing work before they start working on a task. Contributors are expected to declare their intentions as soon as possible and prepare their work with complete related information to make their work highly visible early on. Integrators should provide contributors with visible and detailed guidelines on how to avoid duplicated work. Social coding platforms are expected to enhance the awareness mechanisms in order to make it more effective and efficient for developers to stay aware of each other. It is also meaningful to provide practical service and tools to support automatic identification of duplicates, visualized comparison between duplicates, etc.

Last but not least, our findings point to several future research directions. Researchers can design awareness tools to increase developers' awareness of others' activities. Such tools not only help prevent duplicate effort on the same tasks but also have the potential functionality to link related contributors for better coordination. Moreover, we think it is meaningful to investigate how integrators' practices in managing the contributors' conflicts affect contributors' continuous participation.

ACKNOWLEDGMENTS

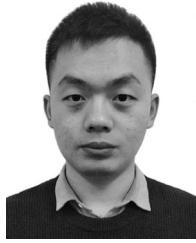
This work was supported by National Grand R&D Plan (Grant No. 2018AAA0102304) and the National Natural Science Foundation of China (Grant No. 61702534).

REFERENCES

- [1] About GitHub, 2019. Accessed: Oct. 19, 2019. [Online]. Available: <https://github.com/about>
- [2] About GitHub issue, 2019. Accessed: Oct. 19, 2019. [Online]. Available: <http://help.github.com/en/articles/about-issues>
- [3] Bugzilla, 2019. Accessed: Oct. 19, 2019. [Online]. Available: <http://www.bugzilla.org>
- [4] Code climate, 2019. Accessed: Oct. 19, 2019. [Online]. Available: <https://codeclimate.com>
- [5] Contributing to scikit-learn, 2020. Accessed: Jun. 11, 2020. [Online]. Available: <https://scikit-learn.org/dev/developers/contributing.html>
- [6] Contributing to the ansible documentation, 2020. Accessed: Jun. 11, 2020. [Online]. Available: https://docs.ansible.com/ansible-devel/community/documentation_contributions.html#community-documentation-contributions
- [7] The duppr dataset, 2018. Accessed: Oct. 19, 2019. [Online]. Available: <https://github.com/whystar/MSR2018-DupPR>
- [8] Get the combined status for a specific ref, 2020. Accessed: May 13, 2020. [Online]. Available: <https://developer.github.com/v3/repos/statuses/#get-the-combined-status-for-a-specific-ref>
- [9] GitHub, 2019. Accessed: Oct. 19, 2019. [Online]. Available: <http://github.com>
- [10] Gitlab, 2019. Accessed: Oct. 19, 2019. [Online]. Available: <http://gitlab.com>
- [11] Good first issues in the project pandas, 2020. Accessed: Jun. 11, 2020. [Online]. Available: <https://github.com/pandas-dev/pandas/labels/good%20first%20issue>
- [12] How should duplicate questions be handled? 2009. Accessed: Oct. 19, 2019. [Online]. Available: <https://meta.stackexchange.com/questions/10841/how-should-duplicate-questions-be-handled>
- [13] Stack overflow, 2019. Accessed: Oct. 19, 2019. [Online]. Available: <http://stackoverflow.com>
- [14] Travis-ci, 2019. Accessed: Oct. 19, 2019. [Online]. Available: <https://www.travis-ci.org>
- [15] Understanding the GitHub flow, 2020. Accessed: Aug. 8, 2020. [Online]. Available: <https://guides.github.com/introduction/flow/>
- [16] B. Adams, B. Adams, B. Adams, and M. Ortú, "Do developers feel emotions? An exploratory analysis of emotions in software artifacts," in *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 262–271.
- [17] M. Ahsanuzzaman, M. Asaduzzaman, C. K. Roy, and K. A. Schneider, "Mining duplicate questions in stack overflow," in *Proc. 13th Int. Conf. Mining Softw. Repositories*, 2016, pp. 402–412.
- [18] R. Arora, S. Goel, and R. K. Mittal, "Supporting collaborative software development over GitHub," *Softw.: Pract. Experience*, vol. 47, no. 10, pp. 1393–1416, 2017.
- [19] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "The secret life of patches: A firefox case study," in *Proc. 19th Work. Conf. Reverse Eng.*, 2012, pp. 447–455.
- [20] O. Baysal, O. Kononenko, R. Holmes, and M. W. Godfrey, "Investigating technical and non-technical factors influencing modern code review," *Empir. Softw. Eng.*, vol. 21, no. 3, pp. 1–28, 2015.
- [21] A. Begel, J. Bosch, and M. A. Storey, "Social networking meets software development: Perspectives from GitHub, MSDN, stack exchange, and TopCoder," *IEEE Softw.*, vol. 30, no. 1, pp. 52–66, Jan./Feb. 2013.
- [22] M. Beller, G. Gousios, and A. Zaidman, "Oops, my tests broke the build: An analysis of travis CI builds with GitHub," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 356–367.
- [23] Y. Benjamini and Y. Hochberg, "Controlling the false discovery rate: A practical and powerful approach to multiple testing," *J. Roy. Stat. Soc.*, vol. 57, no. 1, pp. 289–300, 1995.
- [24] N. Bettenburg, R. Premraj, T. Zimmermann, and S. Kim, "Duplicate bug reports considered harmful... really?" in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2008, pp. 337–345.
- [25] C. Bird, A. Gourley, and P. Devanbu, "Detecting patch submission and acceptance in OSS projects," in *Proc. 4th Int. Workshop Mining Softw. Repositories*, 2007, p. 26.
- [26] C. Bird, D. Pattison, R. D'Souza, V. Filkov, and P. Devanbu, "Latent social structure in open source projects," in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2008, pp. 24–35.
- [27] H. Borges, A. Hora, and M. T. Valente, "Understanding the factors that impact the popularity of GitHub repositories," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2016, pp. 334–344.
- [28] A. Bosu, J. C. Carver, C. Bird, J. Orbeck, and C. Chockley, "Process aspects and social dynamics of contemporary code review: Insights from open source development as well as industrial practice at microsoft," *IEEE Trans. Softw. Eng.*, vol. 43, no. 1, pp. 56–75, Jan. 2017.
- [29] F. Calefato and F. Lanobile, "SocialCDE: A social awareness tool for global software teams," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, pp. 587–590.
- [30] F. Calefato, F. Lanobile, F. Maiorano, and N. Novielli, "Sentiment polarity detection for software development," *Empir. Softw. Eng.*, vol. 23, pp. 1352–1382, 2018.
- [31] S. Chacon and B. Straub, *Pro Git (Second Edition)*. New York, NY, USA: Apress, 2018.
- [32] P. Cohen, S. G. West, and L. S. Aiken, *Applied Multiple Regression/Correlation Analysis for the Behavioral Sciences*. London, U.K.: Psychology Press, 2014.
- [33] M. A. Connolly and K.-Y. Liang, "Conditional logistic regression models for correlated binary data," *Biometrika*, vol. 75, no. 3, pp. 501–506, 1988.
- [34] K. Crowston, K. Wei, Q. Li, U. Y. Eseryel, and J. Howison, "Coordination of free/libre open source software development," in *Proc. Int. Conf. Inf. Syst.*, 2005, pp. 11–23.
- [35] K. Crowston, K. Wei, Q. Li, and J. Howison, "Core and periphery in free/libre and open source software team communications," in *Proc. 39th Annu. Hawaii Int. Conf. Syst. Sci.*, 2006, vol. 6, pp. 118a–118a.
- [36] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Leveraging transparency," *IEEE Softw.*, vol. 30, no. 1, pp. 37–43, Jan./Feb. 2013.

- [37] L. Dabbish, C. Stuart, J. Tsay, and J. Herbsleb, "Social coding in GitHub: Transparency and collaboration in an open software repository," in *Proc. ACM Conf. Comput. Supported Cooperative Work*, 2012, pp. 1277–1286.
- [38] P. Dourish and V. Bellotti, "Awareness and coordination in shared workspaces," in *Proc. ACM Conf. Comput.-Supported Cooperative Work*, 1992, pp. 107–114.
- [39] D. Ford, M. Behroozi, A. Serebrenik, and C. Parnin, "Beyond the code itself: How programmers really look at pull requests," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.: Softw. Eng. Soc.*, 2019, pp. 51–60.
- [40] G. Gousios, M. Pinzger, and A. V. Deursen, "An exploratory study of the pull-based software development model," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 345–355.
- [41] G. Gousios, M.-A. Storey, and A. Bacchelli, "Work practices and challenges in pull-based development: The contributor's perspective," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 285–296.
- [42] G. Gousios, A. Zaidman, M.-A. Storey, and A. V. Deursen, "Work practices and challenges in pull-based development: The integrator's perspective," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 358–368.
- [43] C. Gutwin, R. Penner, and K. Schneider, "Group awareness in distributed software development," in *Proc. ACM Conf. Comput. Supported Cooperative Work*, 2004, pp. 72–81.
- [44] A. E. Hassan, "Predicting faults using the complexity of code changes," in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 78–88.
- [45] A. Hindle, M. W. Godfrey, and R. C. Holt, "Release pattern discovery: A case study of database systems," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2007, pp. 285–294.
- [46] C. Jensen and W. Scacchi, "Role migration and advancement processes in OSSD projects: A comparative case study," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 364–374.
- [47] Y. Jiang, B. Adams, and D. M. German, "Will my patch make it? and how fast?: Case study on the linux kernel," in *Proc. 10th Work. Conf. Mining Softw. Repositories*, 2013, pp. 101–110.
- [48] E. Kalliamvakou, D. Damian, K. Blincoe, L. Singer, and D. M. German, "Open source-style collaborative development practices in commercial projects using GitHub," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, 2015, pp. 574–585.
- [49] Y. Kammerer and P. Gerjets, "The role of search result position and source trustworthiness in the selection of web search results when using a list or a grid interface," *Int. J. Hum.-Comput. Interaction*, vol. 30, no. 3, pp. 177–191, 2014.
- [50] F. Konietzschke, L. A. Hothorn, and E. Brunner, "Rank-based multiple test procedures and simultaneous confidence intervals," *Electron. J. Stat.*, vol. 6, pp. 738–759, 2012.
- [51] O. Kononenko, O. Baysal, and M. W. Godfrey, "Code review quality: How developers see it," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 1028–1038.
- [52] O. Kononenko, T. Rose, O. Baysal, M. Godfrey, D. Theisen, and B. D. Water, "Studying pull request merges: A case study of shopify's active merchant," in *Proc. 40th Int. Conf. Softw. Eng.: Softw. Eng. Pract.*, 2018, pp. 124–133.
- [53] K. R. Lakhani and E. V. Hippel, "How open source software works: "free" user-to-user assistance," in *Produktentwicklung Mit Virtuellen Communities*. Berlin, Germany: Springer, 2004, pp. 303–339.
- [54] K. R. Lakhani and R. G. Wolf, "Why hackers do what they do: Understanding motivation and effort in free/open source software projects," *Perspectives Free Open Source Softw.*, pp. 3–22, 2005.
- [55] A. Lazar, S. Ritchey, and B. Sharif, "Improving the accuracy of duplicate bug report detection using textual similarity measures," in *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 308–311.
- [56] A. Lee, J. C. Carver, and A. Bosu, "Understanding the impressions, motivations, and barriers of one time code contributors to floss projects: A survey," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 187–197.
- [57] Z. Li, G. Yin, Y. Yu, T. Wang, and H. Wang, "Detecting duplicate pull-requests in GitHub," in *Proc. 9th Asia-Pacific Symp. Internetware*, 2017, Art. no. 20.
- [58] B. Lin, G. Robles, and A. Serebrenik, "Developer turnover in global, industrial open source projects: Insights from applying survival analysis," in *Proc. 12th Int. Conf. Global Softw. Eng.*, 2017, pp. 66–75.
- [59] J. D. Long, F. Du, and N. Cliff., *Ordinal Analysis of Behavioral Data*. Hoboken, NJ, USA: Wiley, 2003.
- [60] C. E. Metz, "Basic principles of ROC analysis," *Seminars Nuclear Medicine*, vol. 8, pp. 283–298, 1978.
- [61] G. A. Miller, "WordNet: A lexical database for english," *Commun. ACM*, vol. 38, no. 11, pp. 39–41, 1995.
- [62] Y. Mizobuchi and K. Takayama, "Two improvements to detect duplicates in stack overflow," in *Proc. IEEE 24th Int. Conf. Softw. Anal. Evol. Reengineering*, 2017, pp. 563–564.
- [63] A. Mockus and L. G. Votta, "Identifying reasons for software changes using historic databases," in *Proc. Int. Conf. Softw. Maintenance*, 2000, pp. 120–130.
- [64] K. Nakakoji, Y. Yamamoto, Y. Nishinaka, K. Kishida, and Y. Ye, "Evolution patterns of open-source software systems and communities," in *Proc. Int. Workshop Princ. Softw. Evol.*, 2002, pp. 76–85.
- [65] A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, D. Lo, and C. Sun, "Duplicate bug report detection with a combination of information retrieval and topic modeling," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2012, pp. 70–79.
- [66] N. Novielli, F. Calefato, D. Dongiovanni, D. Girardi, and F. Lanobile, "Can we use se-specific sentiment analysis tools in a cross-platform setting?" in *Proc. 17th Int. Conf. Mining Softw. Repositories*, 2020.
- [67] J. W. Osborne and A. Overbay, "The power of outliers (and why researchers should always check for them)," *Practical Assessment Res. Eval.*, vol. 9, no. 6, pp. 1–12, 2004.
- [68] R. Pham, L. Singer, O. Liskin, F. F. Filho, and K. Schneider, "Creating a shared understanding of testing culture on a social coding site," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 112–121.
- [69] G. Pinto, I. Steinmacher, and M. A. Gerosa, "More common than you think: An in-depth study of casual contributors," in *Proc. 23rd Int. Conf. Softw. Anal. Evol. Reeng.*, 2016, vol. 1, pp. 112–123.
- [70] L. Ponzanelli et al., "Supporting software developers with a holistic recommender system," in *Proc. 38th Int. Conf. Softw. Eng.*, 2017, pp. 94–105.
- [71] M. M. Rahman and C. K. Roy, "An insight into the pull requests of GitHub," in *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 364–367.
- [72] F. Ramsey and D. Schafer, *The Statistical Sleuth: A Course in Methods of Data Analysis*. Boston, MA, USA: Cengage Learning, 2012.
- [73] L. Ren, S. Zhou, C. Kästner, and A. Wasowski, "Identifying redundancies in fork-based development," in *Proc. IEEE 26th Int. Conf. Softw. Anal. Evol. Reeng.*, 2019, pp. 230–241.
- [74] P. C. Rigby and C. Bird, "Convergent contemporary software peer review practices," in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, pp. 202–212.
- [75] P. C. Rigby, D. M. German, L. Cowen, and M.-A. Storey, "Peer review on open-source software projects: Parameters, statistical models, and theory," *ACM Trans. Softw. Eng. Methodol.*, vol. 23, no. 4, 2014, Art. no. 35.
- [76] P. C. Rigby, D. M. German, and M. Storey, "Open source software peer review practices: A case study of the apache server," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 541–550.
- [77] P. C. Rigby and M.-A. Storey, "Understanding broadcast based peer review on open source software projects," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 541–550.
- [78] P. Runeson, M. Alexandersson, and O. Nyholm, "Detection of duplicate defect reports using natural language processing," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 499–510.
- [79] M. Shahin, M. A. Babar, and L. Zhu, "Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices," *IEEE Access*, vol. 5, pp. 3909–3943, 2017.
- [80] J. Sheoran, K. Blincoe, E. Kalliamvakou, D. Damian, and J. Ell, "Understanding "watchers" on GitHub," in *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 336–339.
- [81] R. Souza and B. C. D. Silva, "Sentiment analysis of travis CI builds," in *Proc. IEEE/ACM 14th Int. Conf. Mining Softw. Repositories*, 2017, pp. 459–462.
- [82] D. Spencer, *Card Sorting: Designing Usable Categories*. New York, NY, USA: Rosenfeld Media, 2009.
- [83] I. Steinmacher, T. Conte, M. A. Gerosa, and D. Redmiles, "Social barriers faced by newcomers placing their first contribution in open source software projects," in *Proc. 18th ACM Conf. Comput. Supported Cooperative Work Soc. Comput.*, 2015, pp. 1379–1392.
- [84] I. Steinmacher, T. U. Conte, C. Treude, and M. A. Gerosa, "Overcoming open source project entry barriers with a portal for newcomers," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 273–284.

- [85] I. Steinmacher, G. Pinto, I. S. Wiese, and M. A. Gerosa, "Almost there: A study on quasi-contributors in open-source software projects," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng.*, 2018, pp. 256–266.
- [86] K.-J. Stol and B. Fitzgerald, "Two's company, three's a crowd: A case study of crowdsourcing software development," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 187–198.
- [87] C. Sun, D. Lo, S.-C. Khoo, and J. Jiang, "Towards more accurate retrieval of duplicate bug reports," in *Proc. 26th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2011, pp. 253–262.
- [88] C. Sun, D. Lo, X. Wang, J. Jiang, and S.-C. Khoo, "A discriminative model approach for accurate duplicate bug report retrieval," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 45–54.
- [89] Y. Tao, D. Han, and S. Kim, "Writing acceptable patches: An empirical study of open source project patches," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2014, pp. 271–280.
- [90] P. Thongtanunam, S. McIntosh, A. E. Hassan, and H. Iida, "Revisiting code ownership and its relationship with software quality in the scope of modern code review," in *Proc. 38th Int. Conf. Softw. Eng.*, 2017, pp. 1039–1050.
- [91] C. Treude, O. Barzilay, and M.-A. Storey, "How do programmers ask and answer questions on the web?: Nier track," in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 804–807.
- [92] C. Treude and M.-A. Storey, "Awareness 2.0: Staying aware of projects, developers and tasks using dashboards and feeds," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 365–374.
- [93] J. Tsay, L. Dabbish, and J. Herbsleb, "Influence of social and technical factors for evaluating contribution in GitHub," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 356–366.
- [94] J. Tsay, L. Dabbish, and J. Herbsleb, "Let's talk about it: Evaluating contributions through discussion in GitHub," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 144–154.
- [95] B. Vasilescu *et al.*, "The sky is not the limit: Multitasking across GitHub projects," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 994–1005.
- [96] B. Vasilescu *et al.*, "Gender and tenure diversity in GitHub teams," in *Proc. 33rd Annu. ACM Conf. Hum. Factors Comput. Syst.*, 2015, pp. 3789–3798.
- [97] B. Vasilescu, A. Serebrenik, M. Goeminne, and T. Mens, "On the variation and specialisation of workload—A case study of the gnome ecosystem community," *Empir. Softw. Eng.*, vol. 19, no. 4, pp. 955–1008, 2014.
- [98] B. Vasilescu, Y. Yu, H. Wang, P. Devanbu, and V. Filkov, "Quality and productivity outcomes relating to continuous integration in GitHub," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 805–816.
- [99] H. Wang, P. Shi, and Y. Zhang, "JointCloud: A cross-cloud cooperation architecture for integrated internet service customization," in *Proc. IEEE 37th Int. Conf. Distrib. Comput. Syst.*, 2017, pp. 1846–1855.
- [100] X. Wang, L. Zhang, T. Xie, J. Anvik, and J. Sun, "An approach to detecting duplicate bug reports using natural language and execution information," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 461–470.
- [101] P. Weißgerber, D. Neu, and S. Diehl, "Small patches get in!" in *Proc. Int. Work. Conf. Mining Softw. Repositories*, 2008, pp. 67–76.
- [102] M. Wessel *et al.*, "The power of bots: Characterizing and understanding bots in OSS projects," *Proc. ACM Hum.-Comput. Interaction*, 2018, Art. no. 182.
- [103] J. West and S. Gallagher, "Challenges of open innovation: The paradox of firm investment in open-source software," *R&D Manage.*, vol. 36, no. 3, pp. 319–331, 2006.
- [104] D. G. Widder, M. Hilton, C. Kästner, and B. Vasilescu, "A conceptual replication of continuous integration pain points in the context of travis CI," in *Proc. 27th ACM Joint Meeting Eur. Softw. Eng. Conf. and Symp. Found. Softw. Eng.*, 2019, pp. 647–658.
- [105] F. Wilcoxon, "Individual comparisons by ranking methods," *Biometrics Bull.*, vol. 1, no. 6, pp. 80–83, 1945.
- [106] Y. Yu, Z. Li, G. Yin, T. Wang, and H. Wang, "A dataset of duplicate pull-requests in GitHub," in *Proc. 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 22–25.
- [107] Y. Yu, H. Wang, G. Yin, and T. Wang, "Reviewer recommendation for pull-requests in GitHub: What can we learn from code review and bug assignment?" *Inf. Softw. Technol.*, vol. 74, pp. 204–218, 2016.
- [108] Y. Yu, G. Yin, T. Wang, C. Yang, and H. Wang, "Determinants of pull-based development in the context of continuous integration," *Sci. China Inf. Sci.*, vol. 59, no. 8, 2016, Art. no. 080104.
- [109] W. E. Zhang, Q. Z. Sheng, J. H. Lau, and E. Abebe, "Detecting duplicate posts in programming QA communities via latent semantics and association rules," in *Proc. 26th Int. Conf. World Wide Web*, 2017, pp. 1221–1229.
- [110] Y. Zhang, B. Vasilescu, H. Wang, and V. Filkov, "One size does not fit all: An empirical study of containerized continuous deployment workflows," in *Proc. 26th ACM Joint Meeting Eur. Softw. Eng. Conf. and Symp. Found. Softw. Eng.*, 2018, pp. 295–306.
- [111] Y. Zhang, D. Lo, X. Xia, and J.-L. Sun, "Multi-factor duplicate question detection in stack overflow," *J. Comput. Sci. Technol.*, vol. 30, no. 5, pp. 981–997, 2015.
- [112] M. Zhou and A. Mockus, "What make long term contributors: Willingness and opportunity in OSS community," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 518–528.
- [113] S. Zhou, S. Stanculescu, O. Lebenich, Y. Xiong, A. Wasowski, and C. Kästner, "Identifying features in forks," in *Proc. 39th Int. Conf. Softw. Eng.*, 2018, pp. 105–116.
- [114] S. Zhou, B. Vasilescu, and C. Kästner, "What the fork: A study of inefficient and efficient forking practices in social coding," in *Proc. 27th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2019, pp. 350–361.
- [115] J. Zhu, M. Zhou, and A. Mockus, "Effectiveness of code contribution: From patch-based to pull-request-based tools," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 871–882.
- [116] W. Zou, J. Xuan, X. Xie, Z. Chen, and B. Xu, "How does code style inconsistency affect pull request integration? An exploratory study on 117 GitHub projects," *Empir. Softw. Eng.*, vol. 24, no. 6, pp. 3871–3903, 2019.



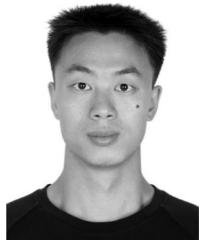
Zhiping Li received the bachelor's degree from Chongqing University, China, and the master's degree in computer science from the National University of Defense Technology, China. He is currently working toward the PhD degree in software engineering at the National University of Defense Technology (NUDT), China. His research goals are centered around the idea of making the open source collaboration more efficient and effective by investigating the challenges faced by open source communities and designing smarter collaboration mechanisms and tools.



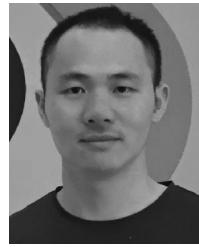
Yue Yu received the PhD degree in computer science from the National University of Defense Technology, China, in 2016. He is an assistance professor with the College of Computer, National University of Defense Technology (NUDT), China. He has won Outstanding PhD Thesis Award from Hunan Province. His research findings have been published on FSE, MSR, IST, ICSME, ICDM, and ESEM. His current research interests include software engineering, data mining, and computer-supported cooperative work.



Minghui Zhou received the BS, MS, and PhD degrees in computer science from the National University of Defense Technology, China, in 1995, 1999, and 2002, respectively. She is a professor with the Department of Computer Science, Peking University, China. She is interested in software digital sociology, i.e., understanding the relationships among people, project culture, and software product through mining the repositories of software projects. She is a member of the ACM.



Tao Wang received the PhD degree in computer science from the National University of Defense Technology, China, in 2015. He is an assistant professor with the College of Computer, National University of Defense Technology (NUDT), China. His work interests include open source software engineering, machine learning, data mining, and knowledge discovering in open source software.



Long Lan received the PhD degree in computer science from the National University of Defense Technology (NUDT), China, in 2017. He was a visiting PhD student with the University of Technology, Sydney, Australia, from 2015 to 2017. He is currently a lecturer with the College of Computer, National University of Defense Technology, China. His research interests focus on the theory and application of artificial intelligence.



Gang Yin received the PhD degree in computer science from the National University of Defense Technology, China, in 2006. He is an associate professor with the College of Computer, National University of Defense Technology (NUDT), China. He has worked in several grand research projects including National 973, 863 projects. He has published more than 60 research papers in international conferences and journals. His current research interests include distributed computing, information security, software engineering, and machine learning.



Huaimin Wang received the PhD degree in computer science from the National University of Defense Technology (NUDT), China, in 1992. He is now a professor and vice-president for academic affairs of the National University of Defense Technology, China. He has been awarded the "Chang Jiang Scholars Program" professor and the Distinct Young Scholar, etc. He has published more than 100 research papers in peer-reviewed international conferences and journals. His current research interests include middleware, software agent, and trustworthy computing.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

Integrating an Ensemble Surrogate Model's Estimation into Test Data Generation

Baicai Sun[✉], Dunwei Gong[✉], Member, IEEE, Tian Tian[✉], and Xiangjuan Yao[✉]

Abstract—For the path coverage testing of a Message-Passing Interface (MPI) program, test data generation based on an evolutionary optimization algorithm (EOA) has been widely known. However, during the use of the above technique, it is necessary to evaluate the fitness of each evolutionary individual by executing the program, which is generally computationally expensive. In order to reduce the computational cost, this article proposes a method of integrating an ensemble surrogate model's estimation into the process of generating test data. The proposed method first produces a number of test inputs using an EOA, and forms a training set together with their real fitness. Then, this article trains an ensemble surrogate model (ESM) based on the training set, which is employed to estimate the fitness of each individual. Finally, a small number of individuals with good estimations are selected to further execute the program, so as to have their real fitness for the subsequent evolution. This article applies the proposed method to seven benchmark MPI programs, which is compared with several state-of-the-art approaches. The experimental results show that the proposed method can generate test data with significantly low computational cost.

Index Terms—MPI program, path coverage testing, evolutionary optimization algorithm, ensemble surrogate model, test data generation

1 INTRODUCTION

DURING the development of MPI programs, communities related to high-performance computing can provide substantial supports [1]. In addition, MPI programs have advantages of high efficiency, good portability, and simple implementation [2]. Therefore, MPI programs have been widely used in the past two decades, and become a de-facto standard for writing parallel programs for computer clusters [3].

MPI programs have a series of characteristics, such as communication, synchronization, and non-determinism, which greatly increase the testing cost and difficulty. Especially, when executing an MPI program with non-determinism, different targets will generally be traversed under the same test data, suggesting the difficulty in generating test data to cover a given target. However, with the recognition that the issue resulted from non-determinism of MPI programs has been well tackled by our previous

work [4], we only consider MPI programs with determinism in this paper. In other words, the method proposed in this paper is also suitable for MPI programs with non-determinism, not just those with determinism. There have generally been a plenty of coverage criteria in software testing, e.g., statement coverage, branch coverage, and path coverage, and different criteria have different emphases. However, we can generally transform test data generation for other structural coverage to that for path coverage [5].

Given the fact that the methods of testing sequential programs have difficulties in testing parallel programs, Souza *et al.* [6] proposed a specific set of test criteria by taking the challenges of MPI programs into consideration. Furthermore, for additional features of MPI programs, e.g., collective and non-blocking communication, Souza *et al.* [7] proposed new structural test criteria by extending this set. These studies have focused on the features of MPI programs and proposed the corresponding structural coverage criteria, they have, however, not provided effective methods of generating test data.

For the path coverage testing of MPI programs without non-determinism, we used a co-evolutionary genetic algorithm (CGA) to automatically generate test data covering target paths [8]. However, when applying it to test complex MPI programs, the computational cost will be too high to be acceptable. The reason is as follows. When generating test data using an EOA, it is required to calculate the fitness of each individual by executing the program, which is generally computationally expensive. If we can first estimate the fitness of all the individuals based on knowledge obtained during the evolution and then calculate the real fitness of a subset of those individuals with the best estimated fitness, we will greatly reduce the number of times the program has to be executed, alleviating the computational cost resulted from individual evaluations.

- Baicai Sun is with the School of Information and Control Engineering, China University of Mining and Technology, Xuzhou, Jiangsu 221116, China. E-mail: baicaisun@gmail.com.
- Dunwei Gong is with the School of Information and Control Engineering, China University of Mining and Technology, Xuzhou, Jiangsu 221116, China, and also with the School of Information Science and Technology, Qingdao University of Science and Technology, Qingdao, Shandong 266061, China. E-mail: dwgong@vip.163.com.
- Tian Tian is with the School of Computer Science and Technology, Shandong Jianzhu University, Jinan, Shandong 250101, China. E-mail: tian_tiantian@126.com.
- Xiangjuan Yao is with the School of Mathematics, China University of Mining and Technology, Xuzhou, Jiangsu 221116, China. E-mail: yaoxj@cumt.edu.cn.

Manuscript received 30 Jan. 2020; revised 9 July 2020; accepted 22 Aug. 2020. Date of publication 25 Aug. 2020; date of current version 18 Apr. 2022. (Corresponding author: Dunwei Gong.) Recommended for acceptance by W. Visser. Digital Object Identifier no. 10.1109/TSE.2020.3019406

In order to reduce the above computational cost, Tong *et al.* [9] proposed an efficient surrogate-assisted EOA based on a Voronoi diagram, which can estimate the fitness of an individual. For reducing the number of individual evaluations when solving an optimization problem, Sun *et al.* [10] proposed a strategy for estimating the fitness of an individual based on the euclidean distance. In addition, Xiao *et al.* [11] proposed a fitness estimation method with an adaptive penalty function, so as to solve the problem of a high cost involved in evaluating an individual.

Moreover, there have been various single surrogate models (SSMs), e.g., polynomial regression [12], Gaussian process regression [13], support vector regression [14], and radial basis function network (RBFN) [15], [16], to be employed to estimate the fitness of an individual, so as to reduce the number of individual evaluations. However, a single surrogate model (SSM) is more prone to over-fitting or under-fitting, which will reduce the estimation accuracy, resulting in poor performance in generalization. In addition, previous studies have demonstrated that an ESM is superior to an SSM in terms of generalization for most cases [17]. As a result, we use an ESM to estimate the fitness of an individual in this paper.

Based on the above analysis, for the path coverage testing of complex MPI programs, we propose a method of integrating an ESM into the process of generating test data in this paper, so as to improve the efficiency of generating test data. In the proposed method, we train an ESM based on the training set formed in the steps of generating test data, which is employed to estimate the fitness of each individual. Following that, we select a small number of individuals with good estimations to execute the program, so as to achieve their real fitness for the subsequent evolution.

This paper has the following threefold novelties and contributions:

- 1) Defining the composition of a sample, and giving a method of forming the training set.
- 2) Proposing a method of constructing an MPI-based ESM.
- 3) Presenting a method of selecting superior individuals based on the rank of the estimated fitness of all individuals in a population.

The rest of this paper is organized as follows. Section 2 reviews the related work. The method of integrating an ESM into the process of generating test data is proposed in Section 3, including overall framework, forming the training set, constructing and applying an ESM, and selecting superior individuals for executing the program. Section 4 applies the proposed method to seven complex MPI programs, and compare it with other state-of-the-art approaches with analyzing the experimental results. Threats to validity are discussed in Section 5. Finally, Section 6 summarizes the whole paper, and points out the topics to be studied in the future.

2 RELATED WORK

2.1 Preliminary Knowledge

We have provided some studies associated with the path coverage testing of MPI programs in [8], among which some basic concepts can be employed in this paper. These concepts

```

/* MASTER CODE*/
#include <mpi.h>
#include <iostream>
#define NUM_SLAVES 2
#define MASTER 0
using namespace std;
int main(int argc, char *argv[]){
    int myid, i, j, sum=0, sbuf[2][2], rbuf[2][2];
    char slave[100] = "slave.exe";
    MPI_Status status;
    1 MPI_Comm icomm;
    2 MPI_Init(&argc, &argv);
    3 MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    4 cin>>sbuf[0][0]>>sbuf[0][1]>>sbuf[1][0]>>sbuf[1][1];
    5 MPI_Comm_spawn(slave, MPI_ARGV_NULL, NUM_SLAVES, \
                    MPI_INFO_NULL, MASTER, MPI_COMM_SELF, \
                    &icomm, MPI_ERRCODES_IGNORE);
    6 for (i = 0; i<2; i++)
        7     for (j = 0; j<2; j++)
            8         { sum += sbuf[i][j]; }
    9 for (i = 0; i < NUM_SLAVES; i++)
    10 { MPI_Send(&sum, 1, MPI_INT, i, i, icomm); }
    11 MPI_BARRIER(icomm);
    12 MPI_Scatter(sbuf, 2, MPI_INT, NULL, 0, MPI_INT, MPI_ROOT, icomm);
    13 MPI_Gather(NULL, 0, MPI_INT, rbuf, 2, MPI_INT, MPI_ROOT, icomm);
    14 MPI_Finalize();
    15 return 0;
}

```

Fig. 1. Master process source code for the example program.

include MPI program, node, control-flow graph, path, and the path similarity. In the following, we will briefly introduce them, and please refer to [8] for more details.

An MPI program refers to a program consisting of a number of processes which execute in parallel and communicate with each other, denoted as $S = \{s^0, s^1, \dots, s^{m-1}\}$, where $s^i (i = 0, 1, \dots, m - 1)$ represents the i th process in S , and m is the number of processes. For S , its input can be represented as $X = (x_1, x_2, \dots, x_n)$, where $x_j (j = 1, 2, \dots, n)$ means the j th input variable of X . If the range of x_j is h_j , then the range of X will be $H = h_1 \times h_2 \times \dots \times h_n$.

This section introduces some basic concepts and illustrates them through an MPI example program, shown in Figs. 1 and 2. Fig. 1 shows the master process of the example program, and is denoted as s^0 , and Fig. 2 provides two slave processes created by the master process, denoted as s^1 and s^2 , respectively. In the program, the *Spawn* primitive aims to create two slave processes, the *Scatter* primitive is employed to send *sbuf* from the master process to *rbuf* in the slave processes, where *sbuf* means a sending buffer, and *rbuf* refers to a receiving buffer. In addition, the *Gather* primitive has the function of collecting *sbuf* from the slave processes, and storing into *rbuf* in the master process. Please refer to [18] for more details about the usage of the above three MPI primitives.

Node. For a process of an MPI program, a node refers to a basic execution unit. For process s^i , the j th node is represented as n_j^i , which corresponds to a series of sequentially executed commands or a communication primitive. In n_j^i , j corresponds to the line number in Figs. 1 and 2.

Control-Flow Graph. The control flow graph of S can be represented as $G = \{V, E\}$, where V and E mean the node set and the edge set, respectively. For two nodes $n_l^i, n_k^i \in V$, if n_k^i is executed after n_l^i , then there will exist an edge, denoted as $\langle n_l^i, n_k^i \rangle$, from n_l^i to n_k^i , called a control edge. For $n_l^i, n_k^j \in V$, $i \neq j$, if n_l^i is a sending node and n_k^j is its corresponding receiving node, then there will exist an edge, $\langle n_l^i, n_k^j \rangle$, from n_l^i to n_k^j , termed a communication edge. As a result, the edge set, E , consists of all the control and communication

```

/* SLAVE CODE*/
#include <mpi.h>
#define MASTER 0
int main(argc, char *argv[]){
    int myid, i, sum = 0, rbuf[2], sbuf[2];
    MPI_Status status;
    1 MPI_Comm icomm;
    2 MPI_Init(&argc, &argv);
    3 MPI_Comm_get_parent(&icomm);
    4 MPI_Comm_rank(MPI_COMM_WORLD, &myid);
    5 MPI_Recv(&sum, 1, MPI_INT, MASTER, myid, icomm, &status);
    6 MPI_BARRIER(icomm);
    7 MPI_Scatter(NULL, 0, MPI_INT, rbuf, 2, MPI_INT, MASTER, icomm);
    8 for (i = 0; i<2; i++)
    9 { sbuf[i]=rbuf[i] * sum; }
    10 MPI_Gather(sbuf, 2, MPI_INT, NULL, 0, MPI_INT, MASTER, icomm);
    11 MPI_Finalize();
    12 return 0;
}

```

Fig. 2. Slave process source code for the example program.

edges. Fig. 3 is the control-flow graph of the example program.

Path. When S is executed under $X \in H$, a series of traversed nodes in s^i form a sub-path, denoted as p^i . The number of nodes in p^i is called the path length of p^i , denoted as $|p^i|$. Furthermore, the path, $P(X)$, traversed by X is composed of side-by-side traversal sub-paths, denoted as $P(X) = \{p^0, p^1, \dots, p^{m-1}\}$.

For two paths, P^* and $P(X)$, where P^* is a target path, and is denoted as $P^* = \{p^{*0}, p^{*1}, \dots, p^{*m-1}\}$, the path similarity between them can be formulated as [8]

$$\text{Sim}(P^*, P(X)) = \frac{1}{m} \sum_{i=0}^{m-1} \text{sim}(p^{*i}, p^i), \quad (1)$$

where $\text{sim}(p^{*i}, p^i)$ is the sub-path similarity, with the following expression:

$$\text{sim}(p^{*i}, p^i) = \frac{|p^{*i} \cap p^i|}{\max\{|p^{*i}|, |p^i|\}}, \quad (2)$$

where $|p^{*i} \cap p^i|$ is the number of successively same nodes between p^{*i} and p^i from the first node, and $\max\{|p^{*i}|, |p^i|\}$ refers to the maximum path length of p^{*i} and p^i .

Moreover, we modeled the problem on test data generation as that of an evolutionary optimization [8]. From the Formula (1), the larger $\text{Sim}(P^*, P(X))$ is, the closer $P(X)$ is to P^* . When $\text{Sim}(P^*, P(X)) = 1$, $P(X)$ is namely the target path P^* , and X is the test datum covering the target path. In this way, we can model the problem of generating test data that cover P^* as the following single-objective optimization problem:

$$\begin{aligned} \max F(X) &= \text{Sim}(P^*, P(X)) \\ \text{s. t. } X &\in H. \end{aligned} \quad (3)$$

It should be noted that the above formula is the fitness function for evaluating an individual in the steps of generating test data using an EOA.

2.2 Test Data Generation

When generating test data based on an EOA, a number of targets are usually aggregated into a fitness function, which reduces the coverage of test data aiming at a specific target to some extent. To overcome this drawback, Panichella *et al.*

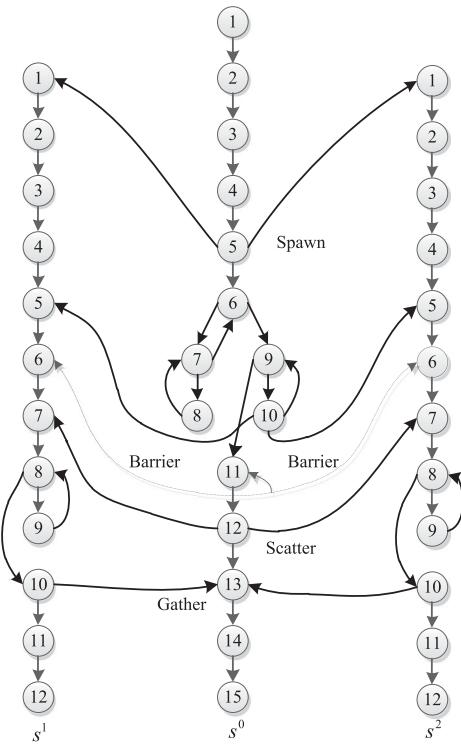


Fig. 3. Control-flow graph for the example program.

[19] proposed a dynamic multi-objective sorting algorithm, which was employed as a many-objective solver to improve the efficiency of generating test data. In order to reduce the cost of regression testing based on a user's needs, Gupta *et al.* [20] developed an automatic regression testing tool to generate test data with high efficiency. In addition, Scalabrinio *et al.* [21] proposed a search-based tool, Ocelot, to automatically generate test data, with the purpose of reducing the testing cost. For improving the efficiency of generating test data based on an EOA, Lv *et al.* [22] proposed a method of generating test data for covering multiple paths by combining metamorphic relations. However, the above methods are only applicable to sequential programs.

For variable correlations in multi-threaded code, Janne-sari and Wolf [23] proposed a parallel method of generating test data, so as to improve the efficiency of detecting concurrent bugs. In order to reduce the computational complexity of finding concurrency bugs, Bo *et al.* [24] proposed an effective method to improve the efficiency of generating test data by combining the advantages of both bug-driven and coverage-guided techniques. For improving the efficiency of generating test data for multi-threaded concurrent programs, Yue *et al.* [25] proposed an input-driven active testing approach with two test input selection strategies based on the diversity metric of test data. For concurrent program testing, Sahooa and Ray [26] compared the efficiency of search-based techniques in test data generation. However, the above methods are only suitable for testing concurrent programs. Concurrent algorithms are indeed beneficial to the development of test data generation techniques for concurrent programs. However, these concurrent algorithms have not provided effective strategies for solving the problem of high computational cost when generating test data. In view of this, our proposed approach estimates the fitness

of each evolutionary individual based on knowledge obtained during the evolution, and only select a small number of individuals with good estimations to execute programs under test, which can alleviate the computational cost resulted from program executions.

For the path coverage testing of MPI programs, we employed a co-evolutionary genetic algorithm to automatically generate test data that cover a path of an MPI program without non-determinism [8]. To alleviate negative influences of these scheduling sequences, we proposed a method of reducing scheduling sequences when generating test data to cover a target path [4]. In addition, we utilized a genetic algorithm to generate test data that cover a path of an MPI program by mutating the communication parameters of a non-deterministic primitive [27].

However, when the above methods are employed to generate test data, it is required to execute an MPI program under test for the maximum number of runs, i.e., the product of the population size and the maximum number of generations in the worst case, so as to evaluate the fitness of each individual. Especially, considering that executing a complex MPI program is time-consuming, if a method requires a large number of program executions, it will be said computationally expensive. To reduce the computational cost for a complex MPI program, this paper employs an ESM to estimate the fitness of each individual, with the aim of greatly reducing the number of times the program has to be executed, hence alleviating the computation cost for individual evaluations.

2.3 Ensemble Surrogate Model

Surrogate model has been widely applied in software testing with various testing criteria. To determine the correctness of a program, Chang *et al.* [28] formed a training set based on test data and their covered paths, and obtained a surrogate model of simulating the execution process of a program. To get representative test data, Christiansen and Dahmcke [29] employed a surrogate model to analyze existing and manually marked data, and combined them with newly generated data. In addition, when applying a surrogate model to software testing, Zhang [30] pointed out that an ESM is superior to an SSM. Therefore, we employ an ESM to estimate the fitness of an individual in this paper, so as to generate test data with high efficiency.

There generally have two ways to train an ESM. One is to train each base surrogate model (BSM) in an iterative manner, and the other is to do it in a parallel manner. The former is so called boosting [31], which is often computationally expensive. The latter is termed bagging [32], with obvious advantages in terms of the time consumption. Furthermore, Zhou [33] pointed out that a bagging ESM often utilizes the method of bootstrap to generate a number of training sets, so as to train each BSM, thus avoiding the over-fitting or under-fitting problem resulted from an SSM. In view of the goal of reducing the computational cost in generating test data, we employ the bagging method to train an ESM.

Given a training set, denoted as $D = \{(X_j, y_j), j = 1, 2, \dots, N\}$, where (X_j, y_j) represents a training sample, X_j and y_j are the input and the output of the sample, respectively, and N refers to the number of samples. In addition, N_p represents the number of BSMs in an ESM. To achieve

an ESM, we first generate N_p training sets using the bootstrap method based on D , denoted as $D^l (l = 1, 2, \dots, N_p)$, with the same size as D . Following that, we train the l th BSM, denoted as BSM^l , based on D^l , which is utilized as the l th estimation model. Finally, either the voting method or the average method is employed according to a specific task to each BSM when forming the estimation output of the ESM. Algorithm 1 provides the principle of an ESM. In the algorithm, $T = \{(X_j^t), j = 1, 2, \dots, M\}$ is an testing input set with its size of M , among which the output of each input is required to estimate using the ESM.

Algorithm 1. The Principle of an ESM

```

Input:  $D, N_p, T$ 
Output:  $\bar{Y}$  (output estimations of  $T$ )
1: for  $l = 1 \rightarrow N_p$  do
2:    $D^l = bootstrap(D);$ 
3:   Train  $l$ th base surrogate model  $BSM^l$  based on  $D^l$ ;
4:   for  $j = 1 \rightarrow M$  do
5:      $\hat{y}_j^l = BSM^l(X_j^t);$ 
6:   end
7: end
8: for  $j = 1 \rightarrow M$  do
9:    $sum = 0;$ 
10:  for  $l = 1 \rightarrow N_p$  do
11:     $sum += \hat{y}_j^l;$ 
12:  end
13:   $\bar{y}_j^t = sum/N_p;$ 
14: end
15: return  $\bar{Y} = [\bar{y}_1^t, \bar{y}_2^t, \dots, \bar{y}_M^t];$ 

```

3 INTEGRATING AN ESM INTO TEST DATA GENERATION

3.1 Overall Framework

In this section, we propose the integration of an ESM into the process of generating test data using an EOA. The purpose of the integration is to improve the efficiency of generating test data by reducing the number of times the program has to be executed.

Algorithm 2 provides the overall framework of the proposed method. For the inputs of the pseudo-code, $ColGen$ is the number of iterations using an EOA to form the training set, $MaxGen$ is the maximum number of generations for evolving a population using an EOA, $|B|$ is the number of target paths (please see Section 4.3 for details). In Algorithm 2, we first initialize a population, set the coverage flag of each path to 0, and let each training set be empty (lines 1 to 3). Then, the current population is taken as input of Algorithm 3, which is executed to form the training sets for all the target paths (line 7). Next, we construct an ESM based on the training set by executing Algorithm 4, and estimate the fitness of each evolutionary individual (line 10). Following that, Algorithm 5 is executed based on the fitness estimations, and a small number of representative individuals are selected to execute the program, with the purpose of achieving their real fitness (lines 11 to 13). Next, during generating test data covering the l th path, if the fitness of an individual is equal to 1, test data generation of the $(l+1)$ th target path will be performed (lines 14 to 22). Finally, we

update the current population to a new one based on Formula (4) and return to line 7 or 10, so as to form each training set or generate test data covering the current target path (line 23).

Algorithm 2. The Framework of the Proposed Method

```

Input:  $MaxGen$ ,  $ColGen$ ,  $|B|$ ,  $w$ ,  $c_1$ ,  $c_2$ ,  $\xi$ ,  $\eta$ 
Output:  $TD$  (test data covering all the target paths)
1: Initialize population;
2:  $Flag[r] = 0$  ( $r = 1, 2, \dots, |B|$ ) (coverage flag);
3:  $D_k = \emptyset$  ( $k = 1, 2, \dots, |B|$ ) (training set);
4: for  $r = 1 \rightarrow |B|$  do
5:   for  $i = 1 \rightarrow MaxGen$  do
6:     if  $i \leq ColGen$  &&  $r == 1$  then
7:       Execute Algorithm 3 with the current population as
        input to form the training set  $D_k$  ( $k = 1, 2, \dots, |B|$ );
8:     end
9:     if  $i > ColGen$  ||  $r \neq 1$  then
10:      Execute Algorithm 4 with  $D_r$  to construct an ESM as
        well as apply it to estimate the fitness of each individual
        in the current population;
11:      Execute Algorithm 5 to select  $selsize$  individuals,
        denoted as  $Ind$ ;
12:      for  $j = 1 \rightarrow selsize$  do
13:        Evaluate  $F(Ind_j)$  by executing the program;
14:        if  $F(Ind_j) == 1$  then
15:          Add  $Ind_j$  to  $TD$ , and  $Flag[r] = 1$ ;
16:          Break;
17:        end
18:      end
19:    end
20:    if  $Flag[r] == 1$  then
21:      Break;
22:    end
23:    Evolve the current population into a new one and
      redefine the new population as the current one;
24:  end
25: end
26: return  $TD$ ;
  
```

It should be noted that Algorithm 3 is employed to form the training sets, the function of Algorithm 4 is to construct and apply an ESM, and Algorithm 5 aims to selecting superior evolutionary individuals. The details of the above algorithms will be given in Sections 3.2, 3.3, and 3.4, respectively.

3.2 Formatting the Training Set

During the process of generating test data using an EOA, a large amount of knowledge associated with testing is generally generated. In order to make full use of these knowledge, we combine the generated test data with their fitness evaluations to form a training set, which is employed to train an ESM. With the recognition that the fitness of an evolutionary individual are not the same for different target paths, we set a training set to store evolutionary individuals and their fitness for each target path.

To form the training set, we first use an EOA to generate a certain number of individuals, i.e., test data, and obtain their traversal paths by executing programs. Then, the fitness of each individual is calculated between the obtained traversal path and each target one, respectively. Finally, for

each target path, we add those individuals and their fitness to the corresponding training set.

In addition, there will inevitably be redundancy between the evolutionary individuals in the steps of forming the training set. Therefore, we calculate the information entropy after adding a new sample, so as to determine whether the sample can be added to the training set or not [34]. In other words, an individual and its fitness can be added if and only if the information entropy of the training set can be increased. The reason lies in that a training set having a large information entropy generally contains samples with uniform distribution, which is beneficial to the generalization performance of a surrogate model trained based on the training set.

The pseudo-code of forming the training set using an EOA is provided in Algorithm 3. In the algorithm, we execute the program under each individual in the population, and obtain the traversal path of the individual (line 2). For each target path, we calculate the fitness of each individual, and update the training set based on the information entropy (lines 3 to 6).

Algorithm 3. Forming the Training Set

```

Input:  $pop = \{(X_j), j = 1, 2, \dots, popsize\}$  (population),
         $popsize$  (population size)
Output:  $D_k$  ( $k = 1, 2, \dots, |B|$ ) (training set)
1: for  $j = 1 \rightarrow popsize$  do
2:   Execute the program under  $X_j$  in  $pop$  and get the traversal
   path  $P(X_j)$ ;
3:   for  $k = 1 \rightarrow |B|$  do
4:     Calculate the fitness  $F(X_j)$  between the  $k$ th target path
   and  $P(X_j)$ ;
5:     Add  $X_j$  and  $F(X_j)$  to  $D_k$  based on the information
   entropy;
6:   end
7: end
8: return  $D_k$ ;
  
```

3.3 Constructing and Applying an ESM

Based on the training set formed in Section 3.2, we construct an estimation model and apply it to estimate an individual's performance in this section. To fulfill this task, we need to address the following two issues. One is the type that the surrogate model belongs to, and the other is the way to train the surrogate model.

For the first issue, considering that the SSM has disadvantages, e.g., low generalization performance, which is mainly manifested in over-fitting or under-fitting. Therefore, we adopt an ESM as the type of the estimation model. In addition, given the fact that RBFNs have been widely used in estimating the fitness of an individual [15], [16], [35], [36], an RBFN is employed as a BSM. It should be noted that other SSMs can also be employed as a BSM in an ESM, however, the focus of this paper is to study the effectiveness and efficiency of applying surrogate models to reduce the testing cost, rather than which type of surrogate models to be used.

RBFN is a network with a unique hidden layer and output layer. The activation function of each hidden layer node is Gaussian, and that of the output node is linear. When RBFN was applied to estimate the fitness of an evolutionary

individual, Wang *et al.* [37] set the number of nodes in the hidden layer as the dimension of the decision variable, and used the K-means algorithm to train the center and width of each Gaussian kernel function. In addition, the pseudo-inverse method has been applied to train the connection weight between each hidden layer node and the output node in [37]. Therefore, we use the above same methods to solve these parameters in this section.

Regarding the second issue, we develop a method of constructing and applying an MPI-based ESM. This method includes a master process and several slave processes, the details of which are as follows: we first use the bootstrap sampling method [38] to obtain several training sets and create several slave processes in the master process. Then, we place an RBFN in each slave process, and train the RBFN in the slave process based on a training set. Following that, the fitness of each evolutionary individual is estimated using the trained RBFN in each slave process. Finally, we use the average method to integrate the fitness estimation of all RBFNs as the individual's final estimation in the master process.

Algorithm 4 gives the pseudo-code for the construction and application of an MPI-based ESM. For the number of RBFNs, we generate the same number of training sets and create the same number of slave processes in this algorithm (lines 1 to 5). Each slave process allocates a generated training set for training an RBFN in the slave process, so as to estimate the fitness of each evolutionary individual (lines 6 to 13). We calculate the average estimations of all RBFNs as the fitness estimation for this individual (line 15).

Algorithm 4. Constructing and Applying an MPI-Based ESM

Input: $pop = \{(X_j), j = 1, 2, \dots, popsize\}$ (population), D_k (training set corresponding to the k th target path), N_p (number of RBFNs)

Output: $\bar{F}(X_j)$ (fitness estimation of individual X_j)

- 1: // Master process;
- 2: **for** $l = 1 \rightarrow N_p$ **do**
- 3: $D_k^l = bootstrap(D_k);$
- 4: **end**
- 5: Create N_p slave processes;
- 6: **for** $l = 1 \rightarrow N_p$ **do**
- 7: Send D_k^l to $RBFN^l$ in the l th slave process;
- 8: **end**
- 9: // Slave process;
- 10: **for** $l = 1 \rightarrow N_p$ **do**
- 11: Train $RBFN^l$ based on $D_k^l;$
- 12: $\hat{F}^l(X_j) = RBFN^l(X_j);$
- 13: **end**
- 14: // Master process;
- 15: Calculate the mean of all $\hat{F}^l(X_j)$ as the fitness estimation, denoted as $\bar{F}(X_j);$
- 16: **return** $\bar{F}(X_j);$

3.4 Selecting Superior Individuals

In the process of generating test data using an EOA, we need to execute the program under each individual in the population, which will lead to expensive computational costs. In view of this, if a small number of individuals are

selected to execute a program for obtaining their real fitness, the number of times the program has to be executed will be greatly reduced, hence alleviating the computational costs resulted from evaluating individuals.

Since we tackle the problem of generating test data for path coverage, the larger the fitness of an individual is, the closer the path traversed by the individual is to the target path, so the closer the individual is to the desired test datum. Therefore, we select individuals with a large fitness estimations to execute the program.

Algorithm 5. Selecting Superior Individuals

Input: $pop = \{(X_j), j = 1, 2, \dots, popsize\}$ (population), $\bar{F}(X_j)$ (fitness estimation of X_j)

Output: Ind (selected individuals)

- 1: **for** $i = 1 \rightarrow popsize$ **do**
- 2: **for** $j = 1 \rightarrow popsize - i$ **do**
- 3: **if** $\bar{F}(X_j) < \bar{F}(X_{j+1})$ **then**
- 4: Exchange X_j and $X_{j+1};$
- 5: **end**
- 6: **end**
- 7: **end**
- 8: **for** $j = 1 \rightarrow selsize$ **do**
- 9: Add X_j to selected individuals $Ind;$
- 10: **end**
- 11: **return** $Ind;$

To fulfill this task, we sort individuals within a population in a descending order of their fitness estimations, and select a small number of individuals with good estimations, called superior individuals, from front to back, so as to further execute the program for calculating their real fitness. Algorithm 5 gives the pseudo-code for selecting superior individuals. In the algorithm, evolutionary individuals are ranked (lines 1 to 7). a small number of individuals are selected (lines 8 to 10).

4 EXPERIMENTS

We apply the proposed method to test various complex MPI programs in this section, and verify whether it can improve the efficiency of generating test data or not through a series of experiments. The contents of this section are organized as follows. The research questions are first raised, followed by the programs under test and experiment settings. Finally, the experimental results are provided and analyzed.

4.1 Research Questions

In this paper, an EOA is employed to generate a number of test inputs, and a training set is formed by combining these test data with their fitness. Based on the training set, an ESM is trained to estimate the fitness of an individual. Therefore, it is necessary to verify whether the ESM can accurately estimate the fitness of an individual. In addition, based on the estimated fitness, we select a small number of individuals with good performance to execute the MPI program, so as to achieve their real fitness. Thus, it is necessary to verify that executing an MPI program with superior individuals can effectively reduce the computational costs resulted from evaluating individuals.

When using the proposed method to generate test data that cover the target paths, if such operations as forming the training set, training an ESM, and selecting superior individuals to execute a program are helpful for improving the efficiency of generating test data, then it will be shown that the proposed method is advantageous. In view of the above analysis, the following research questions are raised.

RQ1 Can the ESM trained based on the formed training set accurately estimate the fitness of an individual?

Given the fact that a single RBFN has been widely used to estimate the fitness of an individual, the ESM and the single RBFN are employed to estimate the fitness of an individual during the evolution to answer this question, respectively.

Here, we denote the comparative method of integrating an RBFN into test data generation as RBFN-S. The proposed method and RBFN-S are employed to generate test data, respectively. If the proposed method is more efficient and effective, then it is rational that the ESM can accurately estimate the fitness of an individual.

RQ2 Has the proposed method a low computational cost due to executing an MPI program only with selected superior individuals?

Considering that particle swarm optimization (PSO) has the advantages of a simple mechanism, easy implementation, and fast optimization speed, Windisch [39] proposed a method of generating test data using PSO, and verified that it is superior to genetic algorithms. In view of this, we employ PSO as an EOA to generate test data, so as to evaluate the method proposed in this paper.

To reduce the testing cost, we employ an ESM to estimate the fitness of each individual in the subsequent evolutions, so as to alleviate the number of evaluations. However, we do not know the experimental results without the proposed strategies. In other words, the computational cost of only employing PSO to generate test data is unknown. Therefore, it is necessary to compare the proposed method with the method of generating test data only using PSO, which can verify whether the proposed method can improve the testing efficiency.

Here, we denote the method of generating test data using PSO without the proposed strategies as PSO-W. To answer this question, we generate test data using the proposed method and PSO-W, respectively. If it is more efficient and effective to generate test data using the proposed method, the computation cost will be reduced if we execute the programs against a small number of individuals with good estimations.

RQ3 Can the proposed method improve the efficiency of generating test data?

In view that CGA has been proposed to generate test data that cover paths of MPI programs and it is superior to genetic algorithms and the random method in terms of efficiency and effectiveness [8], we generate test data using the proposed method and CGA, respectively. If the proposed method is more efficient and effective than CGA, the proposed method will have a capability in improving the effectiveness and efficiency in generating test data, and the experimental results will be more valuable and trustworthy for MPI programs.

We adopt the student's t-test to show whether there is a significant difference or not in terms of an indicator between

TABLE 1
Basic Information of the Programs Under Test

Program	# of processes	# of primitives	lines of code (LOC)
<i>Convex</i>	14	49	569
<i>QR_value</i>	8	34	575
<i>Cjacobi</i>	7	97	721
<i>Heat</i>	12	72	613
<i>DepSolver</i>	6	63	8,988
<i>Kfray</i>	8	116	12,728
<i>ClustalW</i>	24	178	23,265

the proposed method and the comparative one in this paper. To fulfill this task, we set the significance level to 0.05, which indicates that if the P-value of an indicator is smaller than 0.05, then the proposed method will have a significant difference with the comparative one. On this circumstance, we further determine whether the proposed method is significant better than the comparative one or not according to their average values.

4.2 The Programs Under Test

In the experiments, seven complex open source MPI programs are selected as test objects. Among them, *Convex*, *QR_value*, and *Cjacobi* were implemented by Guoliang Chen (glchen@ustc.edu.cn) and his team members [40], with *Convex* being utilized to seek the smallest convex polygon from all the given points in a plane. For *QR_value*, it aims to obtain the eigenvalues of a matrix. Regarding *Cjacobi*, its function is to diagonalize a symmetric matrix by an orthogonal similarity transformation. In addition, *Heat* is a parallel solver for heat equations [41], and was developed by David Lecomber (david@allinea.com) *et al.* With respect to *Depsolver*, *Kfray*, and *ClustalW*, they were employed in [42]. Among them, *Depsolver* is a parallel multi-media 3D electrostatic solver, *Kfray* is a ray tracing program which is utilized to create real images, and *ClustalW* is a commonly used multi-gene sequence alignment tool. The developers of the above three programs are Carlos Rosales Fernandez (carlos.rosales.fernandez@gmail.com), Ait-Si-Amer (aitsiame@polytech.upmc.fr), and Mick Elliot (micke@sfsu.ca), respectively. It should be noted that the communication behaviors of the above benchmarks include both blocking/non-blocking point-to-point communication and collective communication. The basic information of these programs is listed in Table 1.

Myers *et al.* [43] pointed out that if a program contains more than 500 lines of code, then the program will belong to a large program. In addition, the computational cost when testing an MPI program lies in the following two aspects: the calculation cost and the communication cost. Among them, the calculation cost is mainly affected by the intra-process calculation, which can be determined by the lines of code to a certain extent. In contrast, the communication cost is mainly determined by the number of inter-process communication, which can be indirectly judged by the number of communication primitives and the number of processes. Based on the above analysis and the data in Table 1, we can observe that the programs under test are diverse in the lines of code, the number of processes, and the number of communication primitives, indicating their sufficient complexity and representativeness.

4.3 Experimental Settings

The experimental platform includes 2 computing nodes. The hardware configuration of each computing node is Intel Core i9-9900K CPU, 32GB RAM, 1TSSD hard disk, and Gigabit Ethernet. Its software configuration is Windows 10 operating system, MPI+C/C++ programming language, and Shark machine learning library [44].

In this paper, the following two steps are taken to select the basis target path set for a program, so that the paths in the set have a good coverage of the statements and branches of the program. First, the set of feasible basis sub-paths is selected for each process of a program according to [45]. Following that, the feasible sub-paths corresponding to all the processes of the program are combined to form a number of target paths, which further constitute the basis target path set.

For the first step, the set of feasible basis sub-paths in s^i is denoted as B_i , and initially $B_i = \emptyset$. We first extract a sub-path, denoted as sp_1^i , using the breadth-first search algorithm [46] based on the CFG of s^i . Then, we determine its feasibility by use of EPAT [47]. If it is feasible, we will put it into B_i . Next, we investigate the second sub-path, denoted as sp_2^i , followed by determining whether it can be linearly expressed by the sub-path in B_i or not through the linear programming tool, lpsolve, in [48]. If it cannot and is feasible, then we will put it into B_i . The above steps will be repeated until all the basis sub-paths of s^i are checked.

Regarding the second step, the number of feasible sub-paths of s^i is denoted as $|B_i|$. To obtain a target path of S , we first generate a copy of B_i . Then, we sample B_i without replacement, and form a target path of S by combining all the sampled sub-paths. If $B_i = \emptyset$, we will further sample the copy of B_i with replacement, and form another target path of S using the same method. The above process will be repeated until the sub-path set with the largest cardinality is empty. Here, all the combined target paths form the set of basis target paths to be covered, denoted as B . In this way, the number of target paths in B is the same as the cardinality of the largest sub-path set, and denoted as $|B|$.

The above method can guarantee that the selected target paths include any statement and branch of the program under test, suggesting their good representativeness. For more details of selecting B , please refer to [4].

When using PSO, the position and speed of each evolutionary individual in a population are updated based on the Formula (4).

$$\begin{cases} V_j^{gen+1} = w \cdot V_j^{gen} + c_1 \cdot \xi \cdot (p_j^{gen} - X_j^{gen}) \\ \quad + c_2 \cdot \eta \cdot (p_g^{gen} - X_j^{gen}) \\ X_j^{gen+1} = X_j^{gen} + V_j^{gen+1} \end{cases}, \quad (4)$$

where gen represents the number of iterations. For the parameter settings of PSO, we also employ the values from [39], which are provided as follows. The value of w decreases linearly from 0.9 to 0.4. The population size is 40 ($j = 1, 2, \dots, 40$). The learning factors, c_1 and c_2 , are 1.49. The V_{max} is set according to the input space of a specific program, and $MaxGen$ is 1,200.

To employ CGA, we continue to use the parameter settings of CGA from [8], and the related parameters are given as follows. The size of the cooperative population is 10, the number and size of sub-populations are $m - 1$ and 30,

respectively, the numbers of representatives and dominant individuals are equal to 2 and 4, respectively, and the periods of evolving the cooperative and sub-populations are 5 and 2, respectively. In addition, roulette-wheel selection, one-point crossover, and one-point mutation with their probabilities of 0.9 and 0.3, respectively, are adopted.

In addition, we analyze the computational complexity of PSO, CGA, and the proposed method when generating test data, so as to highlight the objective of this paper. Considering that the population size of PSO is 40, the complexity, i.e., the number of times the program has to be executed, when generating test data using PSO in the worst case is $40 \cdot MaxGen \cdot |B|$. In view that CGA contains a cooperative population with its size of 10 and $m - 1$ sub-populations with each size of 30, and $m - 1$ sub-populations in all processes are equivalent to a cooperative population, the computational complexity of CGA in the worst case is $(10 + 30) \cdot MaxGen \cdot |B|$. With respect to the proposed method, the complexity is divided into two parts, one is the number of times the program has to be executed, and the other is the number of training and applying surrogate models. In view of this, the computational complexity of the former and the latter using the proposed method is $(40 - selsize) \cdot ColGen + selsize \cdot MaxGen \cdot |B|$ and $|B| + 40 \cdot MaxGen \cdot |B| - 40 \cdot ColGen$, respectively, in the worst case. Based on the above analysis, the efficiency of the proposed method depends on whether the training and application of surrogate models can significantly reduce the computational cost of program executions.

Moreover, Quinlan [49] set the number of BSMs to 1, 2, ..., 50, for achieving the optimal number of BSMs in an ESM. In order to obtain the optimal value of Np in a larger scope, we set the value of Np to integers ranging from 1 to 150 in this paper.

We employ the following two indicators when answering the research questions, the success rate and the time consumption. Among them, the success rate reflects the effectiveness of a method, which is calculated by the ratio of the number of runs which successfully find the desired test data to the total number of runs. For the time consumption, it is employed to measure the efficiency of a method, which is time spent in generating test data, or reaching the maximum number of generations when failing to generate test data. It is clear that the higher the success rate and the lower the time consumption of a method are, the more advantageous the method is. In order to alleviate the negative influence of random factors on the performance of a method, each method is run 20 times independently for each basis target path set, and the experimental results of each run are recorded, followed by calculating the success rate and the average time consumption.

4.4 Key Parameter Settings

Before generating test data using the proposed method, we need to determine three key parameters, i.e., $ColGen$, Np , and $selsize$.

To determine the optimal value of $ColGen$, we take *Convex* as an example, and select its 15 basis target paths. In addition, we set Np and $selsize$ to 75 and 20, respectively. For the selected target paths, we first set $ColGen$ from 1 to 1,200 with the step size of 10, and evolve a population using

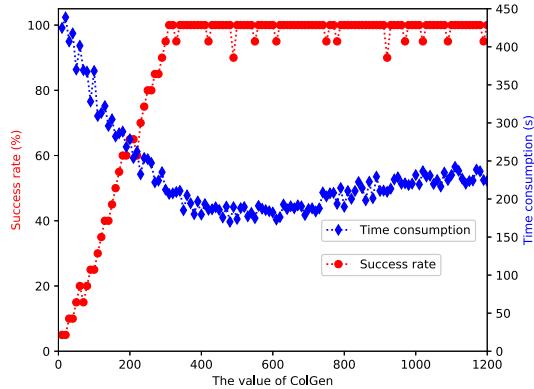


Fig. 4. The curves of the success rate and the average time consumption w.r.t. the value of *ColGen*.

PSO for *ColGen* generations, so as to form the corresponding training set for each target path. Then, an MPI-based ESM is trained based on each training set, followed by estimating the fitness of each individual in the subsequent population using the ESM. Finally, twenty superior individuals are selected based on the estimated fitness to execute the program. The above steps are run 20 times, and the success rate and the average time consumption are calculated. We will obtain the optimal value of *ColGen* when the maximum success rate and the minimum average time consumption are achieved.

Fig. 4 depicts the curves of the success rate and the average time consumption w.r.t. the value of *ColGen*. In this figure, the “o” indicates the success rate, and “◊” refers to the average time consumption (the same symbol has the same meaning in the following). In this figure, the success rate is the maximum when *ColGen* is equal to 310, and thereafter it changes little with the value of *ColGen*. In addition, the average time consumption reaches the minimum value when *ColGen* is 480. In view of these, it is rational to set the value of *ColGen* as 480.

To determine the optimal value of *Np*, we adopt the same set of target paths and parameter settings as those in the above-mentioned experiments except for setting the value of *ColGen* as 480. In the experiments, we set the value of *Np* from 1 to 150 with the step size of 1, and determine its optimal value using the similar steps and criteria for determining the optimal value of *ColGen*.

Fig. 5 depicts the curves of the success rate and the average time consumption w.r.t. the value of *Np*. Fig. 5 reports

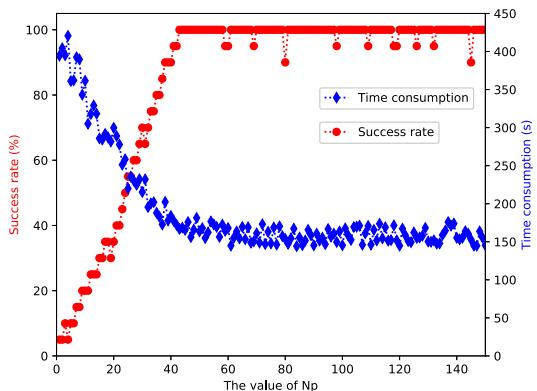


Fig. 5. The curves of the success rate and the average time consumption w.r.t. the value of *Np*.

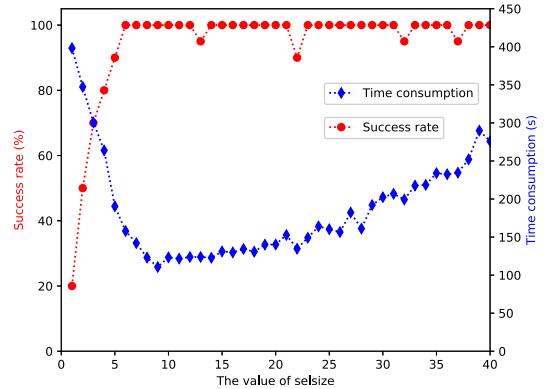


Fig. 6. The curves of the success rate and the average time consumption w.r.t. the value of *selsize*.

that the success rate reaches the maximum value once *Np* is 43, and thereafter it varies little as *Np* increases. For the average time consumption, it is the minimum in case of the value of *Np* being 56. Therefore, we set the optimal value of *Np* as 56.

In order to determine the optimal number of selecting superior individuals, we adopt the same basis target path set as that in the above-mentioned experiments except for the value of *ColGen* and *Np* being 480, and 56, respectively. Besides, we select 1 to 40 individuals sorted by the estimated fitness to execute *Convex*, and determine its optimal value using the similar steps and criteria for determining the optimal value of *ColGen*.

Fig. 6 depicts the curves of the success rate and the average time consumption w.r.t. the value of *selsize*. From this figure, the success rate is the maximum on the circumstance of *selsize* being larger than or equal to 6, and thereafter it varies little with the value of *selsize*. Regarding the average time consumption, it achieves the minimum value when *selsize* is 10. Therefore, it is appropriate to set the value of *selsize* as 10.

It should be noted that each method is run 20 times independently, and each run is to seek all desired test data that cover the basis path set of an MPI program. As a result, if test data covering one or more paths in the path set cannot be generated, this run will be deemed unsuccessful, thereby reducing the success rate and forming the outliers of the success rate in Figs. 4, 5, and 6.

For each program under test, we can obtain its optimal parameters using the above method, listed in Table 2. In addition, considering that each run of the proposed method for tuning the parameters of an MPI program under test is

TABLE 2
The Optimal Values of Key Parameters

Program	<i>Np</i>	<i>ColGen</i>	<i>selsize</i>
<i>Convex</i>	56	480	10
<i>QR_value</i>	49	430	8
<i>Cjacobi</i>	77	570	9
<i>Heat</i>	72	550	8
<i>DepSolver</i>	81	640	11
<i>Kfray</i>	86	660	10
<i>ClustalW</i>	93	630	13
Average	73	566	10

TABLE 3
Average Time Consumption of Different Surrogate Models

Program	B	Proposed method/s	RBFN-S/s	Reduction rate/%
<i>Convex</i>	15	124.2	192.3	35.4
<i>QR_value</i>	11	207.9	403.7	48.5
<i>Cjacobi</i>	17	258.6	387.1	33.2
<i>Heat</i>	19	137.5	254.2	45.9
<i>DepSolver</i>	24	2605.7	4991.8	47.8
<i>Kfray</i>	33	2827.5	7382.5	61.7
<i>ClustalW</i>	27	3592.8	9606.4	62.6
<i>Average</i>	20.9	1393.5	3316.8	47.9

probably time-consuming, we pick up one universal set of parameters consisting in the average of each parameter at the last row in Table 2, i.e., $ColGen = 566$, $Np = 73$, and $selsize = 10$, so as to improve the generalization of the proposed method and facilitate the use of other testers.

4.5 The Experimental Results and Analysis

(1) Answering RQ1

When answering RQ1, we generate test data using the proposed method and RBFN-S to cover the target path set, respectively, and calculate the average time consumption of 20 runs. To further determine the difference in efficiency between the above two methods, the average number of evaluated individuals need to be compared. Table 3 lists the average time consumption and the reduction rate. In this table, columns 3 and 4 are the average time consumption using the proposed method and RBFN-S, respectively. Column 5 is the reduction rate, calculated by $(v - v^*)/v \times 100\%$, where v^* and v are the average time consumption of the proposed method and RBFN-S, respectively.

We can see from Table 3 that, (1) for each program under test, the average time consumption of the proposed method is smaller than that of RBFN-S, where *Convex* has the smallest average time consumption, and the above two methods take 124.2 s and 192.3 s, respectively. The average time consumption of *ClustalW* is the biggest, where the two methods take 3592.8 s and 9606.4 s, respectively. For all the seven programs, the average time consumptions of the two methods are 1393.5 s and 3316.8 s, respectively, and (2) the reduction rate on the average time consumption is different for different programs, where *ClustalW* has the biggest reduction rate, which is up to 62.6 percent, and *Cjacobi* has the smallest that, which is 33.2 percent. For all the programs, the average reduction rate between the above methods is equal to 47.9 percent. The experimental results show that

TABLE 4
Average Number of Evaluated Individuals
of Different Surrogate Models

Program	B	Proposed method	RBFN-S	Reduction rate/%
<i>Convex</i>	15	39013.8	60580.4	35.6
<i>QR_value</i>	11	67133.4	133466.0	49.7
<i>Cjacobi</i>	17	87243.7	129441.7	32.6
<i>Heat</i>	19	66358.2	124733.5	46.8
<i>DepSolver</i>	24	98126.9	186199.1	47.3
<i>Kfray</i>	33	101512.3	264354.9	61.6
<i>ClustalW</i>	27	116718.5	317169.8	63.2
<i>Average</i>	20.9	82301.0	173706.5	48.1

TABLE 5
P-Values of the Student's t-Test

Program	Time consumption	# of evaluated individuals
<i>Convex</i>	0.003	0.001
<i>QR_value</i>	< 0.001	< 0.001
<i>Cjacobi</i>	0.005	0.007
<i>Heat</i>	< 0.001	< 0.001
<i>DepSolver</i>	< 0.001	< 0.001
<i>Kfray</i>	< 0.001	< 0.001
<i>ClustalW</i>	< 0.001	< 0.001

the proposed method can greatly reduce the time consumption when generating test data.

Table 4 lists the average number of evaluated individuals and the reduction rate. The meaning of each column in this table can be similarly understood according to Table 3. It can be seen from Table 4 that, (1) for each program, the average number of evaluated individuals in the proposed method is smaller than that in RBFN-S, where the average number of evaluated individuals for *Convex* is the smallest, 39013.8 and 60580.4, respectively. The average number of evaluated individuals for *ClustalW* is the biggest, where the two methods evaluate 116718.5 and 317169.8 individuals, respectively. For all the programs, the average numbers of evaluated individuals are 82301.0 and 173706.5, respectively, and (2) the reduction rate on the average number of evaluated individuals is different for different programs. *ClustalW* corresponds to the maximum reduction rate, which is up to 63.2 percent, and *Cjacobi* is the smallest, which is 32.6 percent. For all the seven programs, the average reduction rate is 48.1 percent. It is clear that the number of evaluated individuals can be greatly reduced using the proposed method.

From Tables 3 and 4, we can see that the number of evaluated individuals is positively correlated with the average time consumption for each program, namely the more the number of evaluated individuals is, the more the average time consumption is. Correspondingly, the average of each indicator of each program has the above relationship.

To check whether the two methods have a significant difference in terms of the above indicators or not, we conduct the student's t-test, with the experimental results being listed in Table 5.

Table 5 shows that, for all the MPI programs under test, there is significant differences in the time consumption and the number of evaluated individuals between the proposed method and RBFN-S. Together the average values in the last row of Tables 3 and 4, the efficiency of generating test data using the proposed method is significantly higher than that using RBFN-S.

For the proposed method and RBFN-S, we compare their success rate. Table 6 lists the success rate and percentage difference between the two methods. In this table, columns 3 and 4 show the success rate of the above two methods, respectively. Column 5 is the percentage difference, calculated by $u^* - u$, where u^* and u are the success rate using the two methods, respectively. It should be noted that if we employ the proposed method or RBFN-S to generate all desired test data covering the basis path set of an MPI program, this run will be deemed to be successful.

TABLE 6
Success Rate of of Different Surrogate Models

Program	$ B $	Proposed method / %	RBFN-S / %	Difference / %
Convex	15	100	85	15
QR_value	11	95	75	20
Cjacobi	17	100	90	10
Heat	19	100	80	20
DepSolver	24	95	90	5
Kfray	33	100	80	20
ClustalW	27	85	75	10
Average	20.9	96.4	82.1	14.3

TABLE 7
Average Time Consumption Between the Proposed Method and PSO-W

Program	$ B $	Proposed method / s	PSO-W / s	Reduction rate / %
Convex	15	124.2	158.4	21.6
QR_value	11	207.9	331.2	37.2
Cjacobi	17	258.6	322.8	19.9
Heat	19	137.5	194.7	29.4
DepSolver	24	2605.7	3772.1	30.9
Kfray	33	2827.5	4932.9	42.7
ClustalW	27	3592.8	6336.5	43.3
Average	20.9	1393.5	2292.7	32.1

TABLE 8
Average Number of Evaluated Individuals Between the Proposed Method and PSO-W

Program	$ B $	Proposed method	PSO-W	Reduction rate / %
Convex	15	39013.8	49884.7	21.8
QR_value	11	67133.4	106692.6	37.1
Cjacobi	17	87243.7	109853.4	20.6
Heat	19	66358.2	95667.3	30.6
DepSolver	24	98126.9	142565.2	31.2
Kfray	33	101512.3	175908.6	42.3
ClustalW	27	116718.5	207652.8	43.8
Average	20.9	82301.0	126889.2	32.5

From Table 6, we can see that, (1) for each program, the success rate using the proposed method is higher than that using RBFN-S, and Convex, Cjacobi, Heat, and Kfray have the highest success rate using the proposed method, which are 100 percent, whereas those of these programs using RBFN-S are 85, 90, 80, and 80 percent, respectively. For all the seven programs, the success rates of the above two methods are 96.4 and 82.1 percent, respectively, and (2) different programs have different percentage differences. The percentage differences of QR_value, Heat and Kfray are the biggest, which are 20 percent, and DepSolver has the smallest percentage difference, which is 5 percent. For all the seven programs, the average percentage difference is 14.3 percent. In view of the above analysis, we easily conclude that the proposed method can effectively generate test data that cover the target paths.

From the above experimental results and analysis, we can draw the following conclusion: the proposed method can generate test data with better performance in effectiveness and efficiency than the comparative one, meaning the advantageous of using the constructed ESM to estimate the fitness of an individual.

TABLE 9
P-Values of the Student's t-Test

Program	Time consumption	# of evaluated individuals
Convex	0.079	0.075
QR_value	0.006	0.006
Cjacobi	0.138	0.132
Heat	0.017	0.014
DepSolver	0.012	0.009
Kfray	< 0.001	0.001
ClustalW	< 0.001	< 0.001

TABLE 10
Success Rate Between the Proposed Method and PSO-W

Program	$ B $	Proposed method / %	PSO-W / %	Difference / %
Convex	15	100	100	0
QR_value	11	95	85	10
Cjacobi	17	100	100	0
Heat	19	100	95	5
DepSolver	24	95	90	5
Kfray	33	100	100	0
ClustalW	27	85	85	0
Average	20.9	96.4	93.6	2.9

(2) Answering RQ2

To answer RQ2, the proposed method and PSO-W are utilized to generate test data that cover the target paths, respectively. Tables 7, 8, 9, and 10 list the related experimental results, and the meaning of each column in these tables can be similarly understood according to Tables 3, 4, 5, and 6, respectively.

From Table 7, we can summarize that, (1) for each program, the average time consumption using the proposed method is smaller than that using PSO-W, where Convex has the smallest average time consumption, and 124.2 s and 158.4 s, respectively. The biggest average time consumption is ClustalW, where 3592.8 s and 6336.5 s are spent by the two methods, respectively. For all the seven programs, the average time consumptions using the two methods are 1393.5 s and 2292.7 s, respectively, and (2) different programs have different values of the reduction rate, where the reduction rate of ClustalW is up to 43.3 percent, which is the biggest, whereas the reduction rate of Cjacobi, only 19.9 percent, is the smallest. For all the programs, 32.1 percent average reduction rate is achieved. The experimental results clearly show that the proposed method can greatly reduce the time consumption when generating test data.

Table 8 lists the average number of evaluated individuals and the reduction rate of the two methods. Table 8 reports that, (1) for each program, the average number of evaluated individuals using the proposed method is smaller than that using PSO-W, where the smallest average number of evaluated individuals is got by Convex, 39013.8 and 49884.7, respectively. The biggest average number of evaluated individuals is got by ClustalW, where 116718.5 and 207652.8 individuals are evaluated by the two methods, respectively. For all the programs, the average numbers of evaluated individuals are 82301.0 and 126889.2, respectively, and (2) the reduction rate of different programs is different. The reduction rate of ClustalW is up to 43.8 percent, which is the maximum, whereas that of Cjacobi, only 20.6 percent, is the

TABLE 11
Average Time Consumption Between the Proposed Method and the CGA

Program	$ B $	Proposed method/s	CGA/s	Reduction rate/%
<i>Convex</i>	15	124.2	180.3	31.1
<i>QR_value</i>	11	207.9	337.5	38.4
<i>Cjacobi</i>	17	258.6	346.2	25.3
<i>Heat</i>	19	137.5	212.9	35.4
<i>DepSolver</i>	24	2605.7	4366.1	40.3
<i>Kfray</i>	33	2827.5	5183.7	45.5
<i>ClustalW</i>	27	3592.8	6749.4	46.8
Average	20.9	1393.5	2482.3	37.5

TABLE 12
Average Number of Evaluated Individuals Between the Proposed Method and the CGA

Program	$ B $	Proposed method	CGA	Reduction rate/%
<i>Convex</i>	15	39013.8	56153.3	30.5
<i>QR_value</i>	11	67133.4	109563.7	38.7
<i>Cjacobi</i>	17	87243.7	117534.0	25.8
<i>Heat</i>	19	66358.2	104012.6	36.2
<i>DepSolver</i>	24	98126.9	165203.4	40.6
<i>Kfray</i>	33	101512.3	185369.4	45.2
<i>ClustalW</i>	27	116718.5	217581.8	46.4
Average	20.9	82301.0	136488.3	37.6

minimum. For all the programs, the average reduction rate is 32.5 percent. Therefore, the number of estimated individuals can be greatly reduced using the proposed method.

To check whether the two methods have a significant difference in terms of the above indicators or not, the student's t-test is conducted, with Table 9 listing the experimental results.

From Table 9, we can see that, (1) for *Convex* and *Cjacobi*, there is no significant differences in the time consumption and the number of evaluated individuals between the proposed method and PSO-W, and (3) apart from the above programs, the proposed method differs significantly PSO-W, which is deduced from their P-values smaller than 0.05. Further, considering the average values at the last row of Tables 7 and 8, it is clear that the efficiency using the proposed method is significantly high than that using PSO-W.

For the proposed model and PSO-W, the success rate is compared. Table 10 lists the success rate and percentage difference between the proposed method and PSO-W.

Table 10 shows that, (1) for each program, the success rate using the proposed method is higher than or equal to that using PSO-W, and all the success rate of *Convex*, *Cjacobi*, *Heat*, and *Kfray* are 100 percent, whereas the success rates of these programs using PSO-W are 100, 100, 95, and 100 percent, respectively. For all the seven programs, the success rate of the above two methods are 96.4 and 93.6 percent, respectively, and (2) different programs have different values of the percentage difference. The percentage differences of *QR_value* is the biggest, 10 percent, and *Convex*, *Cjacobi*, *Kfray*, and *ClustalW* have the smallest percentage difference, which are zero. For all the seven programs, the average percentage difference is 2.9 percent. As a result, the proposed method can effectively generate test data that cover the target paths.

TABLE 13
P-Values of the Student's t-Test

Program	Time consumption	# of evaluated individuals
<i>Convex</i>	0.010	0.016
<i>QR_value</i>	0.005	0.003
<i>Cjacobi</i>	0.034	0.029
<i>Heat</i>	0.002	0.001
<i>DepSolver</i>	0.001	< 0.001
<i>Kfray</i>	< 0.001	< 0.001
<i>ClustalW</i>	< 0.001	< 0.001

TABLE 14
Success Rate Between the Proposed Method and the CGA

Program	$ B $	Proposed method/%	CGA/%	Difference/%
<i>Convex</i>	15	100	95	5
<i>QR_value</i>	11	95	85	10
<i>Cjacobi</i>	17	100	95	5
<i>Heat</i>	19	100	90	10
<i>DepSolver</i>	24	95	85	10
<i>Kfray</i>	33	100	95	5
<i>ClustalW</i>	27	85	80	5
Average	20.9	96.4	89.3	7.1

Through the experimental results and analysis of this group of experiments, we can draw the following conclusion: the proposed method can generate test data with better performance in effectiveness and efficiency than the comparative one, indicating that it is advantageous of selecting superior individuals to execute an MPI program.

(3) Answering RQ3

When answering RQ3, the proposed method and CGA are utilized to generate test data to cover the target path set, respectively. Tables 11, 12, 13, and 14 list the related experimental results. In addition, the meaning of each column in Tables 11, 12, 13, and 14 can be similarly understood according to Tables 3, 4, 5, and 6, respectively.

Table 11 shows that, (1) for each program, the average time consumption of the proposed method is smaller than that of CGA, where *Convex* has the smallest average time consumption, 124.2 s and 180.3 s, respectively. The biggest average time consumption is *ClustalW*, 3592.8 s and 6749.4 s are spent by the two methods, respectively. For all the seven programs, the average time consumptions of the two methods are 1127.6s and 2482.3 s, respectively, and (2) different programs have different reduction rates, where the reduction rate of *ClustalW* is up to 46.8 percent, the biggest one, in contrast, the reduction rate of *Cjacobi* is 25.3 percent, which is the smallest. For all the programs, 37.5 percent average reduction rate is achieved. The experimental results significantly reflect that the proposed method can greatly reduce the time consumption.

Table 12 lists the average number of evaluated individuals and the reduction rate of the two methods. From Table 12, (1) for each program, the average number of evaluated individuals of the proposed method is smaller than that of CGA, where the smallest average number of evaluated individuals is got by *Convex*, 39013.8 and 56153.3, respectively, whereas the biggest average number of evaluated individuals is got by *ClustalW*, 116718.5 and 217581.8, respectively. For all the programs, the average numbers of

evaluated individuals are equal to 82301.0 and 136488.3, respectively, and (2) the reduction rate is different for different programs. The reduction rate of *ClustalW* is up to 46.4 percent, the maximum one, in contrast, the reduction rate of *Cjacobi*, 25.8 percent, is the minimum. For all the programs, the average reduction rate is 37.6 percent. Therefore, the number of evaluated individuals can be greatly reduced using the proposed method.

To check whether the two methods have a significant difference in terms of the above indicators or not, the student's t-test is conducted, with the experimental result being listed in Table 13.

According to Table 13, for all the seven programs, the time consumption and the number of evaluated individuals of the proposed method is significantly different from those of CGA. Together with the average values at the last row of Tables 11 and 12, the proposed method is significantly efficient than CGA.

Table 14 represents the success rate and percentage difference between the proposed method and CGA.

From Table 14, (1) for each program, the success rate of the proposed method is higher than that of CGA, and all the success rates of *Convex*, *Cjacobi*, *Heat*, and *Kfray* are 100, whereas the success rates of these programs using CGA are 95, 95, 90, and 95 percent, respectively. For all the seven programs, the success rate of the above two methods are 96.4 and 89.3 percent, respectively, and (2) the percentage differences of different programs are different. *QR_value*, *Heat*, and *DepSolver* have the biggest percentage difference, 10 percent, whereas *Convex*, *Cjacobi*, *Kfray*, and *ClustalW* have the smallest percentage difference, 5 percent. For all the seven programs, the average percentage difference is 7.1 percent. Therefore, the proposed method can effectively generate test data.

From the experimental results and analysis of the last group of experiments, we can conclude that the proposed method is beneficial to improve the efficiency of generating test data.

To summarize, (1) the MPI-based ESM in the proposed method accurately estimates the fitness of an individual, (2) the proposed method has a low computational cost by selecting superior individuals to execute a program, and (3) it is beneficial to improve the efficiency of generating test data. In addition, considering that we draw the above conclusions for representative MPI programs, the results can be extensible to other programs.

5 THE THREATS TO VALIDITY

We analyze the threats that possibly affect the experimental results, and divide them into the following two catalogues: internal and external threats.

5.1 Internal Threats

When executing Algorithm 3, if the value of *ColGen* is too small, an insufficient training set will be formed. At this time, the estimation model trained based on the training set will have poor generalization performance. On the contrary, it will pose a large computational cost resulted from executing programs to calculate the fitness of each individual, which is oppose to the purpose of this paper. To minimize this threat, test data from 1 to 1,200 generations are collected

and form the training set by combining them with their fitness. When the proposed method achieves the best performance in effectiveness and efficiency, the value of *ColGen* will be optimal. In the experiments, its value is set after many try and error, and different values are set for different programs.

When executing Algorithm 4, all the RBFNs will have an inadequate difference if the value of *Np* is too small. At this time, the ESM may result in the over-fitting or under-fitting problem. Conversely, a large computational resources will be required to implement the ESM. In order to reduce this threat, *Np* is set from 1 to 150, and the ESM with the corresponding size is trained based on the formed training set. The optimal value of *Np* will be obtained in case of the time consumption and the success rate of the proposed method being the best. Similarly, in the experiments, its value is obtained after a number of try and error, and different programs require different values.

During executing Algorithm 5, it will be harmful to reducing the computational costs of generating test data if too many individuals are selected. Contrarily, the success rate of generating test data cannot be guaranteed. In order to alleviate the threat, 1 to 40 individuals sorted by the estimated fitness are selected to execute a program, and the optimal number of estimated individuals is determined on the circumstance of the time consumption and the success rate of the proposed method achieving their best values.

Although the methods of setting parameters may not be optimal in Section 4.4, 20 runs are conducted for each program, with the purpose of obtaining their values as optimal as possible.

The parameter settings of CGA and PSO also affect the experimental results, which was discussed in [8], [39]. In order to reduce such threat, the parameters of CGA and PSO are cited from [8] and [39], respectively. It can be seen from the experimental results that these parameter settings are reasonable.

5.2 External Threats

The selected programs under test generally have an important influence on the generalization of the experimental conclusions. If these programs are not representative, the experimental conclusions will be hardly to be directly generalized to more complex programs. To alleviate this threat as much as possible, we select programs with various numbers of processes, communication primitives, and LOC in the experiments, suggesting their good representativeness.

The selected target paths also impact the generalization of the experimental conclusions. To have good performance in generalization, we propose a novel method of selecting the target paths of each program, through which any statement and branch of the program under test can be included in at least one target path, indicating their good performance in test adequacy.

Besides, the time consumption is a vital indicator to reflect the efficiency in generating test data. In the experiments, the time consumption has a close relation with the configuration of the implementation environment, such as the number and the type of computers and processors, as well as the operating system. It is clear that environments with different configurations will have different time

consumptions when fulfilling the same task. To alleviate this threat, we run the same task for 20 times in the same environment for each program, record all the time consumptions, and calculate their average value.

6 CONCLUSION AND FUTURE WORK

We have proposed a method of using an ESM to make full use of test data generated during the evolution in this paper. In the proposed method, PSO is employed to generate a number of test inputs, which are combined with their fitness to form a training set. Following that, an ESM is trained using the training set, and utilized to estimate the fitness of each individual. Based on the estimation, a small number of superior individuals are selected to execute the program, with the purpose of achieving their real fitness for the subsequent evolution.

We have also applied the proposed method to seven complex MPI programs and compared with several state-of-the-art approaches. The experimental results show that the proposed method can improve the test efficiency.

It should be pointed out that although the proposed method is advantages in terms of the time consumption and the success rate, various attempts are required to further improve the efficiency of the proposed method, such as the choice of the ESM composed of other BSMs, which will be our research topic in the future.

ACKNOWLEDGMENTS

This article was jointly supported by National Key Research and Development Program of China(2018YFB1003802-01), and National Natural Science Foundation of China (61773384, 61763026, 61673404, 61573362, 61503220).

REFERENCES

- [1] G. Hager and G. Wellein, *Introduction to High Performance Computing for Scientists and Engineers*. Boca Raton, FL, USA: CRC Press, 2010.
- [2] W. Gropp, E. Lusk, N. Doss, and A. Skjellum, "A high-performance, portable implementation of the MPI message passing interface standard," *Parallel Comput.*, vol. 22, no. 6, pp. 789–828, 1996.
- [3] D. Akhmetova *et al.*, "Interoperability of GASPI and MPI in large scale scientific applications," in *Proc. Int. Conf. Parallel Process. Appl. Math.*, 2017, pp. 277–287.
- [4] B. Sun, J. Wang, D. Gong, and T. Tian, "Scheduling sequence selection for generating test data to cover paths of MPI programs," *Inf. Softw. Technol.*, vol. 114, pp. 190–203, 2019.
- [5] A. S. Ghiduk, "Automatic generation of basis test paths using variable length genetic algorithm," *Inf. Process. Lett.*, vol. 114, no. 6, pp. 304–316, 2014.
- [6] S. Souza, S. R. Vergilio, P. Souza, A. Simao, and A. C. Hausen, "Structural testing criteria for message-passing parallel programs," *Concurrency Comput., Pract. Experience*, vol. 20, no. 16, pp. 1893–1916, 2008.
- [7] P. S. Souza, S. R. Souza, and E. Zaluska, "Structural testing for message-passing concurrent programs: An extended test model," *Concurrency Comput., Pract. Experience*, vol. 26, no. 1, pp. 21–50, 2014.
- [8] T. Tian and D. Gong, "Test data generation for path coverage of message-passing parallel programs based on co-evolutionary genetic algorithms," *Autom. Softw. Eng.*, vol. 23, no. 3, pp. 469–500, 2016.
- [9] H. Tong, C. Huang, J. Liu, and X. Yao, "Voronoi-based efficient surrogate-assisted evolutionary algorithm for very expensive problems," *IEEE Congr. Evol. Comput. (CEC)*, pp. 1996–2003, 2019.
- [10] C. Sun, J. Zeng, J. Pan, S. Xue, and Y. Jin, "A new fitness estimation strategy for particle swarm optimization," *Inf. Sci.*, vol. 221, pp. 355–370, 2013.
- [11] A. Xiao, B. Wang, C. Sun, S. Zhang, and Z. Yang, "Fitness estimation based particle swarm optimization algorithm for layout design of truss structures," *Math. Problems Eng.*, vol. 2014, 2014, Art. no. 671872.
- [12] Z. Zhou, Y. S. Ong, M. H. Nguyen, and D. Lim, "A study on polynomial regression and Gaussian process global surrogate model in hierarchical surrogate-assisted evolutionary algorithm," in *Proc. IEEE Congress Evol. Comput.*, 2005, vol. 3, pp. 2832–2839.
- [13] B. Liu, Q. Zhang, and G. G. Gielen, "A Gaussian process surrogate model assisted evolutionary algorithm for medium scale expensive optimization problems," *IEEE Trans. Evol. Comput.*, vol. 18, no. 2, pp. 180–192, Apr. 2014.
- [14] H. Xiang, Y. Li, H. Liao, and C. Li, "An adaptive surrogate model based on support vector regression and its application to the optimization of railway wind barriers," *Structural Multidisciplinary Optim.*, vol. 55, no. 2, pp. 701–713, 2017.
- [15] H. Yu, Y. Tan, J. Zeng, C. Sun, and Y. Jin, "Surrogate-assisted hierarchical particle swarm optimization," *Inf. Sci.*, vol. 454, pp. 59–72, 2018.
- [16] C. Sun, Y. Jin, J. Zeng, and Y. Yu, "A two-layer surrogate-assisted particle swarm optimization algorithm," *Soft Comput.*, vol. 19, no. 6, pp. 1461–1475, 2015.
- [17] T. G. Dietterich, "Ensemble methods in machine learning," in *Proc. Int. Workshop Multiple Classifier Syst.*, 2000, pp. 1–15.
- [18] W. D. Gropp, W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming With the Message-Passing Interface*, vol. 1. Cambridge, MA, USA: MIT Press, 1999.
- [19] A. Panichella, F. M. Kifetew, and P. Tonella, "Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets," *IEEE Trans. Softw. Eng.*, vol. 44, no. 2, pp. 122–158, Feb. 2018.
- [20] N. Gupta, V. Yadav, and M. Singh, "Automated regression test case generation for web application: A survey," *ACM Comput. Surv.*, vol. 51, no. 4, 2018, Art. no. 87.
- [21] S. Scalabrino *et al.*, "OCELOT: A search-based test-data generation tool for C," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 868–871.
- [22] X.-W. Lv, S. Huang, Z.-W. Hui, and H.-J. Ji, "Test cases generation for multiple paths based on PSO algorithm with metamorphic relations," *IET Softw.*, vol. 12, no. 4, pp. 306–317, 2018.
- [23] A. Jannesari and F. Wolf, "Automatic generation of unit tests for correlated variables in parallel programs," *Int. J. Parallel Program.*, vol. 44, no. 3, pp. 644–662, 2016.
- [24] L. Bo, S. Jiang, J. Qian, R. Wang, and X. Wang, "Efficient test case generation for thread-safe classes," *IEEE Access*, vol. 7, pp. 26984–26995, 2019.
- [25] H. Yue, P. Wu, T.-Y. Chen, and Y. Lv, "Input-driven active testing of multi-threaded programs," in *Proc. Asia-Pacific Softw. Eng. Conf.*, 2015, pp. 246–253.
- [26] B. K. Sahoo and M. Ray, "A comparative study on test case generation of concurrent programs," *World J. Eng. Technol.*, vol. 4, no. 02, 2016, Art. no. 273.
- [27] T. Tian, D. Gong, F.-C. Kuo, and H. Liu, "Genetic algorithm based test data generation for MPI parallel programs with blocking communication," *J. Syst. Softw.*, vol. 155, pp. 130–144, 2019.
- [28] R. Chang, S. Sankaranarayanan, G. Jiang, and F. Ivancic, "Software testing using machine learning," U.S. Patent 8,924,938, Dec. 30, 2014.
- [29] H. Christiansen and C. M. Dahmke, "A machine learning approach to test data generation: A case study in evaluation of gene finders," in *Proc. Int. Workshop Mach. Learn. Data Mining Pattern Recognit.*, 2007, pp. 742–755.
- [30] D. Zhang, *Advances in Machine Learning Applications in Software Engineering*. Pennsylvania, USA: IGI Global, 2006.
- [31] Y. Freund, R. Iyer, R. E. Schapire, and Y. Singer, "An efficient boosting algorithm for combining preferences," *J. Mach. Learn. Res.*, vol. 4, no. Nov, pp. 933–969, 2003.
- [32] L. Breiman, "Bagging predictors," *Mach. Learn.*, vol. 24, no. 2, pp. 123–140, 1996.
- [33] Z.-H. Zhou, *Ensemble Methods: Foundations and Algorithms*. London, U.K.: Chapman and Hall/CRC, 2012.
- [34] M. C. Chewry and H. P. Wynn, "Maximum entropy sampling," *J. Appl. Statist.*, vol. 14, no. 2, pp. 165–170, 1987.
- [35] R. G. Regis, "Evolutionary programming for high-dimensional constrained expensive black-box optimization using radial basis functions," *IEEE Trans. Evol. Comput.*, vol. 18, no. 3, pp. 326–347, Jun. 2014.

- [36] S. Z. Martínez and C. A. C. Coello, "MOEA/D assisted by RBF networks for expensive multi-objective optimization problems," in *Proc. 15th Annu. Conf. Genetic Evol. Comput.*, 2013, pp. 1405–1412.
- [37] H. Wang, Y. Jin, C. Sun, and J. Doherty, "Offline data-driven evolutionary optimization using selective surrogate ensembles," *IEEE Trans. Evol. Comput.*, vol. 23, no. 2, pp. 203–216, Apr. 2019.
- [38] B. Efron and R. J. Tibshirani, *An Introduction to the Bootstrap*. Boca Raton, FL, USA: CRC Press, 1994.
- [39] A. Windisch, S. Wappeler, and J. Wegener, "Applying particle swarm optimization to software testing," in *Proc. 9th Annu. Conf. Genetic Evol. Comput.*, 2007, pp. 1121–1128.
- [40] G. Chen, H. An, L. Chen, Q. Zheng, and J. Shan, *Parallel Algorithm Practice*, Beijing Higher Education Press, 2004.
- [41] M. Müller, B. de Supinski, G. Gopalakrishnan, T. Hilbrich, and D. Lecomber, "Dealing with MPI bugs at scale: Best practices, automatic detection, debugging, and formal verification," Slides presented in this tutorial, integrating presentations from Dresden, Allinea, LLNL, and Utah, 2011. [Online]. Available: http://www.cs.utah.edu/fv/publications/sc11_with_handson.pptx
- [42] H. Yu, "Combining symbolic execution and model checking to verify MPI programs," in *Proc. IEEE/ACM 40th Int. Conf. Softw. Eng., Companion*, 2018, pp. 527–529.
- [43] G. J. Myers, T. Badgett, T. M. Thomas, and C. Sandler, *The Art of Software Testing*, vol. 2. Hoboken, NJ, USA: Wiley, 2004.
- [44] C. Igel, V. Heidrich-Meisner, and T. Glasmachers, "Shark," *J. Mach. Learn. Res.*, vol. 9, pp. 993–996, 2008.
- [45] J. Yan and J. Zhang, "An efficient method to generate feasible paths for basis path testing," *Inf. Process. Lett.*, vol. 107, no. 3/4, pp. 87–92, 2008.
- [46] R. Zhou and E. A. Hansen, "Breadth-first heuristic search," *Artif. Intell.*, vol. 170, no. 4/5, pp. 385–408, 2006.
- [47] Z. Xu and J. Zhang, "A test data generation tool for unit testing of C programs," in *Proc. 6th Int. Conf. Quality Softw.*, 2006, pp. 107–116.
- [48] M. Berkelaar *et al.*, "Ipsolve: Open source (mixed-integer) linear programming system," 2004. [Online]. Available: <http://ipsolve.sourceforge.net/5.1/>
- [49] J. R. Quinlan *et al.*, "Bagging, boosting, and C4.5," in *Proc. 13th Nat. Conf. Artif. Intell.*, 1996, pp. 725–730.



Baicai Sun received the MS degree in control engineering from Qufu Normal University, China, in 2016. He is currently working toward the PhD degree with the School of Information and Control Engineering, China University of Mining and Technology, China. His research interests include search-based software engineering, surrogate-assisted evolutionary optimization, and machine learning. He has published some research papers in international journals as the first author or the corresponding author, including TSE, TOSEM and ISTetc.



Dunwei Gong (Member, IEEE) received the BSc degree in mathematics from the China University of Mining and Technology, Xuzhou, China, in 1992, the MSc degree in control theory and applications from Beihang University, Beijing, China, in 1995, and the PhD degree in control theory and control engineering from the China University of Mining and Technology, Xuzhou, China, in 1999, respectively. He is currently a professor in computational intelligence and the director of the Centre for Intelligent Optimization and Control, School of Information and Control Engineering, China University of Mining and Technology, Xuzhou, China. He is a guest professor with the School of Information Engineering, Xiangtan University, Xiangtan, China. He has more than 180 publications. His current research interests include computation intelligence in many-objective optimization, dynamic and uncertain optimization, as well as applications in software engineering, scheduling, path planning, big data processing and analysis.



Tian Tian received the PhD degree in control theory and control engineering from the China University of Mining and Technology, China, in 2014. She is currently with the School of Computer Science and Technology, Shandong Jianzhu University, China. Her research interests include genetic algorithms and software testing.



Xiangjuan Yao received the PhD degree in control theory and control engineering from the China University of Mining and Technology, China, in 2011. She is currently a professor with the School of Mathematics, China University of Mining and Technology, China. Her main research interests include intelligence optimization and search-based software testing.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

What Leads to a Confirmatory or Disconfirmatory Behavior of Software Testers?

Iflaah Salman^{ID}, Pilar Rodríguez^{ID}, Burak Turhan^{ID}, Member, IEEE,
Ayşe Tosun^{ID}, Member, IEEE, and Arda Güreller

Abstract—Background: The existing literature in software engineering reports adverse effects of confirmation bias on software testing. Confirmation bias among software testers leads to confirmatory behavior, which is designing or executing relatively more specification consistent test cases (confirmatory behavior) than specification inconsistent test cases (disconfirmatory behavior). **Objective:** We aim to explore the antecedents to confirmatory and disconfirmatory behavior of software testers. Furthermore, we aim to understand why and how those antecedents lead to (dis)confirmatory behavior. **Method:** We follow grounded theory method for the analyses of the data collected through semi-structured interviews with twelve software testers. **Results:** We identified twenty antecedents to (dis)confirmatory behavior, and classified them in nine categories. Experience and Time are the two major categories. Experience is a disconfirmatory category, which also determines which behavior (confirmatory or disconfirmatory) occurs first among software testers, as an effect of other antecedents. Time Pressure is a confirmatory antecedent of the Time category. It also contributes to the confirmatory effects of antecedents of other categories. **Conclusion:** The disconfirmatory antecedents, especially that belong to the testing process, e.g., test suite reviews by project team members, may help circumvent the deleterious effects of confirmation bias in software testing. If a team's resources permit, the designing and execution of a test suite could be divided among the test team members, as different perspectives of testers may help to detect more errors. The results of our study are based on a single context where dedicated testing teams focus on higher levels of testing. The study's scope does not account for the testing performed by developers. Future work includes exploring other contexts to extend our results.

Index Terms—Software testing, cognitive biases, confirmation bias, grounded theory, interviews

1 INTRODUCTION

CONFIRMATION bias is the cognitive tendency to look for evidence that confirms, rather than refutes, one's prior beliefs [1]. In software testing, confirmation bias occurs when developers or testers exercise a program with the data that is consistent with its specified behaviour instead of inconsistent data [2]. Confirmation bias leads to confirmatory behaviour by software testers during testing [3]. For example, if requirements specification state that ...*the phone number field accepts seven digits from 0 to 9*; a consistent test case would validate the behaviour of the field by providing in, e.g., 0123456 as an input test data. An inconsistent test

case would validate the field's behaviour with inconsistent data, e.g., a - a letter instead of a digit.

The higher the level of confirmation bias, the more adverse effects it has on software testing [4], [5], [6], [7]. For example, Çalikli and Bener observed a positive correlation between software defect density and confirmation bias levels of software developers [4], [8]. In their experiments, Teasley *et al.* observed that participants designed two to four times more positive test cases compared to negative test cases¹ (i.e., confirmation bias) [7]. Similarly, Causevic *et al.* also found a significant difference between the number of positive and negative test cases designed by the participants in an experimental study on test-driven development [9]. Salman *et al.*'s work also supports these findings, in their experiment, participants designed significantly more consistent test cases, with respect to provided specifications, compared to inconsistent test cases in performing functional testing [3].

Mohanani *et al.* found the primary studies that investigated the effects and antecedents to confirmation bias in software testing were all experiments [10]. For example, an experimental study observed that a lack of logical reasoning skills is an antecedent to confirmation bias [10]. The authors identified a need to conduct more qualitative research that explores how cognitive biases are manifested in the software engineering (SE) industry rather than only focusing on causal relationships. The primary studies on antecedents to

- Iflaah Salman is with M3S Group, University of Oulu, 90570 Oulu, Finland. E-mail: iflaah.salman@oulu.fi.
- Pilar Rodríguez is with M3S Group, University of Oulu, 90570 Oulu, Finland, and also with Escuela Técnica Superior de Ingenieros Informáticos, Universidad Politécnica de Madrid, 28040 Madrid, Spain. E-mail: pilar.rodriguez@upm.es.
- Burak Turhan is with the Faculty of Information Technology, Monash University, Clayton, VIC 3800, Australia, and also with M3S Group, University of Oulu, 90570 Oulu, Finland. E-mail: burak.turhan@monash.edu.
- Ayşe Tosun is with the Faculty of Computer and Informatics Engineering, İstanbul Technical University, 34467 İstanbul, Turkey. E-mail: tosunay@itu.edu.tr.
- Arda Güreller is with Ericsson, İstanbul, Turkey. E-mail: arda.gureller@ericsson.com.

Manuscript received 5 Sept. 2019; revised 13 Aug. 2020; accepted 19 Aug. 2020.

Date of publication 27 Aug. 2020; date of current version 18 Apr. 2022.

(Corresponding author: Iflaah Salman.)

Recommended for acceptance by E. Murphy-Hill.

Digital Object Identifier no. 10.1109/TSE.2020.3019892

¹ Positive/negative test case is another terminology for what we call consistent/inconsistent test case. We use the latter one for the remainder of this paper.

confirmation bias in software testing are limited in providing an insight into the phenomenon [10]. These findings, therefore, establish a need of not only to explore more antecedents to confirmation bias but also to understand why and how it occurs among software testers.

The goal of this paper is to explore antecedents that may lead to confirmation bias in software testing. The antecedents may belong to the working environment, could be part of the testing process or are personal attributes of software testers. We additionally aim to understand why and how these antecedents lead software testers to confirmatory behaviour. In order to address our objectives, we apply the Glaserian grounded theory to explore the phenomenon of confirmation bias among software testers. We conducted twelve semi-structured interviews with software testers to collect our data. They were all employees of the same company but worked in different projects.

We identified nine antecedents to confirmatory behaviour and eight to disconfirmatory behaviour. A disconfirmatory behaviour is contrasting to a confirmatory behaviour, i.e., it may mitigate confirmation bias. Additionally, three more antecedents were found that lead to both (confirmatory and disconfirmatory) behaviours by software testers. *Both* refers to the completeness of a test suite² that may also mitigate confirmation bias. Experience of testing in general and particular to the project are two major disconfirmatory antecedents. They determine the confirmatory or disconfirmatory behaviour of testers due to other antecedents. Project's testing experience also improves the completeness of a test suite. Time pressure is a major confirmatory antecedent for software testers. It contributes to promoting the confirmatory influence of other confirmatory antecedents. For example, time pressure promotes the confirmatory behaviour of a tester in case of a minor functional change (a confirmatory antecedent).

Our study contributes by generating a grounded theory that explains the phenomenon of confirmation bias among software testers. We also contribute with the identification of thirteen new antecedents, relative to the existing ones in the SE literature, that may lead to confirmation bias. We also provide a list of disconfirmatory antecedents that can be used by practitioners to alleviate confirmation bias.

Section 2 presents the related work and the conceptual background. Research Method is detailed in Section 3. The results are presented in Section 4, and discussed along with the validity threats in Section 5. Section 6 concludes the paper.

2 RELATED WORK AND BACKGROUND

Humans rely on simplifying heuristics for judgement of uncertain events instead of relying on formal logic [11]. These heuristics usually offer a workable solution, but may also lead to systematic errors in decision making, known as cognitive biases [11], [12]. The concept of cognitive biases was first introduced by Tversky and Kahneman in the early 1970s, and is defined as, "cognitive biases are cognitions or mental behaviours that prejudice decision quality in a significant number of decisions

for a significant number of people" [1, p. 59], [10], [12]. Cognitive biases are also referred to as judgement biases or decision biases [13]. The human mind is inherent to cognitive biases [1], [13], [14]. Kahneman *et al.* elaborate on why humans are incapable to recognise their own cognitive biases by referring to two modes of thinking; system-one and system-two, also referred to as the dual-process theory [14], [15]. System-one – intuitive, thinking is fast and effortless, which makes it more prone to biases [10], [16]. System-two – reflective, thinking is effortful, intentional and slow, therefore, less prone to biases [10], [16]. Thoughts are usually determined by system-one, humans are unaware of it because it is proficient in its operation [15], [16]. In addition to system-one, noisy information processing, emotion and social influences can also generate cognitive biases [10]. Cognitive biases also form *biasplexes* because some biases may overlap, interact and reinforce each other [17]. Therefore, when biases occur it is uncertain which one is the cause and which one is the effect [10].

Confirmation bias is a cognitive bias [1], [11]. Mohanani *et al.* categorise confirmation bias to the category of interest biases among the categories defined for cognitive biases in the SE discipline [10]. Confirmation bias negatively affects multiple areas of SE, e.g., maintenance [18], design [19] and testing [2], [20]. The software testing studies that investigated confirmation bias use different terminologies, e.g., positive test bias, but essentially refer to the same phenomenon of confirmation bias [3], [21].

The earliest (1993 – 1994) work exploring the impact of confirmation bias in software testing was conducted by Leventhal *et al.* and Teasley *et al.* [2], [6], [7]. These authors, in their family of experiments conducted with advanced testers (senior-level and graduate students in computer science), observed multiple factors that may cause the manifestation of positive test bias in functional software testing. The results showed that a higher level of expertise and completeness of specifications may cause less positive test bias [6], [7]. Another studied factor, error feedback (the effect of presence or absence errors), was not confirmed to cause the effect possibly due to the types of software used in the experiments [6].

The second era of focus begins from 2010 when multiple studies examined the effects of confirmation bias in software testing. Çalikli and Bener [5] and Çalikli *et al.* [22], in their series of experiments, assessed confirmation bias levels of the participants by deriving measures from psychological instruments, Wason's Rule Discovery and Selection Task. In an experiment with software engineers and graduate students, Çalikli *et al.* found that company culture affected the confirmation bias levels [23]. The authors also investigated the effects of logical reasoning skills acquired through education, experience and activeness in testing and development, job titles (tester, developer, analyst, researcher), development methods, company size (large, small and medium enterprises), educational background (undergraduate) and educational level (bachelor's, master's) [5], [8]. They found that confirmation bias levels were low due to logical reasoning skills and for those participants who were experienced but inactive in testing or development [5], [8]. The rest of the factors were not observed to affect confirmation bias levels except the job title - researcher [5], [8]. Çalikli and Bener related the lower levels of confirmation bias of researchers to their critical and analytical skills [5].

2. It refers to the completeness of design/execution in terms of consistent and inconsistent test cases.

In a test-driven development (TDD) experiment, carried out in the industry, Causevic *et al.* observed that participants created more positive test cases compared to negative test cases [9]. The experimenters also observed that negative test cases have a higher tendency of finding defects compared to positive test cases [9]. Eldh investigated whether negative testing reveals ‘real important faults’ of the system under test (SUT) by applying negative testing techniques referred to as ‘attacks’ by Whittaker *et al.* [24], [25]. The author found that negative testing could not find any major faults for the SUT, albeit notable ones [24]. Eldh attributed the findings to the high quality of the SUT and the types of the executed negative test cases [24].

According to Salman *et al.*, confirmation bias occurs when a software tester designs relatively more *consistent* test cases (consistent with the requirements specification) in comparison to *inconsistent* test cases [3]. An *inconsistent* test case validates a behaviour of the software application that is not explicit in the requirements specification or is an outside-of-the-box test case, within the context of the SUT [3]. In functional test case design, a consistent test case is an indication of a confirmatory behaviour; similarly, an inconsistent test case indicates a disconfirmatory behaviour on a tester’s end [3].

Multiple qualitative studies on cognitive biases have used interview data collection method for grounded theory and case studies. The objectives were to identify the occurrence of cognitive biases in the studied context, antecedents to cognitive biases and mitigation techniques for cognitive biases [26], [27], [28], [29], [30]. For example, Cunha *et al.* conducted semi-structured interviews for a cross-case analysis of decision-making in project management [26]. The authors identified antecedents to multiple cognitive biases, e.g., the absence of records of the learned lessons from previous projects, can lead to availability bias³ during decision making by a project manager [26]. These studies support the use of interview data collection method and grounded theory as an appropriate approach for exploring and understanding the phenomenon of confirmation bias.

This section shows that the existing literature on confirmation bias is limited to controlled experiments only. These studies tested hypotheses about isolated factors as potential antecedents to confirmation bias. A qualitative study is, therefore, required to explore what other antecedents lead to confirmation bias and how? Our study, by applying grounded theory, explores other antecedents to confirmation bias in software testing. We also aim to understand how these antecedents lead to confirmation bias. The postulates generated by our theory can be verified by further empirical studies.

3 RESEARCH METHOD

We apply grounded theory (GT) as our research method. The objective of GT is to generate a theory that is grounded in data [31]. According to Urquhart, “*Theory asserts a plausible relationship between concepts and sets of concepts, and the*

resulting theory can be reported in a narrative framework or a set of propositions” [31, p. 5]. GT is suitable to address our study’s objectives because of a lack of empirical evidence on the antecedents to confirmation bias, and why testers manifest confirmation bias is yet unknown; to the best of our knowledge [10]. Therefore, we try to understand, “What’s going on here?” [32, p.120]. “Here”, refers to our context of understanding, why and how confirmation bias occurs. We apply the Glaserian version of GT because we wanted the specific research questions to emerge during the data analysis [32]. Our ontological position is positivism. By using the GT’s inductive theory-building process, we first present the theory of the phenomenon under study in narrative form in Section 4. An integrative diagram of the theory is then presented in Section 5 that explains the inter-relationship of the derived concepts and categories. We follow the guidelines by Stol *et al.* for reporting this study [32].

3.1 Goal

In the context of this study, confirmatory behaviour occurs when a tester designs or executes consistent test case(s), and a disconfirmatory behaviour otherwise. In order to have complete coverage for the SUT, a tester should manifest both confirmatory and disconfirmatory behaviours. In other words, a test suite should be complete in terms of consistent and inconsistent test cases. We refer to it as the completeness of a test suite in this study.

Certain antecedents may lead to a compromise of one behaviour over the other, e.g., disconfirmatory over confirmatory. Thus, possibly not only promoting confirmation bias but also limiting the completeness of a test suite. The objective of this study is to explore the antecedents to the confirmatory and disconfirmatory behaviour of software testers while performing testing. It is worth to find out antecedents also for disconfirmatory behaviour because the absence of them may imply the promotion of confirmatory behaviour, which may lead to confirmation bias by software testers. We, therefore, answer the following research questions:

RQ1: What are the antecedents to confirmatory and disconfirmatory behaviour among software testers?

RQ2: Why or how do the antecedents influence the behaviour of software testers as confirmatory or disconfirmatory?

The objective of the study initially helped define RQ1. RQ2 emerged during the data analysis, which also, in turn, refined RQ1. The application of the Glaserian coding techniques enabled us to break down the research question into specific research questions [31].

3.2 Context, Participants, and Data Collection

We used interviews as a primary data collection method for our study. Interviews are a qualitative source of data that perfectly aligns with the inductive process of GT [31].

We interviewed twelve professionals working in a world leading company in Information and Communication Technology domain. To maintain anonymity, we refer to this company as Company-ICT in our study. The Company-ICT is offering services in networks, digitisation of solutions, managing IT services and providing solutions in IoT areas. The Company-ICT develops internal software projects using Agile software development. The projects have dedicated

3. “Availability bias refers to a tendency of being influenced by the information that is easy to recall and by the information that is recent or widely publicised” [10, p.21].

TABLE 1
Participants (*Exp. in Testing* Does Not Account the Duration of Testing as a Software Developer)

P#	Job Title	Exp. in Company-ICT	Exp. in Testing	Tester Type	Interview Length
P1	Software Engineer	2 yr 11 mos	6 mos	Manual	71 min
P2	Test Engineer	6 yr	7 yr	Manual	79 min
P3	Senior IT Test Engineer	5 yr	9 yr 6 mos	Manual, Automation	57 min
P4	Senior Software Test Engineer	2 yr	2 yr	Manual	80 min
P5	Test Engineer	5 mos	9 yr	Manual	58 min
P6	Solution Architect	5 yr 6 mos	12 yr	Automation	76 min
P7	Experienced Integration Engineer	4 yr	4 yr	Manual	81 min
P8	Integration Engineer	2 yr	2 yr	Automation	58 min
P9	Configuration Manager & Test Engineer	4 yr	6 yr	Manual	68 min
P10	IT System Expert (Software Test Expert)	3 yr	6 yr	Manual	51 min
P11	Test Team Manager	5 yr 6 mos	5 yr 6 mos	Manual	53 min
P12	Software Test Engineer	1 yr	4 yr 6 mos	Manual	39 min

testing teams who perform higher levels of testing, e.g., integration testing, while developers are responsible for performing unit testing. In case of a small project team, one person may perform multiple roles, despite their job title, e.g., software architect also performs testing when required. The company also has dedicated test automation teams that are not part of any particular project; they automate manual test suites of the projects. We chose this company for two main reasons: 1) it acts as a vendor to conduct system tests on behalf of its business contractors, thus, 2) it has been collaborating with academia, international partners in EU and nationally funded projects for improving its testing process, test effectiveness and measurement.

The interviewed professionals participated in this study voluntarily. We specified the recruitment of test engineers or engineers with testing experience to our contact person at the company because we wanted a sample well aligned with the goal of our study. The champion approached the pool of more than fifty software testing engineers through their respective managers. Twelve engineers positively responded to the call of the champion for participation. The sample consisted of 10 test engineers. The additional two were: a solution architect and a software engineer. The solution architect was also partly performing activities as a test engineer, and the software engineer was involved in both development and testing (as a test engineer). The participants belong to different projects or domains at the company's two sites. We refer to all these participants as testers from now onward in this study. Based on the characteristics of our participants and set-up of the company, testing performed as a developer is not accounted for in this study's scope. We focus on the higher levels of testing performed by testers. It is important to note that our study does not aim to achieve statistical generalisation with this sample because, in qualitative research, researchers generalise to theory instead of a population [31], [33]. We are exploring the phenomenon in the defined context rather than achieving representativeness [33]. However, the issue of achieving generalisability with a positivist GT approach is discussed later in Section 5.5.

The format of the interviews was semi-structured. Before conducting the actual interviews, we piloted the script with a software engineer from a different company. The objective of the pilot interview was to improve the wording of the questions and timing of the session. The interviews were

conducted in October 2017 via Skype through video-calls, and voice-calls when the video was not viable. On average, it took 65 min per person to interview. We collected approximately 13 hr of audio (with informed consent) and 127 pages of verbatim transcribed data. One of the authors went through the transcriptions and audio files again to tally the content and to ensure that technical terms were correctly transcribed. The interview script is available as an online appendix⁴.

The characterisation of participants is given in Table 1. The participants have at least two years of working experience at the Company-ICT, except for two of them. Only one participant has only 6 months of testing experience otherwise the average testing experience is approx. 6 years. Two of the testers are automation test engineers, one is performing testing both manually and in an automated way, the rest are all manual testers.

3.3 Data Analysis

The data analysis procedures in GT are systematic [31]. The applied coding techniques are open coding, selective coding and theoretical coding. The application of the constant comparison method (CCM) to the coding techniques made coding an iterative process [31], [32], [34]. We followed the guidelines by Urquhart and Boeije for applying the mentioned techniques [31], [34]. An example of deriving two of the antecedents (past experience, project experience) belonging to a single category, *experience*, through the applied coding techniques is illustrated in Fig. 1. The sample raw data from three interviews, P2, P3 and P10, is shown separately in the figure. We first applied open coding to the individual interviews and then filtered it to the relevant concepts, i.e., antecedents, per the objective of our study. The open coding is shown as **bold** texts in the excerpts. After the application of CCM within the interviews and among the interviews, and the application of selective coding, concepts emerged. The emerged concepts were then grouped under a category, which is a higher level of abstraction. In the illustration - Fig. 1, the category is *experience*, which is one of the identified antecedents to disconfirmatory behaviour. We also applied memoing and memo sorting along the process of selective coding and CCM. It enabled us to classify the antecedents as (dis)confirmatory and to capture the relationships between

4. <http://doi.org/10.5281/zenodo.3376920>

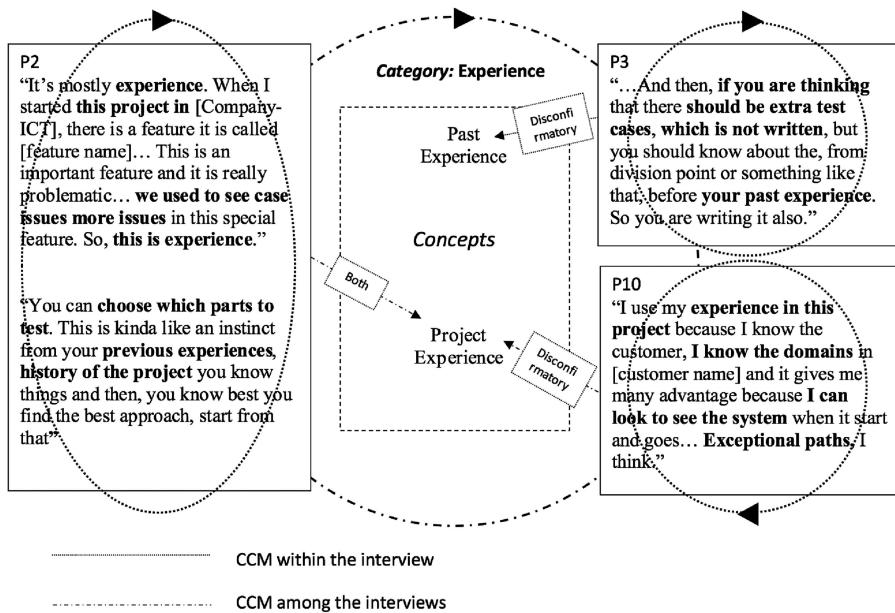


Fig. 1. GT coding mechanisms.

TABLE 2
Data Extraction

Concept	Definition
Antecedent	Of the testing process (e.g., reviews) OR from the environment (project, organisational) OR personal attributes (e.g., experience) that leads to the manifestation of a (dis)confirmatory behaviour.
Antecedent Classification	
Confirmatory	The evidence indicates the promotion or manifestation of a confirmatory behaviour.
Disconfirmatory	The evidence indicates the promotion or manifestation of a disconfirmatory behaviour.
Both	The evidence indicates the manifestation of confirmatory and disconfirmatory behaviours without stating the relative preference of one behaviour over the other.
Unknown	The effect on behaviour is evident but leading to either a confirmatory or disconfirmatory behaviour is not explicitly mentioned.

the emerging concepts. This led us to the integrative diagram as a result of theoretical coding, which is the third stage of coding in the Glaserian GT [31], [32]. We used NVivo⁵ data analysis tool for coding.

We implement coding validity steps because of our positivist ontological position, as recommended by Urquhart [31]. In our context, *intercoder reliability* refers to, two or more coders identify the same code (antecedent) and use same classification (e.g., confirmatory, disconfirmatory) for the code, when coding independently [35]. *Intercoder agreement* assurance requires that the coders discuss and reconcile their coding discrepancies [35]. We performed multiple steps to ensure intercoder reliability and agreement for the identification of antecedents to confirmatory or disconfirmatory behaviour.

One of the authors (the interviewer) initially formed a list of the terms that interviewees used to indicate their confirmatory or disconfirmatory behaviour during testing. Afterwards, we developed a coding protocol that comprised coding guidelines and the previously formed list of terms. One of the authors is experienced in applying grounded

theory coding techniques, two of the authors contributed with knowledge on software testing and cognitive biases. One of the authors brought in expertise in software testing. Therefore, we also developed a unanimous understanding of confirmation bias and its manifestation in the studied context. The four authors then performed a pilot coding of a randomly chosen interview (P2) from the set of twelve interviews. The objective was to identify antecedents and classify them as confirmatory or disconfirmatory, and validate the coding process. The objective was also to ensure that codes produced by any single knowledgeable coder would be reproducible by other equally knowledgeable coders, as all the authors may not be available to code the data [35].

The joint discussion session, after the pilot coding, revealed a need for a refined understanding of an antecedent and introduction of more categories for classifying antecedents. As, we noticed that some of the antecedents could neither be classified as confirmatory nor disconfirmatory, rather both or unknown - see Table 2. Additionally, we decided to disregard the data where the interviewee's understanding of the terms differed from the theoretical definitions, e.g., referring functional testing to as non-functional testing. This decision was taken to prevent personal interpretations on the coder's behalf because further

5. <https://www.qsrinternational.com/nvivo>

communication with the participant also couldn't clarify those misunderstandings. The percentage of such data accumulated to 0.92% by the end of the data analysis of all interviews.

Due to the recognition of the issue with antecedents' classification, we assessed intercoder reliability⁶ of our pilot coding only for the identified antecedents. The agreement between (coder1, coder2) was 54%, (coder1, coder3) was 39% and (coder1, coder4) was 46%. We assessed agreements in pairs with coder1 because it was decided that coder1 would code the rest of the interviews. In order to improve the intercoder reliability measure, we performed the following steps.

The other three coders then revised their identified antecedents and reclassified them according to the classification presented in Table 2. It was followed by one-on-one discussions with coder1 on the identified antecedents and their classification. For example, between coder1 and coder2; a code was defined for higher authority involvement in deciding the density of issues to deliver a patch with. After analysing the excerpts and context, it was decided that this does not qualify as an antecedent because it is not influencing a tester's behaviour during testing activities. Later, coder3 and coder4 also agreed to that decision. These discussions provided input to coder1 to revise the coding of the P2 interview. Coder1 then coded the rest of the interviews in three iterations. After each iteration, we held a joint discussion session in which confusions regarding the coding were resolved. For example, a confusing excerpt that was coded for the antecedent *change request* was resolved to illegible evidence towards any of the classification: "Actually, we make a task separation between us [with his colleague] before making test cases, and we start to create test cases of our products. And, then, we come together and we try to question our test cases, we try to reveal the solution documents together and we try to understand, actually, we have a short [technique name]. We are trying to compare our test cases with these objectives, smart. And, we are trying to meet the requirements which we gathered. There is no specific thing like we do. We're just discussing." - P7. This was an answer to a question about the participant's discussion with their colleague, which did not indicate any effect (per Table 2) on P7's behaviour. In each iteration, coder1 also shared the classification of the excerpts of the emerging concepts (antecedents) to be validated by others. These steps ensured that coder1's bias was minimised and coding is reproducible.

4 RESULTS

We have identified 20 antecedents, which are classified into nine categories. First, we define the category, then we introduce the respective antecedents with the description and evidence of *why* and *how* they influence the dis(confirmatory) behaviours. The definitions of the formed categories are based on the collected data. Table 3 summarises the answers to RQ1 and RQ2. The columns C (confirmatory), D (disconfirmatory), B (both) and U (unknown) present for how many participants the respective antecedent influenced as C/D/B/U for their behaviour. The Total column presents

the total number of participants and the total number of excerpts that provided the evidence for C/D/B/U for the respective antecedent as $x(y)$. It is important to note that x is not a sum of the counts reported in the previous columns because in a few cases the same interview provided multiple evidence. *Why & How* (RQ2) summarises how the antecedent influences the behaviour. The example evidence excerpts presented in this section are revised from grammatical and comprehension perspective because the interviewees were not native English speakers.

4.1 Experience

Experience refers to the knowledge of an individual tester that they acquired by working in either a different, similar or the same project, roles and company, and the application of this knowledge for software testing. The data analysis showed that *experience* mainly leads to disconfirmatory behaviour. However, it also promotes a confirmatory behaviour, which results in improved completeness of a test suite.

4.1.1 Past Experience

Past experience refers to the experience of the participant in general or they did not associate it with any particular project domain. It promotes disconfirmatory behaviour by designing inconsistent test cases. For example, "You are making the happy path [consistent scenarios] and then you are making some negative scenarios, that are functional or communicated by the customer. And then, you think of the need for extra test cases, which are not written [in specifications], but you should figure them out based on the decision points [of the functionality] or other similar conditions that you have learned from your past experience... Most of such test cases are failure cases [inconsistent] because the happy cases should be [written] in the documents." - P3. In this evidence *extra test cases* contextually refers to inconsistent test cases. Past experience also results in the occurrence of both behaviours because an experienced tester knows how to approach an SUT, as explained by P7: "... the more you do testing, the more you get experienced, and the more you know how to approach a product or a system... I can say both [consistent and inconsistent]. I cannot comment about the preference of one over the other".

4.1.2 Project Experience

This experience is acquired either by working in the same project over the years or in the same domain, which may not be limited to the Company-ICT. Project experience renders an enhanced perspective on the project due to which disconfirmatory behaviour manifests. This is explained by P10: "I use my experience in this project because I know the customer, I know the domains in [customer's company name], and it gives me many advantages because I can view the system, when it starts and runs..."; it helps in designing more "Exceptional paths [inconsistent tests]" - P10. This antecedent also increases the completeness of the suite by prompting both behaviours, e.g., P9 stated: "I joined this project when it started four years ago. As the project scaled up over the years from a little code base, little tests; I learnt the project well all along. Due to that knowledge, I can see [visualise] the end-to-end part of the [domain name]... 90 percent of the times, I can know, yes this is

6. Percentage of the total number of common antecedents divided by the total number of identified antecedents.

TABLE 3
RQ1: Antecedents and Evidence, RQ2: Why and How

Antecedent (RQ1)	Why & How (RQ2)	C	D	B	U	Total
<i>Experience</i>						
Past Experience	Experience in testing enables how to approach the system, and which particular functionality checks to test for.	0	3	1	0	4 (6)
Project Experience	It enables testers to visualise the end-to-end functional flow of a system, and a good learning of the customer domain.	0	5	4	1	10 (28)
<i>Priority (of a functionality, user stories or platforms)</i>						
High Priority	Consistent test cases have higher priority, but inconsistent test cases also acquire an equal priority in case of high priority scenarios, of a certain functionality.	2	0	2	0	4 (11)
Medium or Low Priority	Testers give either less or no consideration to the designing or execution of inconsistent test cases.	2	0	0	0	2 (3)
<i>Requirements</i>						
Ambiguous Requirements	Impedes correct and complete test case designing, which results into the design/execution of mostly consistent test cases.	3	1	0	2	6 (14)
Clarifying Requirements	Clarifying ambiguous requirements leads to the designing of both consistent and inconsistent test cases.	0	1	3	2	6 (17)
Incomplete Requirements	Confirmatory because it limits the testers to test only what is minimally specified.	1	0	0	1	2 (2)
<i>Functionality Retesting</i>						
Production Bug Fix	Testing the fix first is confirmatory that is followed by the testing of relevant inconsistent cases and other consistent test cases.	0	1	3	2	6 (10)
Change Request	Both behaviours occur while designing/executing cases for a change request.	1	0	5	1	7 (8)
Change/Fix Size	Minor change is confirmatory due to its minor impact on the system. Both behaviours occur in case of a major functional change.	3	0	2	0	3 (9)
<i>Test Suite Reviews</i>						
Internal Party Review	When performed by members of the same project, it is disconfirmatory, and also enhance the completeness of a suite.	1	7	3	2	6 (26)
External Party Review	By customers who define which devices to test, and set priorities. Therefore, only adhering to those priorities is confirmatory.	1	0	0	1	1 (9)
Automated Test Suite Review	Manual testers review to validate conformity with the manual suite. They are not expert in automation to assist with the handling and coverage of inconsistent test cases.	1	0	0	2	3 (9)
<i>Testing Mode (manual or automated)</i>						
Automated Testing	Confirmatory because it is difficult to automate every inconsistent test case and to handle unexpected results.	2	1	1	0	4 (5)
<i>Test Execution Feedback</i>						
Detection of Errors	It leads to further testing to find more errors, and sometimes the addition of more inconsistent test cases.	0	5	0	0	5 (7)
Absence of Errors	Disconfirmatory because it leads to rethinking of the test approach, and assessing a test suite from a different perspective.	0	4	0	0	4 (7)
<i>Time</i>						
Time Pressure	Consistent test cases are prioritised because they ensure the behaviour of the SUT per the documented specifications.	4	1	3	1	9 (27)
No Time Pressure	Leads to more testing e.g., through exploratory testing and the execution of more inconsistent test cases.	0	1	0	2	3 (8)
<i>Perspective Change (testing from a changed perspective)</i>						
Developer & Tester	Disconfirmatory because testing as a tester, compared to development, changes and broadens perspective for the SUT.	0	1	0	0	1(7)
Complement Testing	A tester executing test cases that were previously executed or designed by another tester promotes disconfirmatory behaviour.	0	2	0	1	3(5)

an exceptional [inconsistent], and yes this is a happy path [consistent], and it is important to test".

4.2 Priority

It relates to the priority of a functionality and user stories or OS platform (e.g., Android). Our data informs that priorities are defined by customers, roles higher in a hierarchy to a software engineer (e.g., product owner, project manager, test manager) and it is based on the functionality (e.g., finance). If the higher management or customers are not setting the

priority, then testers themselves define them based on their experience, i.e., *past experience* or *project experience*.

4.2.1 High Priority

In the context of automating manual test suites by an automation engineer, higher priority scenarios take precedence. According to the data, testing of consistent scenarios usually have a higher priority. However, if a functionality or particular scenario is a high priority, then it also leads to both behaviours. P6 explains this in the context of automating a manual

suite that they, then, automate consistent and inconsistent test cases with equal priority: "...It should be definitely both [consistent, inconsistent]. When we are talking about the top priority test scenarios or the negative scenarios [inconsistent], it's almost equally important like a happy path [consistent] test scenario." Our analysis further suggests that high priority testing may not lead to an enhanced coverage; we discuss this in Section 5.

4.2.2 Medium or Low Priority

If functionality is not high in priority, then it could be a medium or low priority. In this case, inconsistent test cases receive either less or no consideration in designing/execution, which results in a confirmatory behaviour manifestation by a tester. P9 explained it as: "If a function is important, all the happy path [consistent] and the exceptionals [inconsistent] are also important. But, some functions may not be very important, and we can skip the exceptional scenario for not important functions" - P9.

4.3 Requirements

This category refers to the documents that serve as requirements specifications for testers to prepare test cases. It includes technical documents, business rules, functional documents, high-level design documents and low-level design documents.

4.3.1 Ambiguous Requirements

Ambiguous requirements are such requirements that are either not well defined or are difficult to understand by the testers. According to P2, such requirements affect the activity of preparing test cases because testers have to ask for clarifications; "...these requirements might be not very clear, sometimes you might need to ask more questions about the documents, to be able to make all your test cases clear and comprehensive enough, to be able to test the system" - P2. Other participants' data indicates that ambiguous requirements promote confirmatory behaviour. Testers design more consistent test cases because it helps them in understanding the requirements well to design inconsistent test cases, afterwards. Otherwise, they only design and execute consistent test cases based on their own understanding. For example, "If I do not understand what's going on [in requirements], then I'm not able to write test cases... I code [design] happy [consistent test cases] just because I'm not clear with what they expect me to do [test]. And, I'm not sure what system does, so I go with happy path and if products do not crash, then I say it's okay" - P12.

4.3.2 Clarifying Requirements

Clarifying requirements is an activity that is performed by testers to clarify ambiguous requirements to improve the testing of a functionality. Testers clarify the requirements with customers, product owners, developers or project managers. It usually leads to the completeness of a test suite when testers manifest both behaviours. P1 states this as: "I don't know [understand] all of them [the requirements]. I will exchange my comments with customers, whether I am understanding them right, or maybe it's not required [a particular functionality], it's not end-

user's behaviour. I will give comments about all of them... Yes, both [consistent and inconsistent], I will check all of them".

4.3.3 Incomplete Requirements

This antecedent is different from the above antecedents because it refers to minimal requirements. In other words, requirements may be ambiguous but may not lack details on the required functionality to be tested. For example: "If no information [is available] about the task. For example, they [authority figure for preparing the documents] wrote only a single sentence about a problem's fix on the production, and developers fix it. It's sometimes difficult for the tester to understand [the functional fix], what did he [the developer] do, and what was the real problem" - P10. The incomplete specifications, in case of a production fix, make it difficult for a tester to perform proper testing because they lack details on the functionality. It promotes confirmatory behaviour because testers, test per the minimal information that limits testing inconsistent scenarios. As further explained by P10: "it will affect, what I don't know [the requirements], so it affects my test cases... Exceptional or failure ones [inconsistent]. Because I don't know the details. Only focus on the happy paths [consistent], maybe I miss [testing] something".

4.4 Functionality Retesting

Retesting refers to retesting a module or functionality after its re-implementation, in case of a reported production bug or a functional change request. In addition to retesting of a particular fix or change, it is also done for the relevant impacted functionalities of the SUT.

4.4.1 Production Bug Fix

The data analysis showed that both behaviours occur due to retesting a fix of the production bug. After validating the fixed scenario, the tester begins to test the inconsistent scenarios of the module, which is followed by the testing of consistent scenarios. For example, P10 stated: "First failures [that failed], and then check the happy paths... I also ask the developer, "which code did you change and which cases does it affect?". First we talk, then I check the failure one, and [then] check the success path". Per this evidence, the tester first validates particularly a fixed scenario. It is a confirmatory behaviour because they are confirming the communicated (serving as a requirement) functional flow. Then, a disconfirmatory behaviour when they validate the other relevant inconsistent scenarios, which is followed by the testing of other confirmatory test cases.

4.4.2 Change Request

Retesting a module, in case of a change request mostly leads to both behaviours. For example in the context of automation testing, P8 stated: "We should delete some methods, and we should have some other control points, and add some other modules or functions to the automation framework... Both. Happy [consistent] paths and exceptional [inconsistent] and failure scenarios". On further enquiry, the participant explained that first they prefer to test consistent scenarios.

4.4.3 Change/Fix Size

The size of the implemented change or a bug-fix also influences the behaviour of testers. In case of a minor change,

testing is confirmatory and limited to a particular functionality. For a major change, the manifestation of both behaviours was reported. A type of a major and minor change is elaborated by P5 as: "for example some text box or button is not in the right place on graphic user interface. Either it is there or is not visible. So, I just test this because it's a makeup thing, just an interface issue. It's not a major big problem. But, if I cannot make any stock or product transfers, i.e., main function is not working at all, of course, that means that all product transfer function will be tested from the top to down". However, time availability also plays a role in retesting: "if you don't have much time; e.g., if you have a small change, it's not affecting all the release, all the software outcome, if it doesn't affect every part of your platform, you can just run a quick happy path [consistent] test cases" - P2.

4.5 Test Suite Reviews

It is the review of test suites that are designed by testers, prior to suite executions, to ensure the completeness of the suite with respect to the SUT. The antecedents of this category indicate two types of reviewers, which promote different kinds of behaviours among testers based on their review-feedback.

4.5.1 Review By Internal Party

These reviews are conducted by the roles who are employees of the Company-ICT. They may be part of the same project or team, i.e., project manager, solution architect, product owner, team lead, development lead, test expert or fellow testers, or testers from other projects. The reviews from members of the same project/team mostly promote a disconfirmatory behaviour by recommending to accommodate more inconsistent test cases. P11, who also reviews others' test suites, stated: "Generally they forget exceptionals [inconsistent] scenarios because they argue that it works. But, I check and [fore] see other different bugs. And generally I suggest, "You can write some exceptional scenarios; [e.g.] sometimes bad things [situations], sometimes field checking; it's important" - P11. According to P3, reviews enhance the completeness of the suite: "When I am adding, most probably, you are not adding the happy [consistent] cases. When you are sending it for review, a very small part, maybe five per cent that they are arguing or asking for an extra [test cases]". On enquiring the type of 'extra' cases, P3 replied: "It's changeable because they are giving review, which you forgot about [test cases]... Both [consistent and inconsistent]" - P3. The data analysis also shows that reviews performed by testers from other projects are confirmatory because they are not knowledgeable about the functionality. It limits their perspective that could promote disconfirmatory behaviour.

4.5.2 Review By External Party

External reviews are performed by customers. The purpose is to get their feedback, if the suite meets their expectations, to continue with the test execution. Per P2: "We have shared this with customer, if these test cases, test suites meet their expectations to be able to test the system... We test on mobile platforms, e.g., iPhone, iPad, Android tab, Android phone. So customer can say, it is enough for us to execute the tests only one Android device, and only one iOS device. This is enough. So, "Continue your tests on the set top box, which is more important for us." They can say this. So, we have to consider this". The review

from customer influenced the coverage, which in this case is limiting testing to certain devices. Additionally, the feedback also defined priorities for testing. It is confirmatory because the tester is confining the testing only to the customer's feedback, per the available evidence.

4.5.3 Automated Test Suite Review

Reviews of automated test suites are internal reviews. However, it is different from internal and external reviews because those are performed only for manual test suites. Contrary to the range of roles involved for manual suites, automated test suites are reviewed only by manual testers. The major reason for this is, automated suites development is based on manual suites. P6 stated this as: "For the main sources are manual test scenarios. We are expected to automate the manual test scenarios as it is, the same steps, the same verification points, the same databases... we have only the manual tests and they just want us to simulate it". The manual testers usually assess and compare the functional flow of the automated tests with manual tests. Therefore, the quality (the level of completeness) of manual suites gets transferred to the automated suites, as P3 stated: "you are simulating the manual testing, so you should take a proof review on the manual testing; It's OK or not". P6 explained: "the common observations they [manual testers] are giving are observation on the happy path [consistent] test scenarios. But, the exceptional [inconsistent], there are some experienced test engineers, giving some feedback about the negative [inconsistent] scenarios, but this is less, maybe one in a ten" - P6. This evidence cannot be considered as *Both* because the frequency of feedback that can lead to the addition of inconsistent test cases is considerably low.

4.6 Testing Mode

This category refers to the mode of testing, i.e., manual testing or automated testing.

Automated Testing

Automated testing is the testing performed in an automated way using tools, e.g., Visual Studio, Selenium. Test automation engineers develop automation scripts that run the tests in an automated way. In comparison to manual testing, automated testing leads to confirmatory behaviour. It is because of the difficulty to code inconsistent test cases and automation tool's limitations in this regard. For example, P3: "Automation is more [about] happy cases, you can say that. Of course, it can handle negative [inconsistent] test cases, but with manual testing, negative cases or unexpected times or results can be handled better. With automation test cases it's more difficult to handle unexpected results". Also, P8 complemented to this; "... from manual testing opinion you should test the whole thing. Because you are a user on the computer, you are using that computer with your hands, with your mouse. You can do anything in that time. But in automation testing, you should write a code. This is not the natural way. This is about the priority of testing, I think. The automation testing mainly focuses the happy paths". According to P2, a tester can benefit from the confirmatory nature of automated testing due to its fast execution time. "If you automate 200 cases... you can run them e.g., in one hour or two hours. Rather than spending a day or two man-days. So, this shows you a general outcome, result of this. And you can

review it. You can say, "Okay let's now concentrate on these failure [inconsistent] cases because we haven't automated [testing of] these cases and these features. Let's focus on them." This gives you a good time to focus on other features, areas and failure cases" - P2.

4.7 Test Execution Feedback

This category refers to the effect of the results of a test suite execution especially when a module is tested the first time. The data analysis has revealed that detection of errors and also an absence of errors lead to a disconfirmatory behaviour by testers.

4.7.1 Detection of Errors

This antecedent refers to the situation when a test case failed due to the presence of error. It is disconfirmatory because it leads testers to find more errors in the SUT by performing further detailed investigations. In case of P9 it leads to exploratory testing, i.e., performing more manual testing for finding errors, hence, a disconfirmatory behaviour: "When I see a bug, I first open a trouble report... And after that I think, [if] there is a bug, maybe other scenarios are also troubled. And also, I do free tests [exploratory testing] at that part, at that time maybe... if there is one bug there must, there can be other bugs. And, I investigate that part, and sometimes I get [detect] other bugs, sometimes not" - P9. It also leads to the addition of more test cases, as P5 stated: "if I find a new important bug, I go deeper, and I also won't let the test cases [go] unseen. I still run the cases, and if I find some exceptional cases that I couldn't consider before, I know they're exceptionals, I go deeper, too... Maybe I didn't consider [it] before [test cases], and it's also not written in the requirements, So I write it down". This antecedent's influence on the behaviour is observed only for manual testers, not for automation test engineers - discussed in Section 5.

4.7.2 Absence of Errors

This is a situation when all test cases of a suite pass, i.e., no error is detected by the suite for the SUT. This promotes a disconfirmatory behaviour among testers because it makes them curious over the situation and to rethink of their test approach. Per P10: "I always think, I'm doing something wrong. "How [could] they develop with no bug?". P2 explains this situation as: "If you can't find some issues with your test set, there might be issues in your test set approach. So you should be able to consider error cases [inconsistent], failure cases [inconsistent], what's going on. Go over your documents, test sets, and then detail some of them, change your mindset, how you created them". Hence, this situation also prompts testers to force the system from a different perspective to reveal its errors. In case of automation testing, testers execute a few test cases manually to reassure a 100 percent pass result. "Green is kind of a very relieving colour, and when you see green all over, you feel very happy. Of course, we are investigating, we are just executing manually a couple of test scenarios. Let's see [if] it really passed all the test scenarios" - P6. It is a disconfirmatory behaviour because it led the tester to investigative more rather than being contented by the test results of automated suite.

4.8 Time

This refers to the available time for two main testing activities: test suite designing and test suite execution.

4.8.1 Time Pressure

It is an insufficient time availability from testers' perspective for performing testing in the situations when they do not arrange overtime. The data analysis shows, in this situation, most testers manifest confirmatory behaviour because they prioritise to validate that SUT accomplishes the specified functionality. Inconsistent test cases fall second because they validate implicit functionality, i.e., not explicitly mentioned in the specifications. P10 stated: "I want to test more but I have a limited time. I only check the happy path [consistent] or, one or two exceptional [inconsistent] test cases. But I think, I should test more and [also] check the other exceptional ones. But I don't have time, and should start [testing] the other project. So, it limits my execution, I think. Exceptional test case execution". Those who were observed to manifest both behaviours, for most of them a confirmatory behaviour occurred prior to a disconfirmatory behaviour. For example, "If I have really a short time, really short time, of course, first I need to see the system is working correctly, the happy path [consistent]. Whether the happy path passes right or wrong. I mean, then exceptional cases [inconsistent] of course. But I have to tell you that happy cases don't take that long time, just pass away" - P5. Testers who first manifest a disconfirmatory behaviour, they compromise on the testing of consistent scenarios. "So for urgency [time shortage], I first start with exceptional [inconsistent] scenarios. And for urgency sometimes, you make exploratory tests, based on our experience of the product... If it's enough urgent, you sometimes, trust the development team that they should have developed these according to requirement" - P7.

4.8.2 No Time Pressure

No time pressure refers to two situations: 1) when testers are finished before the deadline, or 2) testers have enough time to perform testing, i.e., without doing overtime. According to P2: "You should make enough time to run even different tests. Sometimes, we have free time and we don't base it on any test set [designed tests]. We just start testing a mixture of functional and non-functional tests". In this case, the tester is performing exploratory testing. The influence on the behaviour is not known because it is not clear, whether they execute consistent test cases or inconsistent ones. However, it definitely leads to the execution of more test cases. However, P9's behaviour is disconfirmatory in this regards: "... if I have enough time, I also execute free tests... not related with any [designed] test cases. Maybe there are [exist] test cases, but I do not know... I also do [test] the exceptional [inconsistent] cases. For example, in this example [case], maybe more of them will change the status [of the feature] four or five times, but when I test, I change it [the case]". P9 also referred to the execution of exploratory testing that consists of inconsistent tests. The test case, in this evidence, is validating the status' feature for the situation for which a test case may not already exist.

4.9 Perspective Change

The change in perspective occurs either due to a change of role or testing the functionality that was previously tested by another tester. The antecedents of this category promote disconfirmatory behaviour because of the changed perspective.

4.9.1 Developer and Tester

A software engineer working in two roles, as a developer and tester in the same project, also influence their behaviour. P1 explained this as: "when I am a developer, I just focus on happy [consistent] paths, maybe one risky [inconsistent] case but mainly happy path to check it out if it is okay. But when I am a functional tester, I will see every risky point. If I am an integration tester, I will force every possible error from the integration part because it's the most risky thing in our environment. So it changes my approach". After a confirmatory behaviour in testing as a developer, testing as tester changes and broadens their perspective of the SUT, which leads to a disconfirmatory behaviour. However, P1 related the reason for disconfirmatory behaviour in testing to their *past experience*.

4.9.2 Complement Testing

This antecedent refers to two situations: 1) a tester executing test cases, designed by another tester, and 2) a tester, testing the functionality that was tested by another tester in the previous test cycle of the same release. P12 explained the first situation as: "We do not actually check the entire cases because our test lead separates the things that we do. So we are seven people and two or three of them write the test cases and the other three or four, run the cases. And if we find something that was not included in those cases, we add it". On enquiring further, they mentioned that the missing cases are usually inconsistent test cases. Hence, the execution of the suite that was designed by another tester prompted a disconfirmatory behaviour because it enabled a different perspective to test the same functionality. This antecedent also leverages improved defect detection, e.g., P4 stated: "I run the [functionality-1 name], [functionality-2 name], [functionality-3 name], for example. Other LSV [system testing] cycles, my other friends run [functionality-2 name], [functionality-1 name] and [functionality-3 name]. If I miss something, miss a failure, miss defects, maybe she finds it. Therefore, we have little defects".

5 DISCUSSION

We first discuss the classification of the identified antecedents and present the integrative diagram, followed by their comparison with the antecedents from the existing literature. The section also presents the implications for research and practice. Finally, the threats to validity are discussed.

5.1 Classification of Antecedents

Based on the results, we can classify the antecedents from three aspects: confirmatory, disconfirmatory, and *both*, that represents the test suite completeness perspective. The categories: *test execution feedback* and *perspective change*, and the antecedents: *no time pressure* and *past experience* are disconfirmatory. The antecedents: *medium or low priority*, *incomplete requirements*, *external party review*, *automated test suite review* and *automated testing* are confirmatory antecedents. *Ambiguous requirements* can also be classified as a confirmatory antecedent. The rest of the antecedents, in addition to providing the evidence for confirmatory or disconfirmatory behaviour, also provide evidence for the manifestation of both behaviours. It is important to note that *both* may not suggest a complete test suite, albeit an improved suite. For example,

the antecedents: *change request* and *clarifying requirements* mainly lead to improved completeness of a test suite. *Production bug fix* leads to an increased execution of the test suite because it is in the context of retesting, as indicated by the antecedent's category. For the two antecedents, *high priority* and *time pressure*, *both* does not suggest completeness of the test suite in terms of design, and also a complete execution of a test suite. It is detailed later in this section. The antecedents: *project experience* and *internal party review*, in addition to the promotion of disconfirmatory behaviour, also improve the completeness of the suite. *Change/fix size* is a special case because, for major change/fix, it is both the behaviours, otherwise it is confirmatory.

Despite the proposed classification of the identified antecedents, our data suggest that exclusive classification of these may not be practical. If an antecedent leads to confirmatory behaviour for some testers, it may also lead to disconfirmatory behaviour for other testers, which could be pertained to certain factors, e.g., personality elements. This issue and other particular aspects related to the identified antecedents are detailed further.

General and Specific Behaviours: The data informs that the general behaviour of testers is to first manifest a confirmatory behaviour, i.e., designing of consistent test cases and then a disconfirmatory behaviour, which is designing of inconsistent test cases. It happens because they identify inconsistent test cases based on consistent test cases. The automation engineers reported the same behaviour sequence when they automate the manual suites, though they only simulate the manual flow. In addition to designing the test suites, test execution also follows the same course, i.e., first confirmatory and then disconfirmatory.

A few participants manifest the opposite sequence of behaviours - specific behaviour. They manifest disconfirmatory attitude prior to confirmatory attitude, which is pertained either to their *experience*, particular nature or assumptions (developers must have rightly implemented the consistent scenarios). For example, in Table 3, the evidence of specific behaviour can be seen for *ambiguous requirements* and *time pressure*, though they are confirmatory antecedents. P1 associated this with their experience. Also, this pertains to tester's nature; in the context of ambiguous requirements, they stated: "It makes me check risky scenarios... Because I think like this, if it is difficult it might be more risky... so I need to see them first" - P1.

If a test suite designing or execution is complete in terms of consistent and inconsistent test cases, then manifesting one behaviour before the other is not an apprehension. However, if completion of one type of test cases is compromised (due to the antecedents), then the behaviours may lead to adverse effects on software quality, as already explained in Section 2.

Functionality Retesting: For testing change requests or production bug fix, the same sequence of general and specific behaviours occurs. *Project experience* also influences the preferred behaviour manifestation by the participants, i.e., confirmatory or disconfirmatory.

High Priority Testing: The testing with inconsistent test cases along the testing with consistent test cases (manifestation of disconfirmatory behaviour leading to *Both* in Table 3), for higher priority functionalities or scenarios, does not

imply improved completeness of a suite design/execution. Since, the testing is still limited to the higher priority items.

Requirements and Agile Software Development: The participants of this study belonged to projects that apply agile software development method Scrum. Hence, the antecedents of the *requirements* category (*ambiguous, incomplete requirements*) may be confined to this software development method. In other words, the emergence of this category suggests a high dependency on Scrum. P2 stated: “*The actual reason, why in this project agile scrum is used, customer sometimes might give you less details, less detailed requirements. So, this causes some issues. Also, you might forget to get [obtain] enough detailed requirements. So, this is a very normal situation. It happens in all projects. That’s why agile scrum is used in this project*”. Paetsch *et al.* stated that agile software development is more adaptive to frequent changes, and is more reliant on direct collaboration instead of documentation oriented processes [36]. As a result, agile is more “code-oriented” and less “document-centric” [36]. Therefore, the requirements to be implemented in the following sprint might not be comprehensive for testers to design complete test suites.

Clarifying Requirements: This antecedent of *Requirements* category promotes *Both* behaviours but it may not always be possible to clarify requirements, e.g., in case of time pressure.

Detection of Errors and Automated Testing: Detection of errors does not promote any kind of behaviour among test automation engineers. The reason for this could be that automated testing, compared to manual testing, do not require an active involvement of a tester during the test suite execution. Once the complete script is run, the results are generated, which are then investigated by the automation tester. The dependency of automated suites on the manual suites may also be a reason for this observation, i.e., the automation testers may find/receive a complete manual test suite to automate.

We could not observe the level of automation as an antecedent to the behaviours of testers performing automated-testing. According to the results, automated suites are developed based on manual suites. Therefore, the modules that are not fully automated, are possibly manually tested. Nonetheless, a possible effect of (the level of) automation is mentioned in Section 4.6, i.e., the fast execution time of automated testing creates time for the (manual) execution of difficult-to-automate inconsistent test cases and other modules that could not be automated. This may promote disconfirmatory behaviour among manual testers.

The analysis also could not support the effect of testing-tools on the behaviour of testers. The participants reported using the tools for maintaining test suites, designing and execution of test suites, test cases statuses, assignments of test cases to others (e.g., to developers for fixing), progress tracking, generating test reports and having a shared platform. These support the testing process, either manual or automated. This may not affect a tester’s (dis)confirmatory behaviour except, for example, the stage of testing (designing or execution) or other reported antecedents (Section 4) inclusive of the general and specific behaviours. For example, P1 reported that the tool they use does not affect their (dis)confirmatory behaviour.

Time Pressure: The data analysis also suggests that *time pressure* leads to *high priority* testing, whether a manual or

automated testing. It is also found to affect the practice of exploratory testing that some testers perform, e.g., in the case of *detection of errors*. Moreover, the participants have reported applying *experience* under time pressure for performing effective testing. Based on the experience, they choose the execution of test scenarios that can be either consistent, inconsistent or both. As P2 explained: “*when time is pushing and both of things [time and previous experience]... You can choose which parts to test. This is like an instinct from your previous experiences and history of the project. You know things [functionalities] and then you find the best approach...*”. Furthermore, the participants attributed limited time availability to Agile practices. A study by Linß *et al.* found ten antecedents to, and five consequences of time pressure, by analysing time pressure in software projects that apply Scrum [37]. It is evidence that time pressure is intrinsic to the agile development method - scrum.

Fig. 2 presents the integrative diagram of the formed categories based on the identified antecedents. The antecedents are separated with a semicolon (;) inside a category box, followed with their classification, e.g., ‘D: Detection of Errors’. The arrows depict relations among the categories, i.e., how one category is influencing the other, e.g., customers define priorities for platform or functionalities (*priority*) when they perform reviews (*external party reviews*). This is indicated by an arrow sign from the *Test Suite Reviews* category to the *Priority* category. The relations between the categories are based on the relations between the antecedents of those categories. These relations are a figurative depiction of the narrative in the results and discussion section. In the figure, the *time pressure* can be seen as impacting other categories by promoting confirmation bias and limiting the completeness of test suite design or execution. The *time pressure* is also diminishing the possibility of exploratory testing, thus decreasing the disconfirmatory effects of the antecedents of *test execution feedback*. In the holistic perspective, the *experience* has emerged as a decisive category for the specific or general behaviour manifestation for other categories, and a contributor to the *priority* category.

Conclusively, confirmation bias is manifested due to the confirmatory antecedents because consistent test cases are designed/executed relatively more than the inconsistent test cases. The antecedents that lead to the disconfirmatory behaviour, and also to a complete test suite (design or execution), suggest the possible mitigation of confirmation bias. Fischhoff mentions five levels of debiasing⁷ interventions: a) warning about the possible bias, b) describing the direction of the typically observed bias, c) personalised feedback, d) training for cognitive mastery and e) debias the task instead of the person [10], [39]. a, b and c are seldom effective, and d is expensive, hence, Fischhoff proposed e [10], [39]. In this perspective, a few of our disconfirmatory antecedents are task/practice-oriented, e.g., *complement testing* that may debias confirmation bias. The antecedents that mostly promote both behaviours for testers, e.g., *change request*, lead to improved completeness of a test suite, if not interrupted by *time pressure*. When time pressure occurs, it affects the completeness of either consistent test scenarios

7. It refers to preventing or alleviating the effects of cognitive biases [10], [38].

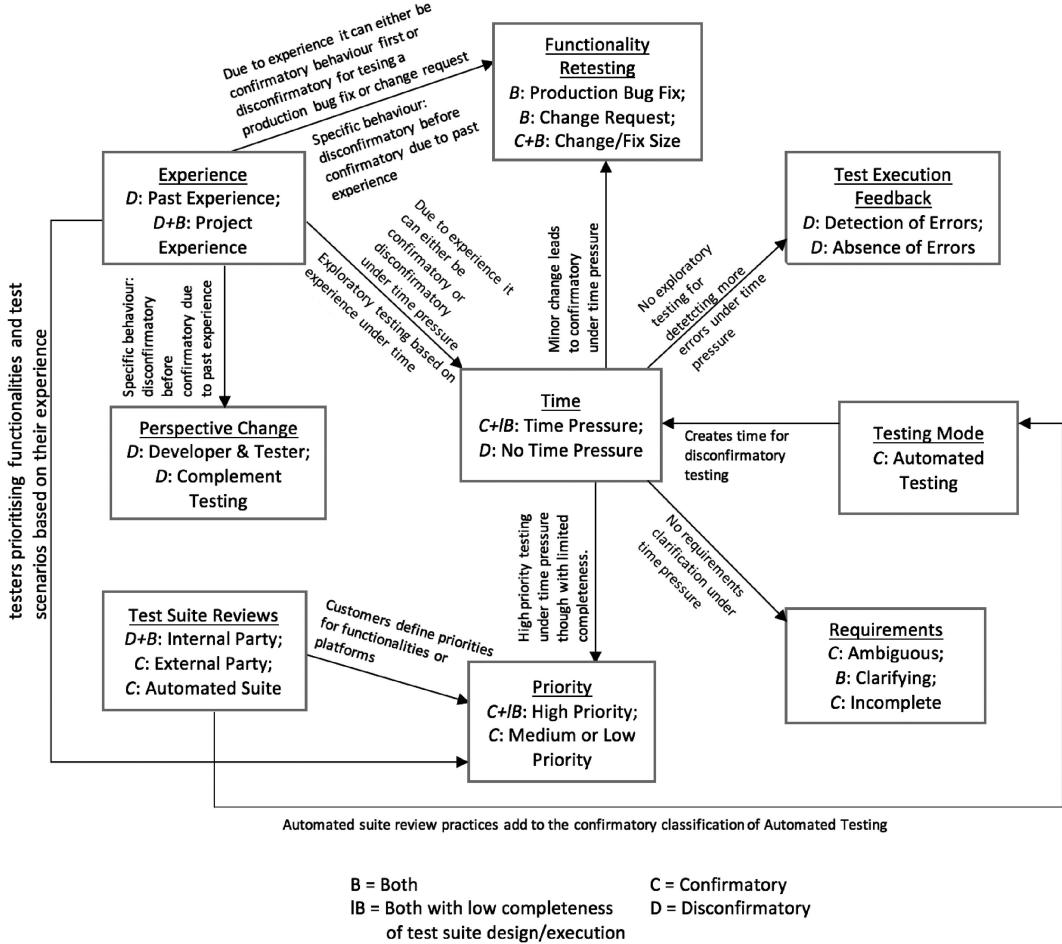


Fig. 2. Integrative diagram depicting the relationships of the identified categories and antecedents to (dis)confirmatory behaviour.

or inconsistent test scenarios, which is respective to the general or specific behaviour of a tester. The specific behaviour (first disconfirmatory) in such a situation although could mitigate confirmation bias but may not assure a defect-free SUT. Since, the consistent test cases are not executed by the tester, that may fail.

5.2 Comparison With Existing Literature

Table 4 presents a comparison of the identified antecedents to the antecedents found in the existing literature. Five out of 12 antecedents of the existing literature are comparable to seven of the antecedents of our study. However, the existing literature does not empirically support the effect of two of the five antecedents on confirmation bias. Our study identified 13 new antecedents compared to the existing literature. There are seven antecedents that the existing literature investigated, but our study could not identify them. However, the existing literature does not empirically support the effect of four of these 7 antecedents on confirmation bias. Table 4 uses different symbols ('E', ↓) for presenting the effect of antecedents because the existing literature uses different assessment methods for measuring confirmation bias. Additionally, our study is a qualitative study and the existing literature are quantitative studies. No effect symbol ('E', ↓) before the antecedent indicates that the existing literature could not experimentally observe it to affect confirmation bias.

The category *experience* and the related antecedents of the existing literature are similar because they all point towards the possible mitigation of confirmation bias due to experience of the testers. The antecedent, *completeness of specifications*, which led to the lower levels of confirmation bias in the studies of Leventhal *et al.* and Teasley *et al.*, is relatable to A6 of our study [6], [7]. Since, the results of A6 may lead to the complete elaboration of all the required and non-required functional behaviour of the SUT. Leventhal *et al.* and Teasley *et al.* defined three levels of specifications [6], [7]. The first and second levels, minimal and positive only specifications are similar to A7 because they all suggest a manifestation of confirmation bias. The effect of the antecedent, *error feedback* was investigated in the context of the presence of errors versus absence of errors by Leventhal *et al.*, which remained inconclusive [6]. Contrary to the hypothesised effect of *error feedback* by Leventhal *et al.* that absence of errors may not decrease confirmation bias levels [6], our study suggests that not finding any error (*absence of errors*) also promote the disconfirmatory or code-breaking behaviour among testers. The disconfirmatory behaviour manifestation in our study may be attributed to the extensive industrial testing experience of the participants. Whereas, Leventhal *et al.* employed graduate students to represent advanced testers, whose maximum professional experience did not exceed over a year as an intern-programmer [6]. Our study showed that *time pressure* is a confirmatory antecedent as well as an antecedent that is

TABLE 4
Comparison With Existing Literature

No.	Our Study	Existing Literature
	<i>Experience</i>	↓ Expertise level [6], [7]; ↓ Experience and activeness in testing and development [5], [8]
A1	D: Past Experience	
A2	D+B: Project Experience	
	<i>Priority</i>	
A3	C+IB: High Priority	-
A4	C: Medium or Low Priority	-
	<i>Requirements</i>	
A5	C: Ambiguous Requirements	-
A6	B: Clarifying Requirements	↓ Completeness of Specifications [6], [7]
A7	C: Incomplete Requirements	
	<i>Functionality Retesting</i>	
A8	B: Production Bug Fix	-
A9	B: Change Request	-
A10	C+B: Change/Fix Size	-
	<i>Test Suite Reviews</i>	
A11	D+B: Internal Party Review	-
A12	C: External Party Review	-
A13	C: Automated Test Suite Review	-
	<i>Testing Mode</i>	
A14	C: Automated Testing	-
	<i>Test Execution Feedback</i>	Error Feedback [6]
A15	D: Detection of Errors	
A16	D: Absence of Errors	
	<i>Time</i>	
A17	C+IB: Time Pressure	Time Pressure [3]
A18	D: No Time Pressure	-
	<i>Perspective Change</i>	
A19	D: Developer & Tester	-
A20	D: Complement Testing	-
	-	E: Company Culture (of different geographic regions) [23]
	-	↓ Logical Reasoning Skills [5], [8]
	-	↓ Job Titles (researchers versus tester, developer, analyst) [5], [8]
	-	Development Methods (e.g., incremental, agile and TDD) [5], [8]; TDD versus TLD [9]
	-	Company size (large, small and medium enterprises) [5], [8]
	-	Educational Background (undergraduate) [5], [8]
	-	Educational Level (bachelor's versus master's) [5], [8]

Key: E = effect on confirmation bias; ↓ = decrease level of confirmation bias

ineffective for testing from the test completeness perspective. However, Salman *et al.* could not find it as a promoting factor for confirmation bias in their experimental study [3].

The participants of our study did not refer to their *logical reasoning skills* (acquired through their education) or *educational background and levels* for their (dis)confirmatory behaviours, in contrast to the evidence shown by Calikli and Bener [5], [8]. The antecedent, *job title* cannot be compared with any antecedent of our study because we did not segregate based on roles. We considered all roles performing core testing oriented activities, e.g., test suite designing and test suite executions, as testers. Additionally, all the participants were practitioners, therefore comparison with *researcher* aspect is impossible. *Company culture*, *company size* and *development methods* are also not directly comparable with any of our

antecedents because our data collection was limited to one company only and none of our participants was solely a developer. It is important to mention that the antecedents with no effect on confirmation bias levels, in the existing literature, are due to not statistically significant results. The discussion on the observed effect sizes of those antecedents, which may imply a possible effect, is beyond the scope of this study.

5.3 Implications for Research and Practice

For research, we propose a multi-case study to explore, whether the antecedents found in this study also hold in other settings because the results of our study are confined to the testers of one company only. A cross-case analysis would also aid towards finding the influence of the antecedents that are particular to the companies, e.g., organisational culture,

development methods, company size, as per quantitatively investigated by Calikli and Bener [5], [8]. More studies applying grounded theory using multiple data collection methods, for exploring the same phenomenon, may reveal new antecedents with more intense evidence. According to our results, ambiguous and incomplete requirements promote a disconfirmatory behaviour, however, under time pressure this leads to a confirmatory behaviour manifestation. More studies are needed on how to improve the requirement specifications that may deteriorate software quality especially in the context of Agile that also constraints time [37]. Yet, the manifestation of confirmation bias in the case of complete requirements is not detrimental for software quality because a tester is then validating all the specified required and not required behaviours of the SUT [3]. In the context of Agile, whether to improve the requirements or to devise solutions to manoeuvre the possibly limited time, is a question that needs scientific attention.

Experimental studies and experimental replications would help strengthen the evidence quantitatively of the identified antecedents. Experimental studies may also help find the relative importance of the identified disconfirmatory antecedents for effective testing, e.g., presence of errors versus absence of errors, and how influential is the role of experience (general experience in testing versus project/domain experience) in this comparison.

We recommend the following *to practitioners:*

Test Suite Reviews: It is important to implement internal test suite review practices if they are not already in place. It is critical that manual test suites are reviewed by the team members of the same project. The same project members are better able to promote disconfirmatory behaviour and also enhance the test suite completeness because they are knowledgeable on the project or domain of the SUT. Despite the review by customers, of the manual suites, internal reviews should still be conducted because customers may focus only on defining the priorities of the functionality rather than promoting a disconfirmatory behaviour. Once the quality of manual suites is assured - a suite that is disconfirmatory and improved in completeness, the dependence of automated suites on manual suites may not be deteriorating for the quality of testing. However, expert test automation engineers should review the automated suites to help less-experienced automation engineers to develop complex test cases especially the inconsistent ones. This could improve the coverage of inconsistent test cases alongside the learning and manifestation of disconfirmatory behaviour by test automation engineers.

The recommended practice of test suite reviews may also be interrupted by time pressure. In such a case, testers with *project (specific) experience* or practice of *complement testing* may cover for skipping test suite reviews. Project experienced testers could be able to achieve possible completeness in designing/executing a test suite(s). The practice of complement testing, i.e., test case designing and execution by two different testers may accommodate more/missing inconsistent test cases to the suite. Thus, ensuring improved completeness for test suite execution.

Experience and Test Execution Feedback: Modules developed by experienced testers may appear less defective or defect-free to inexperienced testers. These modules should be tested by experienced testers because the apparent

absence of errors may prompt more code-breaking (disconfirmatory) behaviour among them compared to inexperienced testers. This may lead to enhanced coverage of inconsistent test cases for the module, which may also reveal errors.

Time and Complete Test Execution: In order to increase test suite execution in terms of (in)consistent test cases, under time pressure, manual test engineers should work in collaboration with automation test engineers. For example, automation engineers run the automated test cases and manual testers run the cases that could not be automated for the SUT. This collaboration may make efficient use of the limited available time with an improved test suite execution. The collaboration may also support other situations that may suffer due to time pressure, e.g., complete execution of inconsistent (manifestation of disconfirmatory behaviour) and not high priority test cases. Automation can be run for not high priority test cases, and manual testers can validate the rest of the functionalities and test cases. Functionality retesting may also benefit from the collaboration in the same manner for time-pressured situations.

5.4 Evaluating the Grounded Theory

We evaluate our grounded theory presented in Fig. 2 according to the Glaserian evaluation criteria [32], [40].

One aspect to evaluate *fit* of the theory is its ability to explain the realities of the studied phenomenon as per viewed by the participants [40]. We shared the generated theory with the participants of our study. The participants found that the identified antecedents and their relationships represent their testing experience, as per said by P6: “*factors [antecedents] are covering my testing experience*”. The respondents also mentioned that the theory also explains the effects of the antecedents on their testing behaviour, especially regarding the disconfirmatory antecedents. This relates to the *work* criterion of the evaluation [32]. In our case, *relevance* relates to the theory’s appeal for practitioners [41]. We achieved it based on the feedback of the participants, as they agreed with the identified antecedents and their inter-relationships in comparison with their testing experience. The last criterion is *modifiability*, which suggests that the theory is flexible to accommodate variations proposed by new data [32], [40]. We were able to modify our theory as we progressed with the analyses of data. Modifiability continued to appear at two points, first, during the classification of the antecedent as (dis)confirmatory and *both*. Second, during the analyses of the relationships among the emerging categories and concepts (antecedents) because of our additional, *why & how* focus of the analyses.

5.5 Threats to Validity

This section elaborates on the threats to validity of our study.

Transcription of the interviews was affected by the accent of the interviewees because they were non-native English speakers. Therefore, we sent summaries of the transcribed content to the respective interviewees for a confirmation on the collected data. However, we received only one response that confirmed the content, other participants did not respond. The interviews conducted without video may not have provided lower quality data because there is not enough empirical evidence to support this possible threat to the data quality [42]. The pilot interview ensured that we collect the right data during the actual

interviews. The percentage of disregarded data (Section 3.3) is minor, we do not believe that it could have caused major threats to our results. We achieved code saturation while coding our data. Code saturation is achieved when code book stabilises, i.e., the data do not suggest any further issues (codes), which in our case are the antecedents [43]. According to Hennink *et al.*, it is possible to achieve code saturation by as few as nine interviews [43]. Our participants were chosen based on a common criterion of experience of software testing; this introduces homogeneity in our sample [44]. Therefore, twelve interviews have been sufficient to achieve (code) saturation when the study's objective is to describe the behaviour of a comparatively homogeneous group [44].

In order to mitigate the potential threat of researcher bias while forming an initial list of the terms that indicated dis(confirmatory) behaviour on the tester's (participant's) end, we also coded the evidence that explained participants' understanding of the terms. The initial list of the terms could be a source of bias for the other coders while performing the pilot coding - Section 3.3. However, the low agreement levels between the coders of the pilot coding, do not suggest such a possibility. According to Urquhart, a researcher applying GT should not have "*preconceived theoretical ideas before starting the research*" [31, p. 16]. In our opinion, familiarity with the relevant literature beforehand has not compromised this characteristic of GT because only limited literature is available. Our study is the first that has explored the *why* and *how* aspect of confirmation bias in software testing. Additionally, the identification of 13 new antecedents and absence of seven existing antecedents from our generated theory (Section 5.2) suggest less influence of any preconceived ideas. We acknowledge possible threats to our theory for not performing theoretical sampling because it was not practically possible. Theoretical sampling enables to address the gaps in the emerging theory [31], [32]. It also helps to increase the scope of the theory by sampling other substantive areas [31]. According to Eisenhardt and Graebner, theoretical sampling refers to the selection of cases that are specifically appropriate for "illuminating and extending relationships and logic among constructs [45, p. 27]". From this aspect, theoretical sampling was implicitly applied from the beginning when our contact person sampled the professionals based on their characteristic, i.e., experience in software testing. However, these software testers belong to a single context.

Our theory is generated from the data of a single company only, i.e., the testers of the Company-ICT, which limits the applicability of the theory to dissimilar contexts. However, a properly performed grounded theory approach produces a theory that is flexible and *modifiable* (GT evaluation criterion) [32], [40]. It can be modified using CCM (a key component of GT) based on the data from other studied contexts [31], [32], [40]. With reference to the concept of *biasplexes* (Section 2), confirmation bias belongs to the *inertia* biasplex [17]. The other cognitive biases of this biasplex are, e.g., the bandwagon effect and anchoring bias [17], [38]. These other biases may also reinforce or overlap with confirmation bias among software testers to form their (dis)confirmatory behaviour. Our study is limited to exploring the phenomenon of confirmation bias without considering its biasplex. Furthermore, our study does not employ data triangulation to improve the strength of evidence, instead is limited to a single data collection method, i.e., interviews.

However, this study can be considered a post-hoc approach to explore further the phenomenon of confirmation bias among testers because we observed its manifestation in our previous experimental study with student-participants as testers [3].

6 CONCLUSION AND FUTURE WORK

We applied grounded theory to explore the antecedents to confirmatory and disconfirmatory behaviours and to understand how they occur among software testers. We identified twenty antecedents to (dis)confirmatory behaviour, classified in nine categories; experience, priority, requirements, functionality retesting, test suite reviews, test execution feedback, time, testing mode and perspective change.

The antecedents that promote confirmatory behaviour, leading to confirmation bias are; ambiguous requirements, incomplete requirements, high priority testing, medium or low priority testing, automated testing, automated test suite reviews, external party reviews, (minor) change/fix size and time pressure. Time pressure plays an important role in the occurrence of confirmatory behaviour among testers, e.g., when they are dealing with ambiguous requirements or performing only a high priority testing.

The antecedents that promote disconfirmatory behaviour and also improve the completeness of a test suite from design and execution perspective are; project experience, past experience, developer and tester perspective, complement testing, detection of errors, absence of errors, no time pressure and internal party reviews. These antecedents may help circumvent confirmation bias and improve the quality of testing. Practitioners are recommended to implement internal party reviews because it may increase the completeness of inconsistent test cases. Similarly, a practice of complement testing, among the testers, may also help in the completeness of inconsistent test cases and increase defect detection. Defect detection (detection of errors), in turn, promotes disconfirmatory behaviour, as propositioned by our grounded theory.

The future work of this study includes the extension and modification of our theory with the data from testers of other companies. In other words, to increase the generality of the theory by sampling other substantive areas. Data triangulation through multiple data collection methods would also increase the validity of findings. For example, conducting an observational study that observes testers over a longer span in addition to interviews would help validate the findings from different sources. This would enable an in-depth analysis of confirmation bias phenomenon, also considering its interactions with other cognitive biases, and thus the behaviour of software testers. Another possible extension of this work is to quantitatively investigate the relative importance of the identified antecedents, in a software testing context.

ACKNOWLEDGMENTS

The authors would like to thank the participants of the Company-ICT for this study; Alper Corlan, Basak Kahraman, Berkay Sertoglu, Cagatay Ince, Elif Deniz, Emin Vilgenoglu, Gulden Karakoyun, Ozgul Ozcan, Selda Aydin, Sezen Kaya, Sinan Verdi, and Ugur Ozcan. This study was supported in part by the Infotech Oulu Doctoral Grant at the University of Oulu to Iflaah Salman.

REFERENCES

- [1] D. Arnott, "Cognitive biases and decision support systems development: A design science approach," *Inf. Syst. J.*, vol. 16, no. 1, pp. 55–78, 2006.
- [2] L. M. Leventhal, B. Teasley, D. S. Rohlman, and K. Instone, "Positive test bias in software testing among professionals: A review," in *Proc. Int. Conf. Hum.-Comput. Interaction*, 1993, pp. 210–218.
- [3] I. Salman, B. Turhan, and S. Vegas, "A controlled experiment on time pressure and confirmation bias in functional software testing," *Empir. Softw. Eng.*, vol. 24, no. 4, pp. 1727–1761, Dec. 2018. [Online]. Available: <http://link.springer.com/10.1007/s10664-018-9668-8>
- [4] G. Calikli and A. Bener, "Influence of confirmation biases of developers on software quality: An empirical study," *Softw. Qual. J.*, vol. 21, no. 2, pp. 377–416, 2013. [Online]. Available: <http://link.springer.com/10.1007/s11219-012-9180-0>
- [5] G. Calikli and A. Bener, "Empirical analysis of factors affecting confirmation bias levels of software engineers," *Softw. Qual. J.*, vol. 23, pp. 695–722, 2015. [Online]. Available: <http://link.springer.com/10.1007/s11219-014-9250-6>
- [6] L. M. Leventhal, B. E. Teasley, and D. S. Rohlman, "Analyses of factors related to positive test bias in Software Testing," *Int. J. Hum.-Comput. Stud.*, vol. 41, pp. 717–749, 1994.
- [7] B. E. Teasley, L. M. Leventhal, C. R. Mynatt, and D. S. Rohlman, "Why software testing is sometimes ineffective: Two applied studies of positive test strategy," *J. Appl. Psychol.*, vol. 79, no. 1, 1994, Art. no. 142.
- [8] G. Calikli and A. Bener, "Empirical analyses of the factors affecting confirmation bias and the effects of confirmation bias on software developer/tester performance," in *Proc. 6th Int. Conf. Predictive Models Softw. Eng.*, 2010, Art. no. 10. [Online]. Available: <http://portal.acm.org/citation.cfm?doid=1868328.1868344>
- [9] A. Causevic, R. Shukla, S. Punnekatt, and D. Sundmark, "Effects of negative testing on TDD: An industrial experiment," in *Proc. Int. Conf. Agile Softw. Develop.*, 2013, pp. 91–105. [Online]. Available: http://link.springer.com/chapter/10.1007/978-3-642-38314-4_7
- [10] R. Mohanani, I. Salman, B. Turhan, P. Rodriguez, and P. Ralph, "Cognitive biases in software engineering: A systematic mapping study," *IEEE Trans. Softw. Eng.*, 2018. [Online]. Available: <https://ieeexplore.ieee.org/document/8506423>
- [11] T. Gilovich, D. Griffin, and D. Kahneman, *Heuristics and Biases*, 8th ed. Cambridge, U.K.: Cambridge Univ. Press, 2002.
- [12] A. Tversky and D. Kahneman, "Judgement under uncertainty: Heuristics and biases," *Oregon Res. Inst. Res. Bull.*, vol. 13, no. 1, 1973. [Online]. Available: <https://doi.org/10.1126/science.185.4157.1124>
- [13] D. Arnott, "A taxonomy of decision biases," Monash University, Australia, 1998. [Online]. Available: <http://www.sims.monash.edu.au/staff/darnott/biastax.pdf>
- [14] D. Arnott and S. Gao, "Behavioral economics for decision support systems researchers," *Decis. Support Syst.*, vol. 122, no. Feb., 2019, Art. no. 113063. [Online]. Available: <https://doi.org/10.1016/j.dss.2019.05.003>
- [15] D. Kahneman, D. Lovallo, and O. Sibony, "Before you make that big decision..." *Harvard Bus. Rev.*, vol. 89, no. 6, pp. 50–60, 2011. [Online]. Available: <http://website.aub.edu.lb/units/ehmu/Documents/before-you-make-that-big-decision.pdf>
- [16] I. Salman, "The effects of confirmation bias and time pressure in software testing," Ph.D. dissertation, Fac. Inf. Technol. Elect. Eng., Univ. Oulu, Oulu, Finland, 2019.
- [17] P. Ralph, "Possible core theories for software engineering," in *Proc. 2nd SEMAT Workshop General Theory Softw. Eng.*, 2013, pp. 35–38.
- [18] K. A. de Graaf, P. Liang, A. Tang, and H. van Vliet, "The impact of prior knowledge on searching in software documentation," in *Proc. ACM Symp. Document Eng.*, 2014, pp. 189–198.
- [19] E. D. Smith, Y. J. Son, M. Piattelli-Palmarini, and A. Terry Bahill, "Ameliorating mental mistakes in tradeoff studies," *Syst. Eng.*, vol. 10, no. 3, pp. 222–240, 2007. [Online]. Available: <http://doi.wiley.com/10.1002/sys.20072>
- [20] G. Calikli, A. Bener, T. Aytac, and O. Bozcan, "Towards a metric suite proposal to quantify confirmation biases of developers," in *Proc. Int. Symp. Empir. Softw. Eng. Meas.*, 2013, pp. 363–372.
- [21] R. S. Nickerson, "Confirmation bias: A ubiquitous phenomenon in many guises," *Rev. General Psychol.*, vol. 2, no. 2, pp. 175–220, 1998.
- [22] G. Calikli, B. Arslan, and A. Bener, "Confirmation bias in software development and testing : An analysis of the effects of company size, experience and reasoning skills," in *Proc. 22nd Annu. Psychol. Program. Interest Group Workshop*, 2010. [Online]. Available: <https://ppig.org/files/2010-PPIG-22nd-Calikli.pdf>
- [23] G. Calikli, A. Bener, and B. Arslan, "An analysis of the effects of company culture, education and experience on confirmation bias levels of software developers and testers," in *Proc. ACM/IEEE 32nd Int. Conf. Softw. Eng.*, 2010, vol. 2, pp. 187–190.
- [24] S. Eldh, "On test design," Ph.D. dissertation, School Innov., Des. Eng., Mälardalen University, Västerås, Sweden, 2011.
- [25] A. A. Jorgensen and J. A. Whittaker, "How to break software," 2000. [Online]. Available: https://www.researchgate.net/publication/315700027_How_to_Break_Software_with_examples
- [26] J. A. O. G. Da Cunha and H. P. De Moura, "Towards a substantive theory of project decisions in software development project-based organizations: A cross-case analysis of IT organizations from Brazil and Portugal," in *Proc. 10th Iberian Conf. Inf. Syst. Technol.*, 2015, pp. 1–6.
- [27] J. A. O. Cunha, H. P. Moura, and F. J. Vasconcellos, "Decision-making in software project management: A qualitative case study of a private organization," in *Proc. 9th Int. Workshop Cooperative Hum. Aspects Softw. Eng.*, 2016, pp. 26–32.
- [28] S. Chakraborty, S. S. Sarker, and S. S. Sarker, "An exploration into the process of requirements elicitation : A grounded approach," *J. Assoc. Inf. Syst.*, vol. 11, no. 4, pp. 212–249, 2010.
- [29] I. Hadar, "When intuition and logic clash: The case of the object-oriented paradigm," *Sci. Comput. Program.*, vol. 78, no. 9, pp. 1407–1426, 2013.
- [30] P. Conroy and P. Kruchten, "Performance norms: An approach to rework reduction in software development," in *Proc. 25th IEEE Can. Conf. Electr. Comput. Eng.*, 2012, pp. 1–6.
- [31] C. Urquhart, *Grounded Theory For Qualitative Research: A Practical Guide*. Thousand Oaks, CA, USA: SAGE, 2012.
- [32] K.-J. Stol, P. Ralph, and B. Fitzgerald, "Grounded theory in software engineering research," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 120–131.
- [33] S. Baltes and P. Ralph, "Sampling in software engineering research: A critical review and guidelines," *ACM Trans. Softw. Eng. Methodol.*, vol. 21, 2020. [Online]. Available: <https://arxiv.org/pdf/2002.07764.pdf>
- [34] H. R. Boeije, "A purposeful approach to the constant comparative method in the analysis of qualitative interviews," *Qual. Quant.*, vol. 36, pp. 391–409, 2002.
- [35] J. L. Campbell, C. Quincy, J. Osserman, and O. K. Pedersen, "Coding in-depth semistructured interviews: Problems of unitization and intercoder reliability and agreement," *Sociol. Methods Res.*, vol. 42, no. 3, pp. 294–320, 2013.
- [36] F. Paetsch, A. Eberlein, and F. Maurer, "Requirements engineering and agile software development," in *Proc. 12th IEEE Int. Workshops Enabling Technol. Infrastructure Collaborative Enterprises*, 2003, pp. 1–6.
- [37] S. Linßen, D. Basten, and J. Richter, "Antecedents and consequences of time pressure in scrum projects : Insights from a qualitative study," in *Proc. 51st Hawaii Int. Conf. Syst. Sci.*, 2018, pp. 4835–4844.
- [38] P. Ralph, "Toward a theory of debiasing software development," *Lecture Notes Bus. Inf. Process.*, vol. 93, pp. 92–105, 2011. [Online]. Available: <http://link.springer.com/10.1007/978-3-642-25676-9>
- [39] B. Fischhoff, "Debiasing," in *Judgment Under Uncertainty: Heuristics and Biases*, D. Kahneman, P. Slovic, and A. Tversky, Eds. Cambridge, MA, USA: Cambridge Univ. Press, 1982.
- [40] R. Hoda, J. Noble, and S. Marshall, "Self-organizing roles on agile software development teams," *IEEE Trans. Softw. Eng.*, vol. 39, no. 3, pp. 422–444, Mar. 2013.
- [41] B. Chametzky and J. College, "Generalizability and the theory of offsetting the affective filter," *Grounded Theory Rev.*, vol. 12, no. 2, pp. 35–43, 2013.
- [42] G. Novick, "Is there a bias against telephone interviews in qualitative research?" *Res. Nursing Health*, vol. 31, no. 4, pp. 391–398, 2008.
- [43] M. M. Hennink, B. N. Kaiser, and V. C. Marconi, "Code saturation versus meaning saturation: How many interviews are enough?" *Qualitative Health Res.*, vol. 27, no. 4, pp. 591–608, 2017.
- [44] G. Guest, A. Bunce, and L. Johnson, "How many interviews are enough?: An experiment with data saturation and variability," *Field Methods*, vol. 18, no. 1, pp. 59–82, 2006.
- [45] K. M. Eisenhardt and M. E. Graebner, "Theory building from cases: Opportunities and challenges," *Acad. Manage. J.*, vol. 50, no. 1, pp. 25–32, 2007.



Iflaah Salman received the MSc and PhD degrees in information processing science from the University of Oulu, Oulu, Finland, in 2014 and 2019, respectively. She is a postdoctoral research fellow at Empirical Software Engineering in Software, Systems and Services (M3S) research unit at the University of Oulu, Oulu, Finland. Her research interests include empirical software engineering, cognitive aspects, and software testing. Previously, she worked as a software quality assurance engineer (2010 to 2012)

at i2c inc., Lahore, Pakistan. For more information please visit <https://www.linkedin.com/in/iflaahsalman/> and follow on <https://www.researchgate.net/>.



Pilar Rodríguez received the BSc, MSc, and PhD degrees in computer science, in 2006, 2008, and 2013, respectively. She is currently an assistant professor at Universidad Politécnica de Madrid, Spain and docent at the University of Oulu, Finland. Her research centers on empirical software engineering, software processes with a particular focus on agile software development, value-based software engineering, human factors in software engineering, and software quality. She has published in premier software engineering journals and conferences. She has served on the program committee for conferences such as ESEM, EASE, and XP, on the review boards of journals such as the *IEEE Transaction on Software Engineering*, and the *Empirical Software Engineering*, and as an organization committee member for conferences such as ICSE and ESEM.



Burak Turhan (Member, IEEE) received the PhD degree from Boğaziçi University, Turkey. He is an associate professor with the Faculty of Information Technology, Monash University, Australia, and an adjunct professor with the University of Oulu, Finland. His research focuses on empirical software engineering, software analytics, quality assurance and testing, human factors, and (agile) development processes. He has published more than 110 articles in international journals and conferences, received several best paper awards, and secured funding for several large scale research projects. He has served on the program committees of more than 30 academic conferences, on the editorial/review boards of several journals including the *IEEE Transactions on Software Engineering*, *Empirical Software Engineering*, *Journal of Systems and Software*, *Information and Software Technology*, and *Software Quality Journal*, as (co-)chair for PROMISE'13, ESEM'17, and PROFES'17, and as a steering/organization committee member for PROMISE, ESEM, and ICSE. He is a member of the ACM, ACM SIGSOFT, and IEEE Computer Society. For more information please visit <https://turhanb.net>.



Ayşe Tosun (Member, IEEE) received the MSc and PhD degrees from the Department of Computer Engineering, Bogazici University, Turkey, in 2008 and 2012, respectively. She is an assistant professor at the Faculty of Computer and Informatics Engineering, and executive board member at Artificial Intelligence and Data Science Applied Research Center at Istanbul Technical University (ITU), Istanbul, Turkey. Prior to joining ITU, she worked as a postdoctoral research fellow with the Department of Information Processing Science,

University of Oulu, Finland. During her PhD, she worked as a research intern at Software Reliability Lab, Microsoft Research, in Cambridge. Her research interests are empirical software engineering, more specifically mining software data repositories, software measurement, software process improvement, software quality prediction models, and applications of AI on building recommendation systems for software engineering.



Arda Güreler received the BSc degree in mathematical engineering from Yıldız Technical University, Turkey, in 2000, and the MSc degree in business information systems from Bogazici University, Turkey, in 2017. He is a senior researcher at Ericsson Research NAP Turkey, working as Research Project Coordinator. He focused on setting up and management of national and international R&D projects in the ICT sector. He started to work for Ericsson, in 2010, where he specialized in managing industry and university collaborations for R&D innovation projects. He currently focuses on the coordination and reporting of collaborations with customers, universities, EU, etc. as well as handling of the intellectual property rights arising from those collaborations.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

Optimization of Software Release Planning Considering Architectural Dependencies, Cost, and Value

Raghvinder S. Sangwan[✉], Ashkan Negahban[✉], Robert L. Nord[✉], and Ipek Ozkaya

Abstract—Within any incremental development paradigm, there exists a tension between the desire to deliver value to the customer early and the desire to reduce cost by avoiding architectural refactoring and rework in subsequent releases. What is lacking is an analytical framework that quantifies opportunities and risks of choosing one or the other of these strategies or a blend of the two. This article demonstrates the use of design structure and domain mapping matrices for analyzing architectural dependencies and proposes an optimization-based decision-making technique to support effective release planning. The optimization models recommend the order in which architectural elements and features should be implemented across different releases so as to: (a) minimize rework cost; (b) maximize early value delivery; or (c) optimize an integrated measure of cost and value. These analytic models can be applied earlier in the life cycle and, hence, provide timely information about the progress and changes that occur at each iteration.

Index Terms—Software release management and delivery, software architecture, nonlinear programming

1 INTRODUCTION

WITHIN any iterative incremental development paradigm, there is a choice between two competing interests, namely early value delivery, and avoiding architectural refactoring and rework in subsequent releases. In certain contexts, early delivery might be an appropriate choice, for example, to enable the release of critically needed capabilities or to gain market exposure and feedback. In other contexts, delayed release in the interest of reducing later rework might better align with project and organizational drivers and concerns. What is lacking, however, is quantifiable guidance for developers that highlights the potential opportunities and risks of choosing one or the other of these alternatives (or a blend of both).

In iterative release planning, developers must consider a range of dependencies:

- 1). *Dependencies among customer requirements or features (discrete units of functionality desired by stakeholders)*: Understanding these allows for optimization of development activities within a given release and ensures that a coherent and useful feature set is released to the end user [12].

- 2). *Dependencies among features and architectural elements (implementation units of software that provide a coherent set of responsibilities)*: Understanding these allows for a staged implementation of the architecture and supports the delivery of customer value.
- 3). *Dependencies among architectural elements*: An architecture embodies design decisions that influence a system's quality attributes. Changes to an architecture involve modifying a system's gross topology as well as its communication and coordination mechanisms. Therefore, analyzing these dependencies provides insight into potential downstream rework costs that may be incurred as a result of choosing to incrementally develop and release the architectural infrastructure which may have serious implications for the future (successful) evolution of a product [2], [25].

We posit that the ability to quantify architecture quality with measurable criteria provides engineering guidance for iterative release planning. We can improve the visibility of architecture quality by providing quantifiable models of the architecture during system development. These analytic models can be applied earlier in the life cycle (as opposed to using code) and, hence, provide reliable information about the progress and changes that occur at each iteration. The optimization models of software release planning described in this paper provide data-driven decision support, are scalable, and can be applied in any real-world situation using the following iterative process:

- *Step 1*: Determine the dependencies between architectural elements and features.
- *Step 2*: Develop an initial release plan using release planning procedures a team currently follows.
- *Step 3*: Set cost, value, and resource parameters of the optimization models based on the release plan.

• Raghvinder S. Sangwan and Ashkan Negahban are with the School of Graduate Professional Studies, Pennsylvania State University, Malvern, PA 19355 USA. E-mail: {rsangwan, aun85}@psu.edu.

• Robert L. Nord and Ipek Ozkaya are with the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA 15213 USA. E-mail: {rn, ozkaya}@sei.cmu.edu.

Manuscript received 9 Dec. 2019; revised 24 July 2020; accepted 24 Aug. 2020. Date of publication 27 Aug. 2020; date of current version 18 Apr. 2022. (Corresponding author: Raghvinder S. Sangwan.)

Recommended for acceptance by Y. Cai.

Digital Object Identifier no. 10.1109/TSE.2020.3020013

\$root	H	N	ω	Δ
A 1	.		1	1
B 2		.	1	
C 3	1		.	1
D 4				.

Fig. 1. A DSM showing dependencies among software modules.

- Step 4: Solve the models under various what-if scenarios to understand the trade-off between cost and early value delivery.
- Step 5: Make necessary adjustments to the release plan based on results of the what-if scenarios; repeat steps 3 – 5 until satisfied with the resulting plan.

We conduct a study with the goal of exploring distinct outcomes of different development paths when contrasting business goals are at stake: (i) maximizing early value delivery to the end user; and, (ii) minimizing development cost due to rework. We consider three paths: value-driven, cost-driven, and a middle ground of using both value and cost to guide each key decision point in release planning. We analyze how development cost changes from iteration to iteration as we optimize for these different outcomes. Finally, we examine the paths followed by the development teams of the system we studied.

The structure of this article is as follows. In Section 2, we review dependency management as manifested by design structure and domain mapping matrices, and describe our development path analysis approach to release planning. In Section 3, we present the Management Station Lite (MSLite) system that we chose as a model problem for our study. Sections 4 and 5 present the details of our optimization analysis of the model problem. In Section 6, we discuss the validity of our approach, and we compare it with related work in Section 7. Last, in Section 8, we summarize conclusions and future work.

2 DEPENDENCY MANAGEMENT AND PATH ANALYSIS

In this section, we review how to capture each of the three dependencies discussed above using design structure and domain mapping matrices, and discuss the notion of propagation ratio and path analysis.

2.1 Design Structure Matrix

A design structure matrix (DSM) [44] maps dependencies among items in a given domain. All elements appear in both the rows and the columns, and dependencies are signaled at the intersection points of the items in the matrix. For example, Fig. 1 shows a DSM for a software system that has been decomposed into four modules, namely A, B, C, and D. The rows and columns of the matrix represent the same modules. The dependencies of a module are read down a column. For instance, reading down the first column, we can see that Module A depends on Module C. We

		Requirement					
\$root		U	V	W	X	Y	Z
Module	A	1					1
	B		1				
	C	1		.	1		
	D					1	1

Fig. 2. A DMM showing dependencies among customer requirements and software modules.

also see that Module A does not depend on Modules B or D because those cells are empty. The identity diagonal represents a dependency of a module on itself.

DSMs are single-domain square matrices, meaning that relations are defined among instances of the same type (for example, software modules in Fig. 1). However, dependencies also occur across different domains. Examples include dependencies among development staff technical competencies and the software components that they will develop. Another common example is the identification of which software components or modules satisfy which customer requirements. Multi-domain dependencies often cause project delays or even failure when detected too late [10]. We discuss such dependencies next.

2.2 Domain Mapping Matrix

The term domain mapping matrix (DMM) was coined to refer to matrices that map the relations among items in two different product development domains; for example, task X requires person Y's expertise [10], [11]. The two domains need not have the same number of items; thus, the resulting DMM is usually a rectangular matrix. Fig. 2 illustrates an example of a DMM applied to the analysis of dependencies between customer requirements and software modules that implement those requirements.

Our approach uses DSM and DMM analysis to provide information on architecture quality during iterative release planning. Table 1 summarizes the use of DSMs and DMM in this context.

2.3 Propagation Ratio

Architecture quality and visibility are closely related. Reasoning about quality with a quantifiable model requires that certain architectural properties be represented objectively and in a repeatable manner across systems for the model to work. Therefore, we use DSM and DMM analysis to provide a representation with support for objective metrics generation. We discuss the propagation ratio metric in this context. According to [30], this metric captures the percentage of elements that can be affected, on average, when a change is made to a randomly chosen element. Propagation ratio (PR) is calculated as the density of a matrix M as represented by the ratio of the total number of filled cells due to direct or indirect dependencies among its elements ($\sum_{i=0}^n M^i$) to the size of the matrix (n^2):

$$PR = \text{Density}\left(\sum_{i=0}^n M^i\right)/n^2. \quad (1)$$

TABLE 1
DSM/DMM Identification

Features (F)		Architectural Elements (AE)
Features (F)	DSM _F	DMM
Architectural Elements (AE)	-	DSM _{AE}

2.4 Path Analysis

Next, we describe our approach to development path analysis for iterative release planning. A release can be an internal iteration that produces a potentially shippable increment or an external release to the customer. Each release in the path is described by the following attributes:

- Sequence number (the release order in the path)
- Features implemented
- Architectural elements implemented
- The DSM and DMM matrices up to that release.

We calculate end-user value at each release by adding the value of all features supported by that release. For the purpose of our study, value reflects the priority points of the features (explained in Section 3.1). Total cost of a release r , TC_r , is calculated as follows:

$$TC_r = IC_r + RC_r, \quad (2)$$

where the implementation cost, IC_r , is the sum of the implementation cost of all features and architectural elements implemented in release r (and not present in an earlier release). We assume that the implementation cost can be estimated for all individual features and architectural elements (independent of dependencies). Rework cost (RC_r) is incurred when new features and/or architectural elements are added to the system during a release, and one or more preexisting features and/or architectural elements have to be modified to accommodate the new ones. This includes features and architectural elements with direct dependencies on the new features and architectural elements as well as those with indirect dependencies captured by the propagation ratio. We compute the rework cost for release r by adding two components:

- The rework cost associated with each new architectural element implemented in release r that has dependencies from preexisting architectural elements in release $(r - 1)$.
- The rework cost associated with each new feature implemented in release r that has dependencies from preexisting features and architectural elements in release $(r - 1)$.

Within the context of our analysis, we use rework cost to provide insight into the improvement or degradation of architectural quality across releases within a given development path.

3 MSLITE STUDY

We analyze the cost and value outcomes of alternative release strategies using DSM and DMM-based dependency analysis with propagation ratio. We picked the MSLite [4], [39], [41] system for the study, a system with which we have

previous experience and access to the code, architecture, and project-planning artifacts. The methodology used for the study consists of first defining the system requirements and system structure using DSM and DMM analysis. Then, through a set of mathematical optimization models, we perform two types of comparisons and analyze five development paths by which to realize the system requirements and system structure:

- (i) Comparing three development paths using what-if scenarios to understand the space and boundaries of decisions with respect to cost and value:
 - *Path 1*: A cost-driven what-if scenario that focuses on minimizing cost.
 - *Path 2*: A value-driven what-if scenario that focuses on maximizing early value delivery.
 - *Path 3*: An integrated approach and what-if scenario to understand trade-offs between cost and value at key decision points in release planning.
- (ii) Comparing the Planned Path and Actual Path to understand the predictive nature of using the architecture earlier in the life cycle than code can be used:
 - *Planned Path*: Development path of MSLite as it was planned.
 - *Actual Path*: Development path of MSLite as it was actually implemented.

3.1 Goals and Requirements of the MSLite System

MSLite is a system that automatically monitors and controls a building's internal functions, such as heating, ventilation, air conditioning, access, and safety. Fig. 3. shows the primary presentation of the component-and-connector (C&C) view [16] of the MSLite system. The accompanying catalog in the system documentation describes the responsibilities of the components and connectors. The components outside the *MSLite Server* subsystem implement the core user functionality, namely, monitoring building facilities and issuing commands to change their properties (for example, to lower indoor temperature). Detecting alarm conditions and automating rules for property changes are other important capabilities of the system. The main purpose of the components of the *MSLite Server* subsystem is to provide support for the quality requirements. For instance, one of the main system goals is to support multiple field systems for heating, ventilation, air conditioning, lighting, secure access, fire detectors, etc. The *Virtual FSS* component handles all the field systems (FSS) descriptions and creates appropriate events to which the other components in the system can listen. Other components such as *Cache* and *Access Control* explicitly support performance and security, respectively.

The system users are facilities managers that use it to manage and configure a network of hardware-based field systems used for controlling building functions triggering alarms under life-critical situations. We express the functional and quality attribute requirements for MSLite as user stories (US) and acceptance test cases (ATC), respectively, as shown in Table 2, and collectively refer to them as features. On a scale of 1 to 9, we express the priorities of these features according to their relative benefit to the end user when implemented and the penalty incurred by the end user if postponed. Furthermore, we assign a user value to

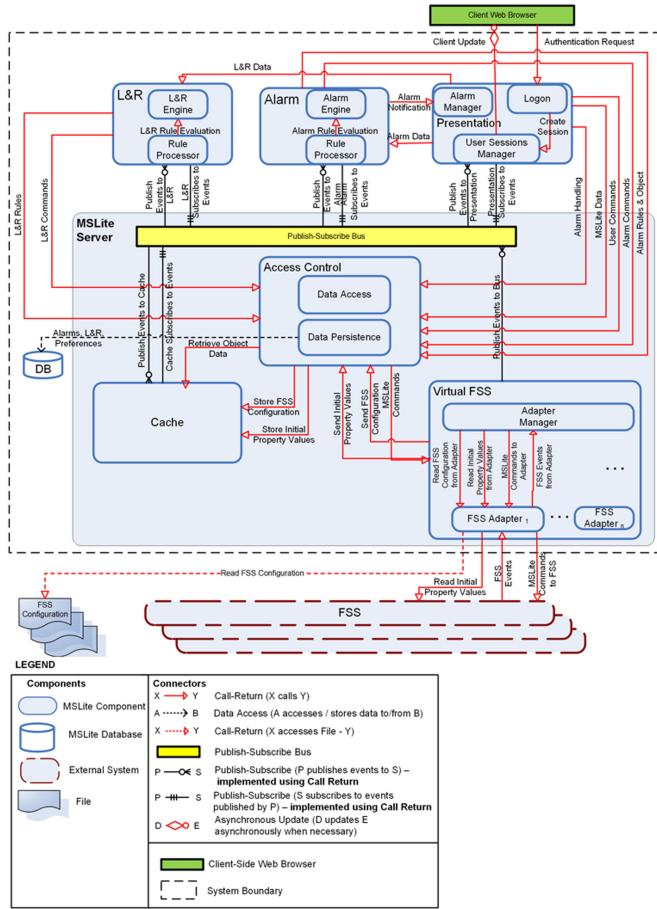


Fig. 3. MSLite top-level C&C architecture view.

each feature as the weighted sum of benefits and penalties, with benefits given a weight two times that for the penalty [49]. Given that customers value delivered functionality, USs have been assigned a higher benefit value than ATCs. The cost column represents relative effort required to develop each feature. Given that ATCs are relatively more difficult to implement, they have been assigned a higher relative cost compared to USs.

3.2 System Definition

We identify interdependencies among MSLite architectural elements (DSM_{AE}), its features (DSM_F), and architectural elements and features (DMM) in Fig. 4. This dependency analysis is used to determine precedence in the implementation of capabilities.

4 OPTIMIZATION OF RELEASE PLANNING

We propose three mixed-integer nonlinear programming (MINLP) models for three strategies:

- Minimizing total cost, i.e., sum of implementation and rework costs (Path 1).
- Maximizing early delivery of value to the end user (Path 2).
- Optimizing an integrated measure of cost and early value delivery (Path 3).

The models are scalable and can handle large problem instances in terms of number of architectural elements and

features, number of releases, and number of non-zero elements in the DSM and DMM. In this section and Section 5, we show the applicability of our models in a real-world project and how they provide decision support in the path definition process.

In this section and Section 5, we show the applicability of our models in a real-world project and how they provide decision support in the path definition process. As shown in Section 5, the models can also be used for performing formal what-if analysis on the trade-off between early value delivery and cost, enabling answering the important question of “how willing is the developer to accept additional rework cost in exchange for the early delivery of features to the customer?” Table 3 summarizes the notations used in the optimization models.

The main decision made by these models is the release number r in which each architectural element j and feature l should be implemented considering the dependency relationships so as to minimize or maximize an objective function. The primary decision variables in these models are:

$$AE_{j,r} = \begin{cases} 1, & \text{if architectural element } j \text{ is implemented in release } r, \\ 0, & \text{otherwise,} \end{cases}$$

$$F_{l,r} = \begin{cases} 1, & \text{if feature } l \text{ is implemented in release } r, \\ 0, & \text{otherwise.} \end{cases}$$

4.1 Path 1: Minimizing Total Cost

The objective function (3) is the cumulative cost of the project calculated as the sum of total cost for each release.

$$\text{Minimize } TC^{Final} = \sum_{r=1}^N TC_r. \quad (3)$$

Due to the large number of constraints and to enhance readability, we will present the constraints in groups. Constraints (4 – 11) pertain to the concepts described in Section 2. Equation (4) calculates the total cost for each release r as the sum of the implementation and rework costs in that release. Equation (5) computes the total implementation cost for release r by adding the implementation cost for architectural elements and features implemented in the release (i.e., we only add $C_{j,r}^{AE}$ and $C_{l,r}^F$ values for which $AE_{j,r} = 1$ and $F_{l,r} = 1$, respectively). There are three types of rework cost due to: (I) violating the dependencies between architectural elements (computed using constraints 6 and 9); (II) violating the dependencies between features (computed using constraints 7 and 10); and, (III) violating the dependencies between features and architectural elements (computed using constraints 8 and 11).

$$TC_r = IC_r + RC_r^{AE} + RC_r^F + RC_r^{AE-F} \quad \forall r \in \{1, 2, \dots, N\} \quad (4)$$

$$IC_r = \sum_{j=1}^{N^{AE}} AE_{j,r} C_j^{AE} + \sum_{l=1}^{N^F} F_{l,r} C_l^F \quad \forall r \in \{1, 2, \dots, N\} \quad (5)$$

$$RC_r^{AE} = \sum_{k=1}^{N^{AE}} \sum_{j=1}^{N^{AE}} AE_{k,r} H_{j,r-1}^{AE} D_{jk}^{AE} C_j^{AE} PR_{r-1}^{AE} \quad \forall r \in \{2, \dots, N\} \quad (6)$$

TABLE 2
MSLite Features

Feature	Feature Description	Relative Benefit	Relative Penalty	Value	Cost	Rationale
US01	Visualize field system properties	9	9	27	2	Without it the facility manager cannot monitor the facilities
US02	Change field system properties	8	9	25	2	Without it the facility manager cannot take corrective action
US03	Add alarm condition	7	9	23	2	The facility manager can still detect abnormal situations by observing field properties, but the penalty of an alarm situation going undetected can be very high
US04	Alarm notification & acknowledgement	7	7	21	1	Without it alarms would have to be monitored manually
US05	Ignore alarm notification	5	6	16	1	Without it important alarms for the user could go unnoticed among other low-priority alarms
US06	Add logic condition and reaction	7	5	19	2	Without it the facility manager would have to monitor those conditions manually
US07	Alarms and logic/reaction pairs persistence	6	5	17	2	Having to re-create alarms and logic/reaction conditions would be cumbersome and tedious
US08	Secure access to system	1	7	9	1	Without it physical security measures would be needed to control access to the workstations running MSLite
ATC14	Connect similar field system	3	8	14	3	This is the most likely case to provide unified field systems management
ATC15	Connect field system using different format and interfaces	1	7	9	4	This case adds no functionality but requires considerable effort
ATC16	Connect field system providing new functionality	4	5	13	4	This case requires effort but can add functionality; however, there is a chance it will never be needed
ATC17	Field system properties update speed	1	7	9	3	Delayed field system properties updates can lead to incorrect user actions and eventually cause alarm situations
ATC18	Alarm notification speed	1	8	10	3	Alarm situations need to be addressed quickly
ATC19	No loss of alarm notifications	1	9	11	4	Lost alarm notifications could endanger the field systems
ATC20	Field system properties data access	1	7	9	3	If data access is not secured (especially write access), an underprivileged user could commit erroneous property changes
ATC21	Field system properties updated	1	9	11	3	If properties are not updated, the facility manager will have a wrong picture of the field system status, leading to incorrect actions
Total		63	117	243	40	

$$RC_r^F = \sum_{l=1}^{N^F} \sum_{m=1}^{N^F} F_{l,r} H_{m,r-1}^F D_{m,l}^F C_m^F PR_{r-1}^F \quad \forall r \in \{2, \dots, N\} \quad (7)$$

$$RC_r^{AE-F} = \sum_{j=1}^{N^{AE}} \sum_{l=1}^{N^F} AE_{j,r} H_{l,r-1}^F D_{l,j}^{AE-F} C_l^F PR_{r-1}^{AE-F} \quad \forall r \in \{2, \dots, N\} \quad (8)$$

$$PR_r^{AE} = \left(\sum_{j=1}^{N^{AE}} \sum_{k=1}^{N^{AE}} H_{j,r}^{AE} H_{k,r}^{AE} M_{j,k}^{AE} \right) / \left(\sum_{j=1}^{N^{AE}} H_{j,r}^{AE} \right)^2 \quad \forall r \in \{1, 2, \dots, N\} \quad (9)$$

$$PR_r^F = \left(\sum_{l=1}^{N^F} \sum_{m=1}^{N^F} H_{l,r}^F H_{m,r}^F M_{l,m}^F \right) / \left(\sum_{l=1}^{N^F} H_{l,r}^F \right)^2 \quad \forall r \in \{1, 2, \dots, N\} \quad (10)$$

$$PR_r^{AE-F} = \frac{\left(\sum_{j=1}^{N^{AE}} \sum_{l=1}^{N^F} H_{j,r}^{AE} H_{l,r}^F M_{j,l}^{AE-F} \right)}{\left(\sum_{j=1}^{N^{AE}} H_{j,r}^{AE} \right) \left(\sum_{l=1}^{N^F} H_{l,r}^F \right)} \quad \forall r \in \{1, 2, \dots, N\}. \quad (11)$$

To clarify how these constraints work in pairs, consider (6) that calculates rework cost of type I for release r by summing any rework cost incurred as a result of implementing each new architectural element k in the release (i.e., $AE_{k,r} = 1$).

For each preexisting architectural element j ($H_{j,r-1}^{AE} = 1$), the rework cost is calculated by multiplying three terms: (a) number of direct and indirect dependencies that element j has on k ($D_{j,k}^{AE}$) given by element (k, j) of the DSM_{AE} in Fig. 4a; (b) implementation cost of the preexisting element j (C_j^{AE}); and, (c) propagation ratio of DSM_{AE} in release $r - 1$ (PR_{r-1}^{AE}). Equation (9) calculates the propagation ratio associated with DSM_{AE} in release r as described in Section 2.3, where the numerator counts the number of architectural element pairs (j, k) with direct or indirect dependencies ($M_{j,k}^{AE} = 1$) that are both present in release r (i.e., $H_{j,r}^{AE} = H_{k,r}^{AE} = 1$), and the

		Architecture Elements													
\$root		1	2	3	4	5	6	7	8	9	10	11	12	13	14
Alarm Notification	1	.													
L&R Engine	2	.													
Alarm Engine	3	1	.	1											
Alarm Manager	4	1	1	.											
Logon	5														
Client Updates	6														
Rule Processor	7	1	1												
Adapter Manager	8														
User Sessions Manager	9			1	1	1	1								
Data Access	10		1	1	1	1	1	1	1	.	1				
Cache	11									1	.				
FSS Adapter	12								1		.				
Data Persistence	13		1	1						1		.			
Publish Subscribe	14	1	1					1	1	1	1			.	

(a)

		Features															
\$root		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
US 01	1	.							1								
US 02	2	1	.						1								
US 03	3		.						1								
US 04	4		1	.					1								
US 05	5		1		.				1								
US 06	6		1			.			1								
US 07	7		1			1	.		1								
US 08	8						.										
ATC 14	9								1	.							
ATC 15	10									1	.						
ATC 16	11										.						
ATC 17	12										1	.					
ATC 18	13											1	.				
ATC 19	14											1	.				
ATC 20	15		1	1													
ATC 21	16	1														.	

(b)

		Features															
\$root		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Alarm Notification	1								1								
L&R Engine	2								1	1							
Alarm Engine	3		1	1					1								
Alarm Manager	4		1	1						1							
Logon	5																
Client Updates	6																
Rule Processor	7																
Adapter Manager	8																
User Sessions Manager	9		1	1	1	1											
Data Access	10																
Cache	11																
FSS Adapter	12																
Data Persistence	13								1								
Publish Subscribe	14	1	1						1	1	1						

(c)

Fig. 4. (a) MSLite architecture DSM (DSM_{AE}) based on the architectural view in Fig. 3, (b) features DSM (DSM_F) showing interdependencies among MSLite features, and (c) DMM of features and architectural elements.

denominator is the square of the number of architectural elements implemented up until and including release r . Note that $M_{j,k}^{AE} = 1$, if $D_{j,k}^{AE} \geq 1$, and 0, otherwise. Type II and III rework costs are calculated in a similar fashion by their respective constraint pair. There are two important things to note:

- a) A high propagation ratio does not necessarily mean a high rework cost (or vice versa) as Equations (6), (7), (8) also consider the level of dependency between the components involved, which is captured by parameters $D_{j,k}^{AE}$, $D_{m,l}^F$, and $D_{l,j}^{AE-F}$.
- b) The cost values in the optimization should not be interpreted as absolute dollar values, but rather as *relative* measures that enable comparison of different development paths.

Equations (12) and (13) determine the presence of architectural elements and features in each release r . In (12), if architectural element j is implemented prior to or in release r , $H_{j,r}^{AE}$ will be set to 1. Otherwise, $H_{j,r}^{AE}$ will be zero, indicating the absence of element j in release r . Equation (13) uses the same logic for presence of feature l in release r . Constraint (14) ensures that the total amount of effort needed for implementing architectural elements and features in release r does not exceed the total resources available for that release (denoted by R_r^{Total}). Constraints (15) and (16) ensure that each architectural element and feature is implemented in exactly one and only one release. This is for initial implementation and excludes any subsequent rework.

$$H_{j,r}^{AE} = \sum_{n=1}^r AE_{j,n} \quad \forall r \in \{1, 2, \dots, N\}, \forall j \in \{1, 2, \dots, N^{AE}\} \quad (12)$$

$$H_{l,r}^F = \sum_{n=1}^r F_{l,n} \quad \forall r \in \{1, 2, \dots, N\}, \forall l \in \{1, 2, \dots, N^F\} \quad (13)$$

$$\sum_{j=1}^{N^{AE}} AE_{j,r} R_j^{AE} + \sum_{l=1}^{N^F} F_{l,r} R_l^F \leq R_r^{Total} \quad \forall r \in \{1, 2, \dots, N\} \quad (14)$$

$$\sum_{r=1}^N AE_{j,r} = 1 \quad \forall j \in \{1, 2, \dots, N^{AE}\} \quad (15)$$

$$\sum_{r=1}^N F_{l,r} = 1 \quad \forall l \in \{1, 2, \dots, N^F\}. \quad (16) \quad (16)$$

Note that an empty intermediate release will not result in any cost savings. Therefore, it is possible that the model finds two (near) optimal development paths with the exact same objective function value (total cost), where one path includes empty intermediate releases and the other does not. To the model, both paths perform equally well, hence it would arbitrarily choose one of them. Constraints (17) and (18) work hand-in-hand to ensure every intermediate release is used for implementing at least one architectural element or feature to prevent the model from creating empty intermediate releases. Such cases may occur when the amount of resources (i.e., the R_r^{Total} values specified by the user) exceed the amount required for completing the project.

$$\varphi_r = (N^{AE} + N^F) - \left(\sum_{j=1}^{N^{AE}} H_{j,r}^{AE} + \sum_{l=1}^{N^F} H_{l,r}^F \right) \quad \forall r \in \{1, 2, \dots, N\} \quad (17)$$

$$\sum_{j=1}^{N^{AE}} AE_{j,r} + \sum_{l=1}^{N^F} F_{l,r} \geq \frac{\varphi_{r-1}}{BigM + \varphi_{r-1}} \quad \forall r \in \{2, \dots, N\}. \quad (18)$$

TABLE 3
Notations Used in the Optimization Model

Notation	Description
<i>Sets and Indices</i>	
N	Number of releases
N^{AE}	Number of architectural elements
N^F	Number of features
r, n	Index for release number, $r, n \in \{1, 2, \dots, N\}$
j, k	Index for architectural elements, $j, k \in \{1, 2, \dots, N^{AE}\}$
l, m	Index for features, $l, m \in \{1, 2, \dots, N^F\}$
<i>Parameters calculated based on the DSMs and DMM</i>	
$D_{j,k}^{AE}$	Number of direct and indirect dependencies that architectural element j has on k
$D_{l,m}^F$	Number of direct and indirect dependencies that feature l has on m
$D_{l,j}^{AE-F}$	Number of direct and indirect dependencies that feature l has on architectural element j
$M_{j,k}^{AE}$	Binary parameter set to 1 if architectural element j depends on k directly or indirectly
$M_{l,m}^F$	Binary parameter set to 1 if feature l depends on m directly or indirectly
$M_{l,j}^{AE-F}$	Binary parameter set to 1 if feature l depends on architectural element j directly or indirectly
<i>Parameters specified by the decision-maker</i>	
C_j^{AE}	Cost of implementing architectural element j
C_l^F	Cost of implementing feature l
V_l^F	Value delivered to the end user by feature l
R_j^{AE}	Resources (e.g., time, personnel) needed to implement architectural element j
R_l^F	Resources needed to implement feature l
R_r^{Total}	Total resources available for release r
β	Discount factor used to consider time-value of functionality delivery ($0 < \beta < 1$)
w_v	Weight assigned to early value delivery used for computing the integrated measure of cost and value ($0 \leq w_v \leq 1$ and $1 - w_v$ will be the weight assigned to total cost)
TC_{UB}^{Final}	Upper bound for total cost of the project
LB_1^{AE}	Minimum required number of architectural elements in release 1
$BigM$	A large positive Real number ($\geq 10^{14}$)
<i>Primary and Auxiliary Decision Variables (calculated and optimized by the model)</i>	
$AE_{j,r}$	Binary decision variable set to 1 if architectural element j is implemented in release r
$F_{l,r}$	Binary decision variable set to 1 if feature l is implemented in release r
$H_{j,r}^{AE}$	Binary decision variable set to 1 if architectural element j exists in release r (implemented in or before release r)
$H_{l,r}^F$	Binary decision variable set to 1 if feature l exists in release r (implemented in or before release r)
TC_r	Total cost for release r
IC_r	Implementation cost for release r
RC_r^{AE}	Rework cost for release r due to violating the dependencies between architectural elements
RC_r^F	Rework cost for release r due to violating the dependencies between features
RC_r^{AE-F}	Rework cost for release r due to violating the dependencies between features and architectural elements
PR_r^{AE}	Propagation ratio for DSM of architectural elements in release r
PR_r^{AE}	Propagation ratio for DSM of features in release r
PR_r^{AE}	Propagation ratio for DMM of features and architectural elements in release r
TV_r	Total cumulative value delivered to end user in release r
φ_r	Number of components (architectural elements and features) remaining to be implemented after release r
<i>Objective Function Value (calculated and optimized by the model)</i>	
TC^{Final}	Total cost of the complete project (minimized in Path 1)
TV_β	Total "discounted" cumulative value delivered to the end user (maximized in Path 2)
$G(TC^{Final}, TV_\beta)$	Integrated measure of cost and early value delivery (optimized in Path 3)

Equation (17) calculates the number of remaining components just after release r , denoted by φ_r . This is done by subtracting the total number of components in release r ($\sum_{j=1}^{N^{AE}} H_{j,r}^{AE} + \sum_{l=1}^{N^F} H_{l,r}^F$) from the total number components in the project ($N^{AE} + N^F$). Then, (18) ensures that the number of components implemented in release r (calculated by $\sum_{j=1}^{N^{AE}} AE_{j,r} + \sum_{l=1}^{N^F} F_{l,r}$) is greater than or equal to a fraction. If the project is already completed by the previous release ($r-1$), then φ_{r-1} and the right-hand side of (18) will be zero, allowing the model to leave release r empty. However, if the project is not completed by the previous release $r-1$, then $\varphi_{r-1} > 0$ and the right-hand side will be a small positive value less than 1, forcing the model to implement at least one component in release r .

Constraint (19) specifies the binary decision variables. Constraint (20) ensures all rework costs and propagation ratios are nonnegative. Finally, constraint (21) specifies two things: (i) a value of zero for all three rework costs in the

first release ($r = 1$). This is needed because constraints (6 – 8) do not cover $r = 1$; and, (ii) at least one component must be implemented in the first release. This is needed because constraint (18) does not cover the first release, so we need to specify this requirement separately for $r = 1$.

$$AE_{j,r}, F_{l,r} \in \{0, 1\} \quad \forall j \in \{1, 2, \dots, N^{AE}\}, \forall l \in \{1, 2, \dots, N^F\}, \\ \forall r \in \{1, 2, \dots, N\} \quad (19)$$

$$PR_r^{AE}, PR_r^F, PR_r^{AE-F}, RC_r^{AE}, RC_r^F, RC_r^{AE-F} \geq 0 \quad \forall r \\ \in \{1, 2, \dots, N\} \quad (20)$$

$$RC_1^{AE} = RC_1^F = RC_1^{AE-F} = 0 \text{ and } 0 \leq \varphi_1 < N^{AE} + N^F. \quad (21)$$

4.2 Path 2: Maximizing Early Delivery of Value to the End User

The optimization model for maximizing early value delivery can be expressed as follows:

$$\text{Maximize } TV_\beta = \sum_{r=1}^N \beta^{r-1} TV_r$$

Subject to

$$\text{Constraints (4 - 21)} \quad (22)$$

$$TV_r = \sum_{l=1}^{N^F} H_{l,r}^F V_l^F \quad \forall r \in \{1, 2, \dots, N\} \quad (23)$$

$$TC^{Final} = \sum_{r=1}^N TC_r \quad (24)$$

$$TC^{Final} \leq TC_{UB}^{Final} \quad (25)$$

$$\sum_{j=1}^{N^{AE}} H_{j,1}^{AE} \geq LB_1^{AE}. \quad (26)$$

The objective (22) is to maximize the discounted total value of the development path. The cumulative value for release r (TV_r) is computed by adding the value for all features present in that release. For example, in the case of MSLite, if the features in release 2 include US01, US02, and US03 (some of which may have been implemented in release 1), then the cumulative value for release 2 will be $TV_2 = 27 + 25 + 23 = 75$ (see Table 2 for features' value). The release cumulative values will then be discounted and summed to obtain the total discounted cumulative value, TV_β . We use a discount factor β ($0 < \beta < 1$) to account for time-value of feature delivery in release r by multiplying TV_r by β^{r-1} . This approach for computing a discounted total value essentially means that 1 unit of value delivered in release r is worth more than 1 unit of value delivered in release $r+1$. This is analogous to the calculation of the net present value (NPV) of a series of cash flows in the field of engineering economy (see Chapter 5 in [48]). Therefore, the model for Path 2 accounts for the cost of delay or opportunity cost as a result of assigning features to later releases. In the above example, suppose US01 is implemented in release 1, and US02 and US03 in release 2. Then, given $\beta = 0.8$, the discounted cumulative value up to the second release will be: (27)(0.8)⁰ + (75)(0.8)¹ = 87.

Equation (23) gives the cumulative value for each release r by adding the value of individual features present in the release – these are features for which $H_{l,r}^F = 1$ as determined by constraint (13). Constraint (24) calculates the final cost of the project, while (25) specifies an upper bound for the final cost. Since cost does not enter directly into the objective function, without constraint (25) the model would not make any effort to avoid unnecessary excessive rework costs. With this constraint, we maximize early value delivery while keeping the total cost below a user-defined threshold TC_{UB}^{Final} .

Constraint (26) enforces a user-defined lower bound for the number of architectural elements to be implemented in the first release. Without this constraint, the model would try to implement as many features as possible in the first release. While this may be acceptable under extreme agility, in most medium to large-scale development projects, such

an extreme approach may not be appropriate. Based on the complexity of the situation and desired agility, the user can set this lower bound to any number greater than or equal to zero. Constraints (4 – 21) from the model for Path 1 also apply to this model. In particular, Constraint (21) plays an additional role in this model. Without this constraint, the model might assign arbitrary non-zero values to RC_1^{AE} , RC_1^F , and RC_1^{AE-F} as total cost does not enter the objective function.

4.3 Path 3: Optimizing an Integrated Measure of Cost and Early Value Delivery

In this path, we combine the objective functions in Paths 1 and 2 by using a weighted function of the two, denoted by $G(TC^{Final}, TV_\beta)$. The optimization model for maximizing this integrated measure of cost and early value delivery can be expressed as follows:

$$\text{Maximize } G(TC^{Final}, TV_\beta) = w_v TV_\beta - (1 - w_v) TC^{Final}$$

Subject to

$$\text{Constraints (4 - 21) and (23 - 26)} \quad (27)$$

$$TV_\beta = \sum_{r=1}^N \beta^{r-1} TV_r. \quad (28)$$

The objective function in (27) is a weighted function of the discounted cumulative value delivered and total cost, where w_v ($0 \leq w_v \leq 1$) is a user-specified weight assigned to the discounted cumulative value and $(1 - w_v)$ is the weight assigned to total cost (defined in equation 4), hence the two weights add up to 1. Since a higher cost is undesirable, a negative coefficient is used for TC^{Final} . Constraints (4 – 21) and (23 – 26) from the previous models also apply to this model. Equation (28) is discussed in Section 4.2. It is worth noting that this model essentially reduces to the early value maximization model in Section 4.2 when $w_v = 1$, and to the total cost minimization model in Section 4.1 when $w_v = 0$.

4.4 Comparison of the Three Optimized Paths

In this section, we apply and compare the three models for the MSLite system. We use the (relative) value and cost of the features from Table 2, the (relative) cost of the architectural elements from Table 4, and the dependencies from the DSMs and DMM in Figs. 4a, 4b, 4c as the basis for our analysis. Without loss of generality, we consider the case where the amount of resources required for implementing architectural element j is the same as its direct implementation cost (i.e., $C_j^{AE} = R_j^{AE}$).

Since the actual system was implemented in five releases, we also use five releases in our analysis ($N = 5$). We calibrate the R_r^{Total} , TC_{UB}^{Final} , and LB_1^{AE} parameters based on resource allocation and component assignments in the development path of MSLite as it was planned and how it was actually implemented (the actual and planned paths are discussed and analyzed in Section 5). As for total resources available for each release (the R_r^{Total} values), we use 20, 35, 30, 15, and 24 for release 1, 2, ..., 5, respectively. This is based on the maximum resource allocation in the planned and actual development paths discussed in Section 5. We set $TC_{UB}^{Final} = 502$, which is the maximum of the final cost calculated for the actual and planned paths, and $LB_1^{AE} = 3$ as both of these paths assign three architectural elements to

TABLE 4
Implementation Cost of Each Architectural Element

Architectural Element	Implementation Cost
Alarm Notification	4
L&R Engine	3
Alarm Engine	3
Alarm Manager	6
Logon	2
Client Updates	5
Rule Processor	8
Adapter Manager	5
User Sessions Manager	5
Data Access	9
Cache	3
FSS Adapter	6
Data Persistence	6
Publish-Subscribe	9

the first release. In the following analysis, we use a discount factor of $\beta = 0.5$ and assign equal weights to the discounted cumulative value and total cost for computing the integrated measure $G(TC_{UB}^{Final}, TV_\beta)$ in Path 3, i.e., $w_v = 0.5$.

The MINLP models are coded in the GAMS optimization software and solved using the Outer Approximation with both Equality Relaxation and Augmented Penalty (OA/ER/AP) algorithm [47], which is available as part of the DICOPT solver in GAMS [18]. The GAMS models are available in a Mendeley Data repository associated with this paper [35]. It is important to note that guaranteeing the optimality of the solution may not be possible due to the nonlinearity and relatively large size of the model (for the MSLite system, the model for Path 3 has 358 variables and 246 equations). Even the most efficient solution approaches cannot guarantee that they will find “the optimal” solution in such cases. However, OA/ER/AP is shown to be effective in finding a near-optimal solution for complex MINLP models in many other application areas such as those studied in [33] and [34]. Therefore, the solutions presented here may not be the global optimal solution, but are expected to be near-optimal.

Table 5 provides the three optimized paths and Table 6 compares their performance with respect to the three performance measures. As expected, Path 1 focuses on implementing AEs early (see releases 1 and 2) to avoid excessive future rework costs, Path 2 focuses on early feature implementation while keeping the total project cost under a threshold/limit (TC_{UB}^{Final}), and Path 3 strives to provide a “balanced” plan.

5 ADDITIONAL ANALYSIS/EXPERIMENTS

This section presents the results of two sets of experiments: (1) comparison of actual and planned paths with the optimized paths in Table 5; and, (2) the effect of the discount factor (β).

5.1 Modeling and Analysis of the Planned and Actual Paths

We compare the planned and actual development paths of MSLite with the three paths prescribed by the optimization models to see how the developers made decisions about

ordering component implementation and how well the planned and actual paths perform under the three performance measures. The planned and actual paths shared a common plan for features, however, there are some differences in implementation of architectural elements. Some architectural elements are enhanced over multiple releases. We assume that implementation cost for an architectural element occurs in the release where its first version is implemented, hence the cost of subsequent enhancements is captured through the propagation ratio and resulting rework cost. Table 7 shows the planned and actual development paths and the total amount of resources used in each release.

Table 8 summarizes the performance of the planned and actual paths and Fig. 5 compares the two paths with the three paths obtained via optimization. Note that in the analysis of the three optimized paths, we used the maximum of the final cost for the planned and actual paths to determine an appropriate upper bound for total cost in the models for paths 2 and 3, hence the choice of $TC_{UB}^{Final} = 502$. Moreover, we used the maximum of the two resource allocation values in Table 7 as the R_r^{Total} for each release, hence the choice of 20, 35, 30, 15, and 24 units of resources for releases 1, 2, ..., 5 respectively.

In Fig. 5, each path releases five increments of the product over the course of development. In Path 1 (cost minimization), there is minimal value delivered to end users early on, as the focus is on building the structural elements that provide the foundation for those quality requirements that cut across the entire system. These elements include *Publish-Subscribe Bus* for performance and modifiability in general, *Data Persistence* for keeping the state of the system and user preferences, *FSS Adapter* for modifiability, *Data Access* for security, and *Cache* for performance. Once the architecture is in place, the model settles into a rhythm of releasing valued features.

Path 2 (early value maximization) shows high value initially, providing more than 50 percent of the total value to the user by release 2. However, the delivery of value tapers off as subsequent releases require additional rework to deal with the growing complexity of dependencies. The high rework costs in this path are incurred as several components needed to be reworked, for example:

- Alarm Engine: reworked when Alarm Manager and Rule Processor are implemented.
- Alarm Notification: reworked when Alarm Engine and Alarm Manager are implemented.
- Client Update: reworked when User Sessions Manager is added.
- L&R Engine: reworked due to Data Access and Persistence, Publish-Subscribe Bus, Rule Processor implementation.
- Logon: reworked when Data Access and User Sessions Manager are added.

In agile projects, this type of analysis can raise awareness of the cost associated with deciding on an implementation path focused solely on delivering value. If it is not acceptable to the customer, then the team and the customer can decide together which value versus cost trade-offs they are willing to accept. Consider the AEs implemented in Release 3, *Data Access* and *Cache*. Based on Table 6, their implementation

TABLE 5
Assignment of Features and Architectural Elements to Releases in the Three Paths

Release (r)	Path 1: Minimize TC^{Final}	Path 2: Maximize TV_β	Path 3: Maximize $G(TC^{Final}, TV_\beta)$
1	<p>Architectural element: FSS Adapter Publish Subscribe Bus</p> <p>Feature: -</p>	<p>Architectural element: Alarm Notification L&R Engine Logon</p> <p>Feature: US02: Change field system properties US04: Alarm notification & acknowledgment US05: Ignore alarm notification US06: Add logic condition and reaction US07: Alarms and logic/reaction pairs persistence ATC14: Connect similar field system</p>	<p>Architectural element: Logon Cache Data Persistence Publish Subscribe Bus</p> <p>Feature: -</p>
2	<p>Architectural element: Logon Adapter Manager User Sessions Manager Data Access Cache Data Persistence</p> <p>Feature: ATC16: Connect field system providing new functionality</p>	<p>Architectural element: Client Updates FSS Adapter</p> <p>Feature: ATC15: Connect field system using different format and interfaces ATC16: Connect field system providing new functionality ATC17: Field system properties update speed ATC19: No loss of alarm notifications ATC20: Field system properties data access ATC21: Field system properties updated</p>	<p>Architectural element: Client Updates User Sessions Manager Data Access</p> <p>Feature: US05: Ignore alarm notification US07: Alarms and logic/reaction pairs persistence ATC14: Connect similar field system ATC16: Connect field system providing new functionality ATC20: Field system properties data access ATC21: Field system properties updated</p>
3	<p>Architectural element: Alarm Engine Alarm Manager Client Updates Rule Processor</p> <p>Feature: US05: Ignore alarm notification ATC14: Connect similar field system ATC20: Field system properties data access</p>	<p>Architectural element: Alarm Engine Data Access Cache Data Persistence Publish Subscribe Bus</p> <p>Feature: -</p>	<p>Architectural element: L&R Engine Alarm Engine Alarm Manager Rule Processor</p> <p>Feature: US01: Visualize field system properties US02: Change field system properties US04: Alarm notification & acknowledgment US06: Add logic condition and reaction ATC17: Field system properties update speed</p>
4	<p>Architectural element: Alarm Notification</p> <p>Feature: ATC17: Field system properties update speed ATC19: No loss of alarm notifications ATC21: Field system properties updated</p>	<p>Architectural element: User Sessions Manager</p> <p>Feature: US01: Visualize field system properties ATC18: Alarm notification speed</p>	<p>Architectural element: Alarm Notification</p> <p>Feature: US03: Add alarm condition US08: Secure access to system ATC18: Alarm notification speed ATC19: No loss of alarm notifications</p>
5	<p>Architectural element: L&R Engine</p> <p>Feature: US01: Visualize field system properties US02: Change field system properties US03: Add alarm condition US04: Alarm notification & acknowledgment US06: Add logic condition and reaction US07: Alarms and logic/reaction pairs persistence US08: Secure access to system ATC15: Connect field system using different format and interfaces ATC18: Alarm notification speed</p>	<p>Architectural element: Alarm Manager Rule Processor Adapter Manager</p> <p>Feature: US03: Add alarm condition US08: Secure access to system</p>	<p>Architectural element: Adapter Manager FSS Adapter</p> <p>Feature: ATC15: Connect field system using different format and interfaces</p>

incurs a significant rework cost (257.3). Both elements support quality requirements of medium importance: performance and security. If the customer is dissatisfied with the increased overall cost compared to Optimized Path 1 (cost minimization), we can envisage multiple scenarios for reducing costs:

- Given that *Data Access* has a high cost and that *Logon* already provides basic security, the customer decides to drop the implementation of *Data Access*.
- Assuming acceptance of some rework cost and that the performance requirement is (partially) met by *Publish-Subscribe Bus* and *Client Updates*, the customer decides to drop the implementation of *Cache*.
- Given that the customer is not willing to drop any value but still wants to reduce cost, the *Data Access*

implementation is moved to an earlier release. Although there is an extra cost incurred because *Data Access* depends on *Cache*, the reduced cost attributed to avoiding rework of other elements (*Logon* and *L&R Engine*) that depend on *Data Access* would compensate.

While requirements are readily accessible in a product backlog in agile projects, additional effort would be needed to make explicit the dependencies among these requirements and the architectural elements they realize.

Fig. 5 shows that Path 3 (integrated cost and value) combines emphasis on early delivery of high-value features, and the architecture to manage dependencies and make delivery more consistent over time.

The MSLite project consisted of 7 teams across 4 continents and its development strategy focused on establishing

TABLE 6
Comparison of the Cost and Value Delivery of the Three Development Paths Prescribed by the Optimization Models. RC_r^{Total} Represents the Sum of all Three Types of Rework Cost for Release r , that is, $RC_r^{Total} = RC_r^{AE} + RC_r^F + RC_r^{AE-F}$

Release # (r)	Path 1: Minimize TC_r^{Final}				Path 2: Maximize TV_β				Path 3: Maximize $G(TC_r^{Final}, TV_\beta)$			
	IC_r	RC_r^{Total}	TC_r	TV_r	IC_r	RC_r^{Total}	TC_r	TV_r	IC_r	RC_r^{Total}	TC_r	TV_r
1	15	0	15	0	20	0	20	112	20	0	20	0
2	34	0	34	13	32	2.4	34.4	174	35	4.5	39.5	80
3	29	0	29	52	30	257.3	287.3	174	30	8.4	38.4	181
4	14	0	14	83	10	56.7	66.7	211	14	4	18	234
5	22	0	22	243	22	69.6	91.6	243	15	12.4	27.4	243
TC_r^{Final}	114.0*				500.0				143.3			
TV_β	45.1				284.1*				129.7			
$G(TC_r^{Final}, TV_\beta)$	-34.5				-108				-6.8*			

TABLE 7
Assignment of Features and Architectural Elements to Releases in the Planned and Actual Paths for Mselite

Release # (r)	Planned Path	Actual Path	Total Resources Used	
			Planned	Actual
1	<i>Architectural element:</i> Adapter Manager Data Access FSS Adapter <i>Feature:</i> -	<i>Architectural element:</i> Adapter Manager Data Access FSS Adapter <i>Feature:</i> -	20	20
2	<i>Architectural element:</i> Cache Client Updates User Sessions Manager Publish Subscribe Bus <i>Feature:</i> US01: Visualize field system properties ATC17: Field system properties update speed	<i>Architectural element:</i> Cache Client Updates User Sessions Manager Publish Subscribe Bus Rule Processor <i>Feature:</i> US01: Visualize field system properties ATC17: Field system properties update speed	27	35
3	<i>Architectural element:</i> Alarm Notification L&R Engine Alarm Engine Logon Rule Processor <i>Feature:</i> US02: Change field system properties US03: Add alarm condition US06: Add logic condition and reaction US08: Secure access to system ATC20: Field system properties data access	<i>Architectural element:</i> Alarm Notification L&R Engine Alarm Engine <i>Feature:</i> US02: Change field system properties US03: Add alarm condition US06: Add logic condition and reaction US08: Secure access to system ATC20: Field system properties data access	30	20
4	<i>Architectural element:</i> Alarm Manager <i>Feature:</i> US04: Alarm notification & acknowledgment US05: Ignore alarm notification ATC18: Alarm notification speed ATC19: No loss of alarm notifications	<i>Architectural element:</i> Alarm Manager <i>Feature:</i> US04: Alarm notification & acknowledgment US05: Ignore alarm notification ATC18: Alarm notification speed ATC19: No loss of alarm notifications	15	15
5	<i>Architectural element:</i> Data Persistence <i>Feature:</i> US07: Alarms and logic/reaction pairs persistence ATC14: Connect similar field system ATC15: Connect field system using different format and interfaces ATC16: Connect field system providing new functionality ATC21: Field system properties updated	<i>Architectural element:</i> Logon Data Persistence <i>Feature:</i> US07: Alarms and logic/reaction pairs persistence ATC14: Connect similar field system ATC15: Connect field system using different format and interfaces ATC16: Connect field system providing new functionality ATC21: Field system properties updated	22	24

TABLE 8
Comparison of the Cost and Value Delivery of the Planned and Actual Development Paths

Release # (r)	Planned Path				Actual Path			
	IC_r	RC_r^{Total}	TC_r	TV_r	IC_r	RC_r^{Total}	TC_r	TV_r
1	20	0	20	0	20	0	20	0
2	27	18.3	45.3	36	35	18.3	53.3	36
3	30	26	56	121	20	14.3	34.3	121
4	15	49.2	64.2	179	15	49.9	64.9	179
5	22	284.8	306.8	243	24	304.9	328.9	243
TC_r^{Final}	492.3				501.4			
TV_β	85.8				85.8			
$G(TC_r^{Final}, TV_\beta)$	-203.3				-207.8			

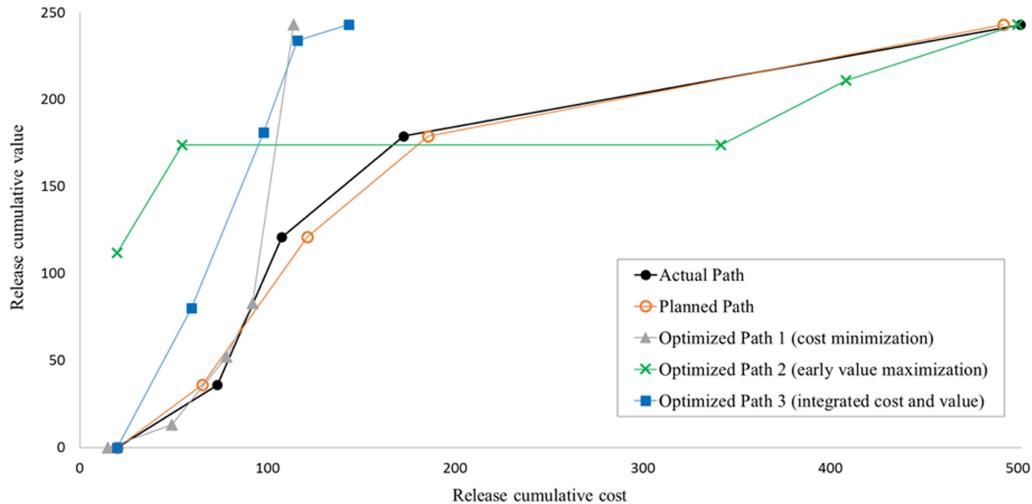


Fig. 5. Comparison of the release cumulative value and cost for the planned and actual paths and the three optimized paths. The points on each line represent the five releases for that path.

TABLE 9
Absolute Differences in Cost and Value Delivery Between the Optimized Paths and Actual Development Path

Release	Absolute difference from Actual Path (cumulative cost)			Absolute difference from Actual Path (cumulative value)		
	Path 1 (min cost) Path 2 (max value) Path 3 (balanced)			Path 1 (min cost) Path 2 (max value) Path 3 (balanced)		
1	5	0	0	0	112	0
2	24.3	18.9	13.8	23	250	44
3	29.6	234.1	9.7	92	303	104
4	80.5	235.9	56.6	188	335	159
5	387.4	1.4	358.1	188	335	159
Average	105.36	98.06	87.64	98.2	267	93.2

the architecture of the MSLite product to achieve effective collaboration and coordination among these geographically distributed teams. The Actual Path follows a similar trajectory as the Planned Path given that the implemented code generally conformed to the planned architecture. Table 9 shows the absolute difference from the Actual Path for each release in terms of cumulative cost and cumulative value. We have also computed the average difference over all five releases. Some of the reasons these paths performed relatively poorly with respect to cost when compared to Path 1 and 3 is that they focused on mitigating the risk of understanding field system technology in release 1 at the expense of extra rework, and implementation of several architectural elements related to handling variability in field systems was done over several iterations. Early implementation of a subset of architectural elements by itself, however, is not enough to minimize rework cost. One should consider, for the entire system, the appropriate order in which architectural elements and the features they support are implemented. Due to the complexity of many real-world software, however, identifying the direct and indirect dependencies among architectural elements and features is not trivial without a systematic/analytical technique such as the one proposed here. We took an overly pessimistic estimate of the extent of rework and did not do a finer grained analysis to isolate changes to sub-elements and dependencies within the affected elements. Nevertheless, the results indicate the potential to improve the planned and actual paths.

Fig. 5 and Table 9 show the initial 3 releases of the actual development path follow the “cost minimization” path closely in terms of delivering value. However, as discussed in the previous paragraph, the established architecture, while good for the features included in first three releases, was not optimal for the remaining features. Therefore, the last two releases of the actual development path (especially the last one) incurred relatively high rework cost.

5.2 The Effect of the Discount Factor (β)

One of the core principles of agile software development is early and continuous delivery of value to the customer. Putting the most essential features in front of the customer frequently allows us to not only validate our assumptions and knowledge of customer needs, but to also adapt to changing market conditions rapidly. Therefore, the time-value of the features can change depending on when they are released, and there may be a cost associated with delay or lost opportunity. In our models, the discount factor (β) is used for adjusting the value of a feature if its release is delayed.

When using the proposed analytical models for decision-making regarding the development path, it is critical to understand and select an appropriate value for this discount factor. To better understand the effect of this parameter, we vary β from 0.05 to 1.00 in 0.05 increments under Path 2 (early value maximization) and solve the model in each case. Interestingly, we find only five candidate optimal paths depending on the choice of β .

TABLE 10
The Effect of Discount Factor (β) on the Optimal Release Plan in Path 2 (Early Value Delivery Maximization)

Candidate Plan	Cumulative release value (TV_r)					Discount factor (β)																			
	r=1	r=2	r=3	r=4	r=5	0.05	0.10	0.15	0.20	0.25	0.30	0.35	0.40	0.45	0.50	0.55	0.60	0.65	0.70	0.75	0.80	0.85	0.90	0.95	1.00
A	114	165	165	165	243	122.7	132.3	143.1	155.3	169.1	184.8	202.7	223.2	246.7	273.6	304.4	339.5	379.7	425.3	477.1	535.6	601.6	675.9	759.1	852.0
B	112	174	184	184	243	121.2	131.4	143.0	156.0	170.8	187.7	207.0	229.0	254.3	283.2	316.2	353.9	396.7	445.4	500.5	562.7	632.7	711.2	799.0	897.0
C	112	174	174	211	243	121.2	131.4	142.9	155.8	170.6	187.5	206.9	229.2	254.7	284.1	317.7	356.1	399.9	449.8	506.3	570.1	642.0	722.8	813.2	914.0
D	98	172	225	234	243	107.2	117.7	129.8	143.7	159.7	178.1	199.4	224.0	252.3	284.7	321.8	364.2	412.5	467.3	529.2	598.9	677.3	765.1	863.0	972.0
E	81	174	234	243	243	90.3	101.0	113.3	127.5	143.9	162.8	184.6	209.8	238.8	272.1	310.2	353.6	403.1	459.2	522.5	593.9	674.0	763.7	863.8	975.0

The Highlighted cells denote the candidate plan with the maximum discounted cumulative value (TV_β) under the corresponding β .

Table 10 compares the release cumulative values (TV_r) and the total discounted cumulative value (TV_β) for the five candidate plans under each β value (looking down the columns). An important practical implication of these results is that, instead of finding the exact “correct” discount factor, the team only needs to determine an appropriate range for β for the situation at hand. This significantly facilitates the decision-making process and enhances the applicability of the optimization models in real-world situations. For the case of MSLite, the developer needs to decide which of these five intervals, i.e., $\beta \in [0.05, 0.15]$, $[0.20, 0.35]$, $[0.40, 0.45]$, $[0.50, 0.90]$, $[0.95, 1.00]$, appropriately reflects the importance of early value delivery for their software.

It is important to note that in Step 4 of the iterative process described in introduction, we also recommend performing a sensitivity analysis on parameter w_v , weight assigned to early value delivery. This analysis provides additional insight on effect of relative importance given to early value delivery and total cost. However, space limitations preclude inclusion of this analysis for MSLite in the current paper.

6 DISCUSSION ON THE VALIDITY OF THE PROPOSED APPROACH

In this study, we investigated how quantifying architecture quality can help developers understand the opportunities and risks associated with the desire to deliver value to the customer early or to minimize the cost associated with architectural refactoring and rework later in the development process. Refactoring cost is incurred when the need to deliver value early causes a development team to defer the design and implementation of architectural elements in order to be expedient. Significant rework may be needed when these architectural elements must be implemented in some later release. Our goal was to provide a quantifiable architecture quality model that can help in making informed decisions when a team is faced with value versus cost trade-offs. As a step toward this goal, we used DSM- and DMM-based dependency analysis with propagation ratio as a criterion for measuring architecture quality.

Our study is based on using a real system and its artifacts and, in retrospect, evaluating the extent to which the propagation ratio and our proposed rework algorithm can assist in predicting rework from iteration to iteration. The key variables that our conclusions depend on are the propagation ratio, value of the requirements, architectural elements, and associated implementation costs. We keep the value of the

requirements consistent across the different paths that we analyze to avoid biasing the results. We calculate the propagation ratio based on the actual artifacts. However, there is a possible effect from confounding instances in which changes in cost variables may rather be attributed to the existence of variations in the degree of other variables such as the competence of the developers and the order in which they implemented the elements. We minimize this impact by basing our conclusions on relative values rather than absolute dollar figures.

We base all of our analysis on actual, real artifacts rather than a hypothetical study. We simulate our results by creating alternative paths for comparative analysis. In addition, to draw consistent conclusions, we keep the resulting architecture and system value the same across all paths so that we can compare the variations occurring in each. However, our goal is to propose an analytical approach to evaluate the architecture, do trade-off analysis and monitor the accumulating rework. Our models enable us to do this and are scalable and applicable in any other real-world situation.

Overall, our study shows that architectural analysis using DSM- and DMM-based dependency analysis with propagation ratio integrated in an optimization framework, can improve project monitoring by focusing on quantifying accumulating rework and value from iteration to iteration.

7 RELATED WORK

Dependency management has been studied extensively at the level of code artifacts and in the context of system engineering [5]. Applying dependency management at the architecture level has shown promise due to increasing tool support for using DSMs for architectural analysis [20]. DMM analysis [10], [11] can augment DSM analyses and can be used to represent the dependencies between capabilities and architectural elements to further focus the goals of iterative release planning where courses of action may change as the project progresses. DSM and DMM models have been built for predicting the effects of requirements and design changes [8], [9], [19], [28], [29], [32], and [38] uses such models for requirements traceability and analyzing change propagation. Studies use DMM to map functional requirements to design parameters [42], [46], [52], and to map product components to requirements [4], [14], [31].

Metrics, such as propagation ratio [30], can be extracted from an architecture represented in the form of a DSM, and further help with this process. Metrics associated with

TABLE 11
Fowler's Technical Debt Taxonomy [14]

	Reckless	Prudent
Deliberate	We don't have time for design.	We must ship now and deal with the consequences.
Inadvertent	What's layering?	Now we know how we should have done it.

dependency provide data for inferring the likely costs of change propagation, especially when dependencies between architectural elements are considered. Carriere *et. al* [7] discuss one such example, where the value of re-architecting decisions needs to be understood to determine if the expense to implement them is justified.

More recently, technical debt analysis has been used increasingly to determine the need for and cost of refactoring a system as it evolves. There is a key difference between debt that results from employing bad engineering practices and debt resulting from intentional decision-making in pursuit of a strategic goal. Fowler [15] details this distinction by describing technical debt using four quadrants, as shown in Table 11.

Technical debt resonates with maintenance and evolution challenges. Lehman [27] observes that for systems to remain useful they must change, and that change will increase their complexity, leading to software decay if refactoring is not done as needed. Parnas [36] calls this phenomenon “software aging,” reflecting the failure of a product owner to modify it to meet changing needs. Lehman’s observations about system evolution still apply in projects that follow agile software development approaches [43]. They provide insight into the inevitability of and need for re-architecting (and the high potential of debt that can accumulate), emulating the cost minimization path as our study demonstrates.

An empirical study conducted with architects at IBM concluded that the ability to assess debt does matter [23]. Yet there is a significant gap in achieving this goal. The interviews found that the following experiences were common: induced and unintentional debts are significant challenges, decisions are managed in an ad hoc manner, and stakeholders lack effective ways to communicate and reason about debt [23]. What makes this challenging are invisible aspects of software evolution and quality [13], [24].

The ability to elicit and improve the visibility of the state of the project from both a project-management perspective and a system-quality perspective is an area of practical interest since it can help provide guidance for adjusting the course of action as the system is being developed and recognizing debt as it accumulates. The concept of a technical debt item is emerging as a systematic way of capturing and bringing visibility to atomic elements of technical debt. A technical debt item connects the development artifact in the code or design with consequences for the quality, value, and cost of the system [25].

Static code analysis tools and plug-ins do provide insights into technical debt analysis and architectural rework [17], [21], [26], [37], [45]. Some of these tools do also include use of DSMs as a way to visualize architectural dependencies, but they do not provide trade-off analysis

capabilities [26], [45]. The Object Management Group codifies the best practices of source code analysis technologies which have implemented the CISQ Quality Characteristic measures to establish a standard for automating a measure of technical debt [1]. However, approaches based on code quality have their limits and do not support architecture-level analysis [6].

8 CONCLUSION AND FUTURE WORK

The main contribution of this paper lies in: (1) developing an analytical framework to quantify cost of architecture refactoring and a decision support tool for release planning; and, (2) providing managerial insights on the trade-offs between early value delivery and costs. We propose the following general iterative process for applying our framework:

- Step 1: Determine the DSM and DMM dependencies
- Step 2: Use current procedures/practice to develop a planned development path
- Step 3: Parameterize the optimization models based on the planned path (similar to what we did for R_r^{Total} , TC_{UB}^{Final} , and LB_1^{AE} parameters)
- Step 4: Solve the models for various β and w_v values
- Step 5: Make necessary adjustments to the initial planned path based on the experimental results and repeat the steps above until satisfied with the planned path.

The optimization models recommend the order in which architectural elements and features should be implemented across different releases so as to: (a) minimize rework cost; (b) maximize early value delivery; or, (c) optimize an integrated measure of cost and value. The ability to quantify the potential for future rework cost during iterative release planning is a key aspect of managing rework strategically [3]. Our approach facilitates reasoning about the economic implications and perhaps deliberately allowing rework cost in the short term to achieve some greater business goal, all the while continuing to monitor architectural dependencies and looking for the opportune time to improve. Our goal was to provide an empirical basis on which to chart and adjust course. As the project evolves, the requirements are subject to change, introducing an element of uncertainty that needs further consideration. Since MSLite is a retrospective study, this uncertainty aspect is not clear. We intend to look at uncertainty in our future research.

In planning our releases, we made assumptions concerning the acceptability of splitting the delivery of functional and quality attribute capabilities across releases. In future applications, we plan to expand the concept of “minimum marketable features” [12] to the broader construct of “minimum releasable capabilities” to ensure that interdependent functional and quality attribute requirements are released in batches that deliver acceptable end-user value.

We accounted for rework using a simple cash flow model in which cost is incurred at the time of the rework. There are economic models that include rework cost that is predicted in future releases. These models become more complex since there are more choices for when to account for the future debt. We intend to explore such models in the future.

These future research directions will also offer us further opportunities to concomitantly validate the applicability of our approach and its extensions in real-world applications.

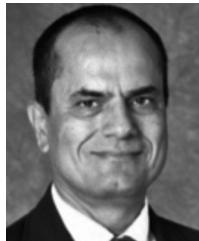
ACKNOWLEDGMENTS

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. References herein to any specific commercial product, process, or service by trade name, trade mark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by Carnegie Mellon University or its Software Engineering Institute.

REFERENCES

- [1] Object Management Group, Automated Technical Debt Measure (ATMD), Version 1, 2017, Accessed: Sep. 1, 2020. [Online]. Available: <https://www.omg.org/spec/ATDM/About-ATDM/>
- [2] B. Boehm and R. Turner, *Balancing Agility and Discipline: A Guide for the Perplexed*. Boston, MA, USA: Addison-Wesley Longman, 2003.
- [3] N. Brown *et al.*, "Managing technical debt in software-reliant systems," in *Proc. FSE/SDP Workshop Future Softw. Eng. Res.*, 2010, pp. 47–52.
- [4] N. Brown, R. Nord, I. Ozkaya, and M. Pais, "Analysis and management of architectural dependencies in iterative release planning," in *Proc. 9th Work. IEEE/IFIP Conf. Softw. Architect.*, 2011, pp. 103–112.
- [5] T. Browning, "Applying the design structure matrix to system decomposition and integration problems: A review & new directions," *IEEE Trans. Eng. Manage.*, vol. 48, no. 3, pp. 292–306, Aug. 2001.
- [6] R. P. L. Buse and T. Zimmermann, "Analytics for software development," in *Proc. FSE/SDP Workshop Future Soft. Eng. Res.*, 2010, pp. 77–80.
- [7] J. Carriere, R. Kazman, and I. Ozkaya, "A cost-benefit framework for making architectural decisions in a business context," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, vol. 2, pp. 149–157.
- [8] Y. Chen, P. Cheng, and J. Yin, "Change propagation analysis of trustworthy requirements based on dependency relations," in *Proc. 2nd IEEE Int. Conf. Inf. Manage. Eng.*, 2010, pp. 246–251.
- [9] D. K. H. Chua and M. A. Hossain, "Predicting change propagation and impact on design schedule due to external changes," *IEEE Trans. Eng. Manage.*, vol. 59, no. 3, pp. 483–493, Aug. 2012.
- [10] M. Danilovic and T. Browning, "Managing complex product development projects with design structure matrices and domain mapping matrices," *Int. J. Proj. Manage.*, vol. 25, no. 3, pp. 300–314, 2007.
- [11] M. Danilovic and B. Sandkull, "The use of dependence structure matrix and domain mapping matrix in managing uncertainty in multiple project situations," *Int. J. Proj. Manage.*, vol. 23 no. 3, pp. 193–203, 2005.
- [12] M. Denne and J. Cleland-Huang, "Software by numbers: Low-risk," in *High-Return Development*. Upper Saddle River, NJ, USA: Prentice Hall, 2003.
- [13] M. Denne and J. Cleland-Huang, "The incremental funding method: Data-driven software development," *IEEE Softw.*, vol. 21, no. 3, pp. 39–47, May/Jun. 2004.
- [14] Q. Dong, 2002, "Predicting and managing system interactions at early phase of the product development process," Ph.D. dissertation, Dept. Mech. Eng., Massachusetts Inst. Technol., Cambridge, MA, USA.
- [15] M. Fowler Technical Debt Quadrant, 2009. Accessed: Oct. 22, 2019. [Online]. Available: <http://www.martinfowler.com/bliki/TechnicalDebtQuadrant.html>
- [16] D. Garlan *et al.*, *Documenting Software Architectures: Views and Beyond* 2nd ed., Boca Raton, FL, USA: AW Professional, 2010.
- [17] O. Gaudin, "Evaluate your technical debt with sonar," 2009. [Online]. Available: <http://www.sonarsource.org/evaluate-your-technical-debt-with-sonar>
- [18] I. E. Grossmann and J. Viswanathan *DICOPT User's Manual*, Washington, DC, USA: GAMS Development Corp., 2008.
- [19] B. Hamraz, N. H. M. Caldwell, D. C. Wynn, and P. J. Clarkson, "Requirements-based development of an improved engineering change management method," *J. Eng. Des.*, vol. 24, pp. 765–793, 2013.
- [20] C. Hinsman, N. Sangal, and J. Stafford, "Achieving agility through architecture visibility," in *Proc. 5th Int. Conf. Qual. Softw. Architectures: Architectures Adaptive Softw. Syst.*, 2009, pp. 116–129.
- [21] R. Kazman *et al.*, "A case study in locating the architectural roots of technical debt," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 179–188.
- [22] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proc. Int. Conf. Softw. Eng.*, 2009, pp. 309–319.
- [23] T. Klinger, P. Tarr, P. Wagstrom, and C. Williams, "An enterprise perspective on technical debt," in *Proc. 2nd Workshop Manag. Techn. Debt*, 2011, pp. 35–38.
- [24] P. Krutchen, R. Nord, and I. Ozkaya, "Technical debt: From metaphor to theory and practice," *IEEE Softw.*, vol. 29 no. 6, pp. 18–21, Nov./Dec. 2012.
- [25] P. Krutchen, R. Nord, and I. Ozkaya *Managing Technical Debt: Reducing Friction in Software Development*, Reading, MA, USA: Addison-Wesley, 2019.
- [26] Lattix. Version 11.5, 2019. Accessed: Oct. 22, 2019. [Online]. Available: <http://www.lattix.com>
- [27] M. M. Lehman, *Program Evolution: Processes of Software Change*. San Diego, CA, USA: Academic Professional, 1985.
- [28] W. T. Lee, W.-Y. Deng, J. Lee, and S.-J. Lee, "Change impact analysis with a goal-driven traceability-based approach," *Int. J. Intell. Syst.*, vol. 25, pp. 878–908, 2010.
- [29] W. Li and Y. B. Moon, "Modeling and managing engineering changes in a complex product development process," in *Proc. Winter Simul. Conf.*, 2011, pp. 792–804.
- [30] A. MacCormack, J. Rusnak, and C. Baldwin, *Exploring the Duality Between Product and Organizational Architectures: A Test of the Mirroring Hypothesis (Version 3.0)*, Cambridge, MA, USA: Harvard Business School, 2008.
- [31] J. R. A. Maier and G. M. Fadel, "Affordance-based design methods for innovative design, redesign and reverse engineering," *Res. Eng. Des.*, vol. 20, pp. 225–239, 2009.
- [32] B. Morkos, P. Shankar, and J. D. Summers, "Predicting requirement change propagation, using higher order design structure matrices: An industry case study," *J. Eng. Des.*, vol. 23, pp. 902–923, 2012.
- [33] A. Negahban, "Optimizing consistency improvement of positive reciprocal matrices with implications for Monte Carlo analytic hierarchy process," *Comput. Ind. Eng.*, vol. 124, pp. 113–124, 2018.
- [34] A. Negahban and M. Dehghanimohammabadi, "Optimizing the supply chain configuration and production-sales policies for new products over multiple planning horizons," *Int. J. Prod. Econ.*, vol. 196, pp. 150–162, 2018.
- [35] A. Negahban, R. S. Sangwan, R. L. Nord, and I. Ozkaya, "Models for Optimization of software release planning considering architectural dependencies amid competing interests in value and cost," Mendeley Data, [Online]. Available: <https://dx.doi.org/doi:10.17632/s5xxjrcvzd.2>, doi: 10.17632/s5xxjrcvzd.2.
- [36] D. L. Parnas, "Software aging," in *Proc. 16th Int. Conf. Softw. Eng.*, 1994, pp. 279–287.
- [37] S. Penchikala, "Architecture analysis tool SonarJ 6.0 supports structural debt index and quality model," 2010. Accessed: Oct. 22, 2019. [Online]. Available: <http://www.infoq.com/news/2010/08/sonarj-6.0>
- [38] A. Salado and R. Nilchiani, "Assessing the impacts of uncertainty propagation to system requirements by evaluating requirement connectivity," in *Proc. 23rd Int. Symp. INCOSE*, 2013, pp. 647–661.
- [39] R. Sangwan, M. Bass, N. Mullick, D. J. Paulish, and J. Kazmeier *Global Software Development Handbook*. New York, NY, USA: Auerbach Publications, 2006.
- [40] R. S. Sangwan and C. J. Neill, "Characterizing essential and incidental complexity in software architectures," in *Proc. Joint 8th Work. IEEE/IFIP Conf. Softw. Architecture, 3rd Eur. Conf. Softw. Architecture*, 2009, pp. 265–268.
- [41] R. S. Sangwan, C. Neill, M. Bass, and Z. El Houda, "Integrating a software architecture-centric method into object-oriented analysis and design," *J. Syst. Softw.*, vol. 81, pp. 727–746, 2008.
- [42] S. Sarkar, A. Dong, and J. S. Gero, "Learning symbolic formulations in design: Syntax, semantics, and knowledge reification," *Artif. Intell. Eng. Des. Anal. Manuf.*, vol. 24, pp. 63–85, 2010.

- [43] R. Sindhgatta, N.C. Narendra, and B. Sengupta, "Software evolution in agile development: A case study," in *Proc. ACM Int. Conf. Companion Object Oriented Program. Syst. Lang. Appl. Companion*, 2010, pp. 105–114.
- [44] D. V. Steward, "The design structure system: A method for managing the design of complex systems," *IEEE Trans. Eng. Manage.*, vol. EM-28, no. 3, pp. 71–74, Aug. 1981.
- [45] Structure 101, Accessed: Oct. 22, 2019. [Online]. Available: <http://structure101.com>
- [46] D. Tang, R. Zhu, S. Dai, and G. Zhang, "Enhancing axiomatic design with design structure matrix," *Concurrent Eng. Res. Appl.*, vol. 17, pp. 129–137, 2009.
- [47] J. Viswanathan and I. E. Grossmann, "A combined penalty function and outer-approximation method for MINLP optimization," *Comput. Chem. Eng.*, vol. 14 no. 7, pp. 769–782, 1990.
- [48] J. A. White, K. E. Case, and D. B. Pratt, *Principles of Engineering Economic Analysis*. 6th ed., New York, NY, USA: Wiley, 2012.
- [49] K. Wiegers, "First things first: Prioritizing requirements," *Softw. Develop.*, vol. 7 no. 9, pp. 48–53, 1999.
- [50] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering: An Introduction*. 2nd ed. New York, NY, USA: Springer, 2000.
- [51] R. K. Yin, *Case Study Research: Design and Methods*, 3rd ed. Newbury Park, CA, USA: Sage, 2002.
- [52] M. Yoshimura, K. Izui, and Y. Fujimi, "Optimizing the decision-making process for large-scale design problems according to criteria interrelationships," *Int. J. Prod. Res.*, vol. 41, pp. 1987–2002, 2003.



Raghvinder S. Sangwan is an associate professor of software engineering with the School of Graduate Professional Studies, Pennsylvania State University, Malvern, PA, USA.



Ashkan Negahban is an assistant professor of engineering management with the School of Graduate Professional Studies, Pennsylvania State University, Malvern, PA, USA.



Robert L. Nord is a senior member of the technical staff at the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA.



Ipek Ozkaya is technical director with the Engineering Intelligent Software Systems at the Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, USA.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

CODIT: Code Editing With Tree-Based Neural Models

Saikat Chakraborty^{ID}, Yangruibo Ding^{ID}, Miltiadis Allamanis^{ID}, and Baishakhi Ray^{ID}

Abstract—The way developers edit day-to-day code tends to be repetitive, often using existing code elements. Many researchers have tried to automate repetitive code changes by learning from specific change templates which are applied to limited scope. The advancement of deep neural networks and the availability of vast open-source evolutionary data opens up the possibility of automatically learning those templates from the wild. However, deep neural network based modeling for code changes and code in general introduces some specific problems that need specific attention from research community. For instance, compared to natural language, source code vocabulary can be significantly larger. Further, good changes in code do not break its syntactic structure. Thus, deploying state-of-the-art neural network models without adapting the methods to the source code domain yields sub-optimal results. To this end, we propose a novel tree-based neural network system to model source code changes and learn code change patterns from the wild. Specifically, we propose a tree-based neural machine translation model to learn the probability distribution of changes in code. We realize our model with a change suggestion engine, Codit, and train the model with more than 24k real-world changes and evaluate it on 5k patches. Our evaluation shows the effectiveness of Codit in learning and suggesting patches. Codit can also learn specific bug fix pattern from bug fixing patches and can fix 25 bugs out of 80 bugs in Defects4J.

Index Terms—Code change, tree-2-tree translation, code synthesis, neural machine translator, empirical software engineering

1 INTRODUCTION

DEVELOPERS edit source code to add new features, fix bugs, or maintain existing functionality (e.g., API updates, refactoring, etc.) all the time. Recent research has shown that these edits are often repetitive [1], [2], [3]. Moreover, the code components (e.g., token, sub-trees, etc.) used to build the edits are often taken from the existing codebase [4], [5]. However, manually applying such repetitive edits can be tedious and error-prone [6]. Thus, it is important to automate code changes, as much as possible, to reduce the developers' burden.

There is a significant amount of industrial and academic work on automating code changes. For example, modern IDEs support specific types of automatic changes (e.g., refactoring, adding boiler-plate code [7], [8], etc.). Many research tools aim to automate some types of edits, e.g., API related changes [9], [10], [11], [12], [13], [14], refactoring [15], [16], [17], [18], frequently undergone code changes related to Pull Requests [19], etc. Researchers have also proposed to automate generic changes by learning either from example edits [20], [21] or from similar patches applied previously to source code [2], [3], [19], [22]. Automated *Code Change*¹ task

1. We use the term *Code Change* and *Code Edit* interchangeably

- Saikat Chakraborty, Yangruibo Ding, and Baishakhi Ray are with the Department of Computer Science, Columbia University, New York, NY 10027 USA. E-mail: {saikatc, raybj}@cs.columbia.edu, yangruibo.ding@columbia.edu.
- Miltiadis Allamanis is with Microsoft Research, CB1 2FB Cambridge, U.K. E-mail: miallarna@microsoft.com.

Manuscript received 3 Dec. 2019; revised 17 July 2020; accepted 25 Aug. 2020. Date of publication 31 Aug. 2020; date of current version 18 Apr. 2022. (Corresponding author: Saikat Chakraborty.)

Recommended for acceptance by M. Nagappan.

Digital Object Identifier no. 10.1109/TSE.2020.3020502

is defined as modification of existing code (i.e., adding, deleting, or replacing code elements) through applying such frequent change patterns [19], [23], [24], [25].

While the above lines of work are promising and have shown initial success, they either rely on predefined change templates or require domain-specific knowledge: both are hard to generalize to the larger context. However, all of them leverage, in somehow, common edit patterns. Given that a large amount of code and its change history is available thanks to software forges like GitHub, Bitbucket, etc., a natural question arises: *Can we learn to predict general code changes by learning them in the wild?*

Recently there has been a lot of interest in using Machine Learning (ML) techniques to model and predict source code from real world [26]. However, modeling changes is different from modeling generic code generation, since modeling changes is conditioned on the previous version of the code. In this work, we investigate whether ML models can capture the repetitiveness and statistical characteristics of code edits that developers apply daily. Such models should be able to automate code edits including feature changes, bug fixes, and other software evolution-related tasks.

A *code edit* can be represented as a tuple of two code versions: $\langle \text{prev}, \text{target} \rangle$. To model the edits, one needs to learn the conditional probability distribution of the *target* code version given its *prev* code version. A good probabilistic model will assign higher probabilities to plausible target versions and lower probabilities to less plausible ones. Encoder-decoder Neural Machine Translation models (NMT) are a promising approach to realize such code edit models, where the previous code version is encoded into a latent vector representation. Then, the target version is synthesized (decoded) from the encoded representation. Indeed some previous works [19], [27], [28] use Seq2Seq NMT models to

capture changes in token sequences. However, code edits also contain structural changes, which need a syntax-aware model.

To this end, we design a two step encoder-decoder model that models the probability distribution of changes. In the first step, it learns to suggest structural changes in code using a tree-to-tree model, suggesting structural changes in the form of Abstract Syntax Tree (AST) modifications. Tree-based models, unlike their token-based counterparts, can capture the rich structure of code and always produce syntactically correct patches. In the second step, the model concretizes the previously generated code fragment by predicting the tokens conditioned on the AST that was generated in the first step: given the type of each leaf node in the syntax tree, our model suggests concrete tokens of the correct type while respecting scope information. We combine these two models to realize CODIT, a code change suggestion engine, which accepts a code fragment and generates potential edits of that snippet.

In this work, we particularly focus on smaller changes as our previous experience [3] shows that such changes mostly go through similar edits. In fact, all the previous NMT-based code transformation works [19], [27], [28] also aim to automate such changes. A recent study by Karampatsis *et al.* [29] showed that small changes are frequent—our analysis of the top 92 projects of their dataset found that, on average, in each project, one line changes take place around 26.71 percent of the total commits and account for up to 70 percent for the bug fix changes. Our focus in this work is primarily to automatically change small code fragments (often bounded by small AST sizes and/or few lines long) to reflect such repetitive patterns. Note that, in theory, our approach can be applied to any small fragment of code in the project repository with any programming language. However, for prototyping, we designed CODIT to learn changes that belong to methods of popular java project in Github.

In this work, we collect a new dataset — *Code-Change-Data*, consisting of 32,473 patches from 48 open-source GitHub projects collected from Travis Torrent [30]. Our experiments show CODIT achieves 15.94 percent patch suggestion accuracy in the top 5 suggestions; this result outperforms a Copy-Seq2Seq baseline model by 63.34 percent and a Tree2Seq based model by 44.37 percent.

We also evaluate CODIT on *Pull-Request-Data* proposed by Tufano *et al.* [19]. Our evaluation shows that CODIT suggests 28.87 percent of correct patches in the top 5 outperforming Copy-Seq2Seq-based model by 9.26 percent and Tree2Seq based model by 22.92 percent. Further evaluation on CODIT’s ability to suggest bug-fixing patches in Defects4J shows that CODIT suggests 15 complete fixes and 10 partial fixes out of 80 bugs in Defects4J. In summary, our key contributions are:

- We propose a novel tree-based code editing machine learning model that leverages the rich syntactic structure of code and generates syntactically correct patches. To our knowledge, we are the first to model code changes with tree-based machine translation.
- We collect a large dataset of 32k real code changes. Processed version of the dataset is available at https://drive.google.com/file/d/1wS1_SN17tbATqlhNMO0O7sEkH9gqJ9Vr.

- We implement our approach, CODIT, and evaluate the viability of using CODIT for changes in a code, and patching bug fixes. Our code is available at <https://git.io/JJGwU>.

2 BACKGROUND

Modeling Code Changes. Generating source code using machine learning models has been explored in the past [31], [32], [33], [34]. These methods model a probability distribution $p(c|\kappa)$ where c is the generated code and κ is any contextual information upon which the generated code is conditioned. In this work, we generate *code edits*. Thus, we are interested in models that predict code given its previous version. We achieve this using NMT-style models, which are a special case of $p(c|\kappa)$, where c is the new and κ is the previous version of the code. NMT allows us to represent code edits with a single end-to-end model, taking into consideration the original version of a code and defining a conditional probability distribution of the target version. Similar ideas have been explored in NLP for paraphrasing [35].

Grammar-Based Modeling. Context Free Grammars (CFG) have been used to describe the syntax of programming languages [36] and natural language [37], [38]. A CFG is a tuple $G = (N, \Sigma, P, S)$ where N is a set of non-terminals, Σ is a set of terminals, P is a set of production rules in the form of $\alpha \rightarrow \beta$ and $\alpha, \beta \in (N \cup \Sigma)^*$, and S is the start symbol. A sentence (i.e., sequence of tokens) that belongs to the language defined by G can be parsed by applying the appropriate derivation rules from the start symbol S . A common technique for generation of utterances is to expand the left-most, bottom-most non-terminal until all non-terminals have been expanded. Probabilistic context-free grammar (PCFG) is an extension of CFG, where each production rule is associated with a probability, i.e., is defined as (N, Σ, P, Π, S) where Π defines a probability distribution for each production rule in P conditioned on α .

Neural Machine Translation Models. NMT models are usually a cascade of an *encoder* and a *decoder*. The most common model is sequence-to-sequence (seq2seq) [39], where the input is treated as a sequence of tokens and is encoded by a sequential encoder (e.g., biLSTM). The output of the encoder is stored in an intermediate representation. Next, the decoder using the encoder output and another sequence model, e.g., an LSTM, generates the output token sequence. In NMT, several improvements have been made over the base seq2seq model, such as *attention* [39] and *copying* [40]. Attention mechanisms allow decoders to automatically search information within the input sequence when predicting each target element. Copying, a form of an attention mechanism, allows a model to directly copy elements of the input into the target. We employ both attention and copying mechanisms in this work.

3 MOTIVATING EXAMPLE

Fig. 1 illustrates an example of our approach. Here, the original code fragment `return super.equals(object)` is edited to `return object == this`. CODIT takes these two code fragments along with their context, for training. While suggesting changes, i.e., during test time, CODIT takes as

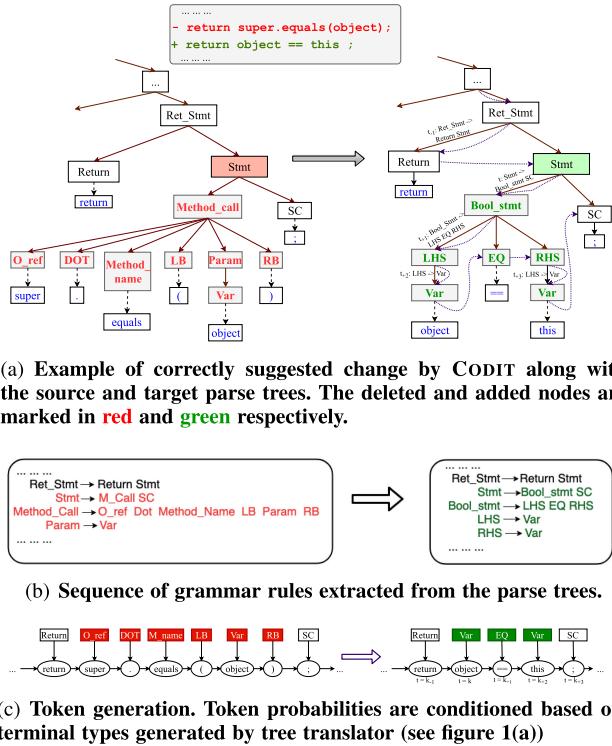


Fig. 1. Motivating example.

input the previous version of the code and generates its edited version.

CODIT operates on the parse trees of the previous (t_p) and new (t_n) versions of the code, as shown in Fig. 1a (In the rest of the paper, a subscript or superscript with p and n correspond to previous and new versions respectively). In Fig. 1, changes are applied only to the subtree rooted at the *Method_call* node. The subtree is replaced by a new subtree (t_n) with *Bool_stmt* as a root. The deleted and added subtrees are highlighted in red and green respectively.

While modeling the edit, CODIT first predicts the structural changes in the parse tree. For example, in Fig. 1a CODIT first generates the changes corresponding to the subtrees with dark nodes and red edges. Next the structure is concretized by generating the token names (terminal nodes). This is realized by combining two models: (i) a tree-based model predicting the structural change (see Section 4.1) followed by a (ii) a token generation model conditioned on the structure generated by the tree translation model (see Section 4.2).

Tree Translator. The tree translator is responsible for generating structural changes to the tree structure. A machine learning model is used to learn a (probabilistic) mapping between t_p and t_n . First, a tree encoder, encodes t_p computing a distributed vector representation for each of the production rules in t_p yielding the distributed representation for the whole tree. Then, the tree decoder uses the encoded representations of t_p to sequentially select rules from the language grammar to generate t_n . The tree generation starts with the root node. Then, at each subsequent step, the bottom-most, left-most non-terminal node of the current tree is expanded. For instance, in Fig. 1a, at time step t , node *Stmt* is expanded with rule *Stmt* \rightarrow *Bool_stmt SC*. When the tree generation process encounters a terminal node, it records the node type to be used by the token generation model and

proceeds to the next non-terminal. In this way, given the LHS rule sequences of Fig. 1b the RHS rule sequences is generated.

Token Generator. The token generator predicts concrete tokens for the terminal node types generated in the previous step. The token generator is a standard seq2seq model with attention and copying [39] but constrained on the token types generated by the tree translator. To achieve this, the token generator first encodes the token string representation and the node type sequence from t_p . The token decoder at each step probabilistically selects a token from the vocabulary or copies one from the input tokens in t_p . However, in contrast to traditional seq2seq where the generation of each token is only conditioned on the previously generated and source tokens, we additionally condition on the token type that has been predicted by the tree model and generate only tokens that are valid for that token type. Fig. 1c shows this step: given the token sequence of the original code < super . equals (object) > and their corresponding token types (given in dark box), the new token sequence that is generated is < object == this >.

4 TREE-BASED NEURAL TRANSLATION MODEL

We decompose the task of predicting code changes in two stages: First, we learn and predict the structure (syntax tree) of the edited code. Then, given the predicted tree structure, we concretize the code. We factor the generation process as

$$P(c_n|c_p) = P(c_n|t_n, c_p)P(t_n|t_p)P(t_p|c_p), \quad (1)$$

and our goal is to find \hat{c}_n such that $\hat{c}_n = \operatorname{argmax}_{c_n} P(c_n|c_p)$. Here, c_p is the previous version of the code and t_p is its parse tree, whereas c_n is the new version of the code and t_n its parse tree. Note that parsing a code fragment is unambiguous, i.e., $P(t_p|c_p) = 1$. Thus, our problem takes the form

$$\hat{c}_n = \arg \max_{c_n, t_n} \underbrace{P(c_n|t_n, c_p)}_{\mathcal{M}_{\text{token}}} \cdot \underbrace{P(t_n|t_p)}_{\mathcal{M}_{\text{tree}}}. \quad (2)$$

Equation (2) has two parts. First, it estimates the changed syntax tree $P(t_n|t_p)$. We implement this with a tree-based encoder-decoder model (Section 4.1). Next, given the predicted syntax tree t_n , we estimate the probability of the concrete edited code with $p(c_n|t_n, c_p)$ (Section 4.2).

4.1 Tree Translation Model ($\mathcal{M}_{\text{tree}}$)

The goal of $\mathcal{M}_{\text{tree}}$ is to model the probability distribution of a new tree (t_n) given a previous version of the tree (t_p). For any meaningful code the generated tree is syntactically correct. We represent the tree as a sequence of grammar rule generations following the CFG of the underlying programming language. The tree is generated by iteratively applying CFG expansions at the left-most bottom-most non-terminal node (frontier_node) starting from the start symbol.

For example, consider the tree fragments in Fig. 1a. Fig. 1b shows the sequence of rules that generate those trees. For example, in the right tree of Fig. 1a, the node *Ret_Stmt* is first expanded by the rule: *Ret_Stmt* \rightarrow *Return Stmt*. Since, *Return* is a terminal node, it is not expanded any further. Next, node *Stmt* is expanded with rule: *Stmt* \rightarrow *Bool_stmt SC*. The tree is further expanded with *Bool_stmt* \rightarrow *LHS EQ*

RHS, LHS \rightarrow Var, and RHS \rightarrow Var. During the tree generation process, we apply these rules to yield the tree fragment of the next version.

In particular, the tree is generated by picking CFG rules at each non-terminal node. Thus, our model resembles a Probabilistic Context-Free Grammar, but the probability of each rule depends on its surroundings. The neural network models the probability distribution, $P(R_k^n | R_1^n, \dots, R_{k-1}^n, t_p)$: At time k the probability of a rule depends on the input tree t_p and the rules R_1^n, \dots, R_{k-1}^n that have been applied so far. Thus, the model for generating the syntax tree t_n is given by

$$P(t_n | t_p) = \prod_{k=1}^{\tau} P(R_k^n | R_1^n, \dots, R_{k-1}^n, t_p). \quad (3)$$

Encoder. The encoder encodes the sequence of rules that construct t_p . For every rule R_i^p in t_p , we first transform it into a single learnable distributed vector representation $r_{R_i^p}$. Then, the LSTM encoder summarizes the whole sequence up to position i into a single vector h_i^p .

$$h_i^p = f_{LSTM}(h_{i-1}^p, r_{R_i^p}). \quad (4)$$

This hidden vector contains information about the particular rule being applied and the previously applied rules. Once all the rules in t_p are processed, we get a final hidden representation (h_τ^p). The representations at each time step ($h_1^p, h_2^p, \dots, h_\tau^p$) are used in the decoder to generate rule sequence for the next version of the tree. The parameters of the LSTM and the rules representations $r_{R_i^p}$ are randomly initialized and learned jointly with all other model parameters.

Decoder. Our decoder has an LSTM with an attention mechanism as described by Bahdanau *et al.* [39]. The decoder LSTM is initialized with the final output from the encoder, i.e., $h_0^n = h_\tau^p$. At a given decoding step k the decoder LSTM changes its internal state in the following way,

$$h_k^n = f_{LSTM}(h_{k-1}^n, \psi_k), \quad (5)$$

where ψ_k is computed by the attention-based weighted sum of the inputs h_j^p as [39] in , i.e.,

$$\psi_k = \sum_{j=1}^{\tau} \text{softmax}(\mathbf{h}_{k-1}^{n-1}{}^T \mathbf{h}_j^p) \mathbf{h}_j^p. \quad (6)$$

Then, the probability over the rules at the k th step is

$$P(R_k^n | R_1^n, \dots, R_{k-1}^n, t_p) = \text{softmax}(W_{tree} \cdot h_k^n + \mathbf{b}_{tree}). \quad (7)$$

At each timestep, we pick a derivation rule R_k^n following Equation (7) to expand the frontier_node (n_f^t) in a depth-first, left-to-right fashion. When a terminal node is reached, it is recorded to be used in \mathcal{M}_{token} and the decoder proceeds to next non-terminal. In Equation (7), W_{tree} and \mathbf{b}_{tree} are parameters that are jointly learned along with the LSTM parameters of the encoder and decoder.

4.2 Token Generation Model (\mathcal{M}_{token})

We now focus on generating a concrete code fragment c , i.e., a sequence of tokens (x_1, x_2, \dots) . For the edit task, the probability of an edited token x_k^n depends not only on the tokens

of the previous version (x_1^p, \dots, x_m^p) but also on the previously generated tokens x_1^n, \dots, x_{k-1}^n . The next token x_k^n also depends on the token type (θ), which is generated by \mathcal{M}_{tree} . Thus,

$$P(c_n | c_p, t_n) = \prod_{k=1}^{m'} P(x_k^n | x_1^n, \dots, x_{k-1}^n, \{x_1^p, \dots, x_m^p\}, \theta_k^n). \quad (8)$$

Here, θ_k^n is the node type corresponding to the generated terminal token x_k^n . Note that, the token generation model can be viewed as a conditional probabilistic translation model where token probabilities are conditioned not only on the context but also on the type of the token (θ_k^n). Similar to \mathcal{M}_{tree} , we use an encoder-decoder. The encoder encodes each token and corresponding type of the input sequence into a hidden representation with an LSTM (Fig. 1c). Then, for each token (x_i^p) in the previous version of the code, the corresponding hidden representation (s_i^p) is given by: $s_i^p = f_{LSTM}(s_{i-1}^p, enc([x_i^p, \theta_i^p]))$. Here, θ_i^p is the terminal token type corresponding to the generated token x_i^p and $enc()$ is a function that encodes the pair of x_i^p, θ_i^p to a (learnable) vector representation.

The decoder's initial state is the final state of the encoder. Then, it generates a probability distribution over tokens from the vocabulary. The internal state at time step k of the token generation is $s_k^n = f_{LSTM}(s_{k-1}^n, enc(x_i^n, \theta_k^n), \xi_k)$, where ξ_k is the attention vector over the previous version of the code and is computed as in Equation (6). Finally, the probability of the k th target token is computed as

$$P(x_k^n | x_1^n, \dots, x_{k-1}^n, \{x_1^p, \dots, x_m^p\}, \theta_k^n) = \text{softmax}(W_{token} \cdot s_k^n + \mathbf{b}_{token} + \text{mask}(\theta_k^n)). \quad (9)$$

Here, W_{token} and \mathbf{b}_{token} are parameters that are optimized along with all other model parameters. Since not all tokens are valid for all the token types, we apply a mask that deterministically filters out invalid candidates. For example, a token type of `boolean_value`, can only be concretized into `true` or `false`. Since the language grammar provides this information, we create a mask ($\text{mask}(\theta_k^n)$) that returns a $-\infty$ value for masked entries and zero otherwise. Similarly, not all variable, method names, type names are valid at every position. We refine the mask based on the variables, method names and type names extracted from the scope of the change. In the case of method, type and variable names, CODIT allows \mathcal{M}_{token} to generate a special `<unknown>` token. However, the `<unknown>` token is then replaced by the source token that has the highest attention probability (i.e., the highest component of ξ_k), a common technique in NLP. The mask restricts the search domain for tokens. However, in case to variable, type, and method name \mathcal{M}_{token} can only generate whatever token available to it in the vocabulary (through masking) and whatever tokens are available in input code (through copying).

5 IMPLEMENTATION

Our tree-based translation model is implemented as an edit recommendation tool, CODIT. CODIT learns source code changes from a dataset of patches. Then, given a code fragment to edit, CODIT predicts potential changes that are likely to take place in a similar context. We implement CODIT

TABLE 1
Summary of Datasets Used to Evaluate CODIT

Dataset	# Projects	# Train Examples	# Validation Examples	# Test Examples	# Tokens		# Nodes	
					Max	Average	Max	Average
Code-Change-Data	48	24,072	3,258	5,143	38	15	47	20
Pull-Request-Data [28]	3	4,320	613	613	34	17	47	23
Defects4J-data [44]	6	22,060	2,537	117	35	16	43	21

extending OpenNMT [41] based on PyTorch. We now discuss CODIT’s implementation in details.

Patch Pre-Processing. We represent the patches in a parse tree format and extract necessary information (e.g., grammar rules, tokens, and token-types) from them.

Parse Tree Representation. As a first step of the training process, CODIT takes a dataset of patches as input and parses them. CODIT works at method granularity. For a method patch Δm , CODIT takes the two versions of m : m_p and m_n . Using GumTree, a tree-based code differencing tool [42], it identifies the edited AST nodes. The edit operations are represented as insertion, deletion, and update of nodes *w.r.t.* m_p . For example, in Fig. 1a, **red** nodes are identified as deleted nodes and **green** nodes are marked as added nodes. CODIT then selects the minimal subtree of each AST that captures all the edited nodes. If the size of the tree exceeds a maximum size of *max_change_size*, we do not consider the patch. CODIT also collects the edit context by including the nodes that connect the root of the method to the root of the changed tree. CODIT expands the considered context until the context exceed a tree size threshold (*max_tree_size*). During this process, CODIT excludes changes in comments and literals. Finally, for each edit pair, CODIT extracts a pair (AST_p, AST_n) where AST_p is the original AST where a change was applied, and AST_n is the AST after the changes. CODIT then converts the ASTs to their parse tree representation such that each token corresponds to a terminal node. Thus, a patch is represented as the pair of parse trees (t_p, t_n) .

Information Extraction. CODIT extracts grammar rules, tokens and token types from t_p and t_n . To extract the rule sequence, CODIT traverses the tree in a depth-first pre-order way. From t_p , CODIT records the rule sequence (R_1^p, \dots, R_t^p) and from t_n , CODIT gets (R_1^n, \dots, R_r^n) (Fig. 1b). CODIT then traverses the parse trees in a pre-order fashion to get the augmented token sequences, i.e., tokens along with their terminal node types: (x_*^p, θ_*^p) from t_p and (x_*^n, θ_*^n) from t_n . CODIT traverses the trees in a left-most depth-first fashion. When a terminal node is visited, the corresponding augmented token (x_*^*, θ_*^*) is recorded.

Model Training. We train the tree translation model (\mathcal{M}_{tree}) and token generation model (\mathcal{M}_{token}) to optimize Equations (3) and (8) respectively using the cross-entropy loss as the objective function. Note that the losses of the two models are independent and thus we train each model separately. In our preliminary experiment, we found that the quality of the generated code is not entirely correlated to the loss. To mitigate this, we used top-1 accuracy to validate our model. We train the model for a fixed amount of n_{epoch} epochs using early stopping (with patience of $valid_{patience}$) on the top-1 suggestion accuracy on the validation data. We use stochastic gradient descent to optimize the model.

Model Testing. To test the model and generate changes, we use beam-search [43] to produce the suggestions from \mathcal{M}_{tree} and \mathcal{M}_{token} . First given a rule sequence from the previous version of the tree, CODIT generates K_{tree} rule sequences. CODIT subsequently use these rule sequence to build the actual AST. While building the tree from the rule sequence, CODIT ignores the sequence if the rule sequence is infeasible (i.e., head of the rule does not match the *frontier_node*, n_f^t). Combining the beam search in rule sequence and the tree building procedure, CODIT generate different trees reflecting different structural changes. Then for each tree, CODIT generates K_{token} different concrete code. Thus, CODIT generates $K_{tree} \cdot K_{token}$ code fragments. We sort them based on their probability, i.e., $\log(P(c_n|c_p, t_p)) = \log(P(c_n|c_p, t_n)) \cdot P(t_n|t_p)$. From the sorted list of generated code, we pick the top K suggestions.

6 EXPERIMENTAL DESIGN

We evaluate CODIT for three different types of changes that often appear in practice: (i) code change in the wild, (ii) pull request edits, and (iii) bug repair. For each task, we train and evaluate CODIT on different datasets. Table 1 provides detailed statistics of the datasets we used.

(i) *Code Change Task.* We collected a large scale real code change dataset (*Code-Change-Data*) from 48 open-source projects from GitHub. These projects also appear in TravisTorrent [30] and have at least 50 commits in Java files. These project are excludes any forked project, toy project, or unpopular projects (all the projects have at least 10 watchers in Github). Moreover, these projects are big and organized enough that they use Travis Continuous integration system for maintainability. For each project, we collected the revision history of the main branch. For each commit, we record the code before and after the commit for all Java files that are affected. In total we found java 241,976 file pairs. We then use GumTree [42] tool to locate the change in the file and check whether the changes are inside a method in the corresponding files. Most of the changes that are outside a method in a java class are changes related to import statements and changes related to constant value. We consider those out of CODIT’s scope. We further remove any pairs, where the change is only in literals and constants. Excluding such method pairs, we got 182,952 method pairs. We set *max_change_size* = 10 and *max_tree_size* = 20. With this data collection hyper-parameter settings, we collected 44,382 patches.

We divide every project based on their chronology. From every project, we divide earliest 70 percent patches into train set, next 10 percent into validation set and rest 20 percent into test set based on the project chronology. We removed any exact test and validation examples from the training set.

We also removed intra-set duplicates. After removing such duplicate patches, we ended up with 32,473 patches in total, which are then used to train and evaluate CODIT.

(ii) *Pull Request Task*. For this task, we use *Pull-Request-Data*, provided by Tufano *et al.* [19] which contains source code changes from merged pull requests from three projects from Gerrit [45]. Their dataset contains 21,774 method pairs in total. Similar to the *Code-Change-Data*, we set $\max_change_size = 10$ and $\max_tree_size = 20$ to extract examples that are in CODIT's scope from this dataset. We extracted 5546 examples patch pair.

(iii) *Bug Repair Task*. For this task, we evaluate CODIT on Defects4J [44] bug-fix patches. We extract 117 patch pairs across 80 bug ids in Defect4J dataset with the same $\max_change_size = 10$ and $\max_tree_size = 20$ limit as Code Change Task. These are the bugs that are in CODIT's scope. To train CODIT for this task, we create a dataset of code changes from six projects repositories in Defects4J dataset containing 24,597 patches. We remove the test commits from the training dataset.

6.1 Evaluation Metric

To evaluate CODIT, we measure for a given code fragment, how accurately CODIT generates patches. We consider CODIT to correctly generate a patch if it exactly matches the developer produced patches. CODIT produces the top K patches and we compute CODIT's accuracy by counting how many patches are correctly generated in top K . Note that this metric is stricter than semantic equivalence.

For the bug fix task, CODIT takes a buggy line as input and generates the corresponding patched code. We consider a bug to be fixed if we find a patch that passes all the test cases. We also manually investigate the patches to check the similarity with the developer provided patches.

6.2 Baseline

We consider several baselines to evaluate CODIT's performance. Our first baseline is a vanilla LSTM based Seq2Seq model with attention mechanism [39]. Results of this baseline indicate different drawbacks of considering raw code as a sequence of token.

The second baseline, we consider, is proposed by Tufano *et al.* [27]. For a given code snippet (previous version), Tufano *et al.* first abstracted the identifier names and stored the mapping between original and abstract identifiers in a symbol table. The resultant abstracted code (obtained by substituting the raw identifiers with abstract names) is then translated by an NMT model. After translation, they concretize the abstract code using the information stored in the symbol table. The patches where the abstract symbols predicted by the model are not found in the symbol table remain undecidable. Such patches, although can be useful to guide developers similar to our \mathcal{M}_{tree} , cannot be automatically concretized, and thus, we do not count them as fully correct patches.

Both the vanilla Seq2Seq and Tufano *et al.*'s model consider the before version of the code as input. Recently, SequenceR [28] proposed way to represent additional context to help the model generate concrete code. We design such a baseline, where we add additional context to c_p . Following

SequenceR, we add copy attention, where the model learns to copy from the contexted code.

To understand the tree encoding mechanism, we used several tree encoders. First method we considered is similar to DeepCom [46], where the AST is represented as a sequential representation called Structure Based Traversal (SBT). Second tree encoding method we consider is similar to Code2Seq, where code AST is represented by a set of paths in the tree. While these tree encoding methods are used for generating Code comment, we leverage these encoding methods for code change prediction. We design a Seq2Seq method with DeepCom encoder (Tree2Seq), and Code2Seq encoder. We also enable the copy attention in both of these baselines.

We also did some experiment on Kommrusch *et al.* [47]'s proposed Graph2Seq network, which they provided implementation as an add on in OpenNMT-Py framework.² However, their implementation is assumed upon a very restricted vocabulary size. To understand better, we modified their code to fit into our context, but neither their original model, nor our modified model works in program change task. When we increase the vocabulary size in their version of the model, it does not scale and quickly exhaust all the memory in the machine (we tried their model on a machine with 4 Nvidia TitanX GPU and 256 GB RAM). When we tested our modification of their code, the model does not converge even after 3 days of training, eventually resulting in 0 correct predictions.

The basic premise of CODIT is based on the fact that code changes are repetitive. Thus, another obvious baseline is to see how CODIT performs *w.r.t.* to code-clone based edit recommendation tool [3]. In particular, given a previous version (c_p) of an edit, we search for the closest k code fragments using similar bag-of-words at the token level similar to Sajnani *et al.* [48]. In our training data of code edits, this step searches in the previous code versions and use the corresponding code fragments of the next version as suggested changes.

Bug-Fixing Baselines. For the bug fix task, we compare CODIT's performance with two different baselines. Our first baseline is SequenceR [28], we compare with the results they reported. We also compare our result with other non-NMT based program repair systems.

7 RESULTS

We evaluate CODIT's performances to generate concrete patches *w.r.t.* generic edits (RQ1) and bug fixes (RQ3). In RQ2, we present an ablation study to evaluate our design choices.

RQ1. How accurately can CODIT suggest concrete edits?

To answer this RQ, we evaluate CODIT's accuracy *w.r.t.* the evaluation dataset containing concrete patches. Table 2 shows the results: for *Code-Change-Data*, CODIT can successfully generate 201 (3.91 percent), 571 (11.10 percent), and 820 (15.94 percent) patches at top 1, 2, and 5 respectively. In contrast, at top 1, SequenceR generates 282 (5.48 percent) correct patches, and performs the best among all the methods. While SequenceR outperforms CODIT in top 1, CODIT outperforms SequenceR with significant margin at top 2 and top 5.

2. <https://opennmt.net/OpenNMT-py/ggnn.html>

TABLE 2
Performance of CODIT Suggesting Concrete Patches

Method	Code Change Data			Pull Request Data		
	Number of examples : 5143			Number of examples : 613		
	Top-1	Top-2	Top-5	Top-1	Top-2	Top-5
Token Based	Seq2Seq	107 (2.08%)	149 (2.9%)	194 (3.77%)	45 (7.34%)	55 (8.97%)
	Tufano <i>et al.</i>	175 (3.40%)	238 (4.63%)	338 (6.57%)	81 (13.21%)	104 (16.97%)
	SequenceR	282 (5.48%)	398 (7.74%)	502 (9.76%)	39 (6.36%)	137 (22.35%)
Tree Based	Tree2Seq	147 (2.86%)	355 (6.9%)	568 (11.04%)	39 (6.36%)	89 (14.52%)
	Code2seq	58 (1.12%)	82 (1.59%)	117 (2.27%)	4 (0.65%)	7 (1.14%)
	CODIT	201 (3.91%)	571 (11.10%)	820 (15.94%)	57 (9.3%)	134 (21.86%)
IR based	\mathcal{B}_{ir}	40 (0.77%)	49 (0.95%)	61 (1.18%)	8 (1.30%)	8 (1.30%)
For Token Based models, predominant source of information in the code are token sequences. For Tree Based models information source is code AST. For IR based method, information retrieval model is used on code.						

In Pull-Request-Data, CODIT generates 57 (9.3 percent), 134 (21.86 percent), and 177 (28.87 percent) correct patches at top 1, 2, and 5 respectively. At top 1, Tufano *et al.*'s [27] method produces 81 patches. At top 2, CODIT produces 134 (21.86 percent) patches, which is comparable with SequenceR's result 137 (22.35 percent). At top 5, CODIT outperforms all the other baselines achieving 9.2 percent gain over SequenceR baseline.

The main advantage point of CODIT is that, since it considers the structural changes separate from the token changes, it can learn the structural change pattern well instead of being dominated by learning the code token changes. However, being a two stage process, CODIT has two different hinge point for failure. If \mathcal{M}_{tree} does not generate the correct tree, no matter how good \mathcal{M}_{token} performs, CODIT is unable to generate correct patch. We conjecture that, this is the reason for CODIT's failure at top 1.

Among the baselines we compared here, SequenceR, and Tree2Seq takes the advantage of copy attention. Tufano *et al.*'s model takes the advantage of reduced vocabulary through identifier abstraction. Tufano *et al.*'s model works best when the code is mostly self contained i.e., when there is always a mapping from abstract identifier to concrete identifier in the symbol table, their method can take full advantage of the vocabulary reduction. When the input code is mostly self contained (i.e., most of the tokens necessary to generate a patch are inside the input code c_p or appears within the context limit in as presented to SequenceR).

In code change task with NMT, the deterministic positions of code tokens are important to put attention over different parts of code. While Code2Seq [49]'s representation of input code as random collection of paths in the code showed success for code comprehension, it does not generalize well for code change due to the stochastic nature of the input. Additionally, copy attention cannot be trivially applied to Code2Seq since, like attention, copy also rely on the defined positions of tokens in the input.

While CODIT replaces any <unknown> prediction by the token with highest attention score (see Section 4.2), unlike SequenceR, CODIT does not learn to copy. The rationale is, unlike SequenceR, CODIT's input code (c_p) is not enhanced by the context. Instead, we present the context to the CODIT through the token mask (see Equation (9)). If we enable copy attention, CODIT becomes highly biased by the tokens that are inside c_p .

Note that, “vocabulary explosion” still remains an open problem for code generation. Neither CODIT nor any other baselines we discussed here solve this problem. CODIT presents a way to learn the structural change and restricts the search domain for token names through a mask. For instance, where \mathcal{M}_{token} needs to generate a token of primitive data type (\mathcal{M}_{token} knows the token type because \mathcal{M}_{tree} already generated a tree with terminal node types), it can restrict the search over the primitive types only. While it is expected that the decoder of an ideal Seq2Seq model would inherently learn appropriate tokens at appropriate positions as it implicitly learns code structure, in reality, because of its unrestricted search space, they tend to mispredict more tokens than CODIT.

In general, CODIT along with all the baselines perform better when generating small patches. For example, a large majority of the correctly generated patches have size of one (i.e., $\Delta_t = 1$, the tree distance between t_p and t_n [50]). However, a non-trivial number of larger patches are also correctly generated. Fig. 2 shows the histogram of the size of correctly predicted patches. For example, there are 202 and 51 correct patches with $\Delta_t \geq 3$ generated by CODIT for Code-Change-Data and Pull-Request-Data respectively.

For qualitative evaluation, we show some non-trivial patches that CODIT can successfully generate. CODIT can learn a wide range of patch patterns. Table 3 shows few examples of different category of patches that CODIT can generate. CODIT also shows promise in generating non-trivial structure based changes. Consider Example 4 where x is renamed to session both the formal parameter and the usage in the body. Note that, since CODIT uses a tree-based model it is good at capturing long-distance dependencies

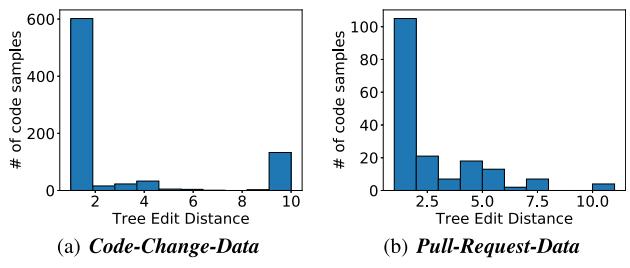


Fig. 2. Patch size (Tree Edit-distance) histogram of correctly generated patches in different datasets.

TABLE 3
Examples of Different Types of Concrete
Patches Generated by CODIT

Example 1. API Change

```
return f.createJsonParser createParser(...)
```

Example 2. Type Change

```
void appendTo(StringBuffer StringBuider buffer)
```

Example 3. Parameter: Add/Delete Method Parameter

```
1. testDataPath(false , true , true , true, false );
2. assertNotificationEnqueued(map , key , value ,hash)
```

Example 4. Refactoring: Modify Method Parameters Name

```
void visit(JSession * session , ... ) throws Exception
{
    visit (((JNode) (* session)), ...);
}
```

Example 5. Statement: Add Statement

```
...
interruptenator.shutdown();
Thread.interrupted();
}
```

Example 6. Inheritance: Abstracting a Method

```
public abstract void removeSessionCookies (...)

+ throw new android...MustOverrideException();
```

Example 7. Exception Change: Add Try Block

```
public void copyFrom( java.lang.Object arr){
+ try{
    android.os.Trace.traceBegin (...);
+ finally{
    android.os.Trace.traceEnd(...);
+ }
}
```

Example 8. Other: Delete Unreferenced Variable

```
public void testConstructor2NPE(){
...
-AtomicIntegerArray aa= new AtomicIntegerArray(a);
shouldThrow ();
...
}
```

Example 9. Final: Make Method Parameter Immutable

```
public String print(final ReadableInstant inst){...}
```

Example 10. Access: Modify Access Type

```
public protected AbstractInstant () { super(); }
```

Every cell shows an example of correctly suggested patches by CODIT. Top line is the patch category, followed by the actual patch. In the patch, Red tokens/lines are deleted and Green tokens/lines are added.

allowing the token-level model to focus on predicting tokens, e.g., such that it can rename the same variable similarly. Another interesting example where CODIT can successfully generate patches is shown in example 6, where CODIT does not only add the **abstract** keyword in the method signature, but also removes the body. Since CODIT is aware of code syntax, it learns that method declarations with an **abstract** keyword have a high probability of an empty method body.

Table 4 shows some additional examples where CODIT can successfully generate patches. In these examples, different exception/error types (i.e., **Exception**, **Error**, **RuntimeException**) are changed to **EOFException** although their usage differs. In the first three examples **EOFException**

TABLE 4
Examples of CODIT's Ability to Generalize in Different Use Cases

```
Ex:1. return Error EOFException.class ;
Ex:2. return Exception EOFException.class ;
Ex:3. return RuntimeException EOFException.class ;
Ex:4. return new Error EOFException(msg) ;
Ex:5. return new Exception EOFException(msg) ;
Ex:6. return new RuntimeException EOFException(msg) ;
```

Exception, Error, RuntimeException are modified to EOFException under different context.

is used as a class reference, while for the others **EOFException** is used to initialize an object. These examples also illustrate CODIT's ability to generalize to different contexts and use-cases. Other structural transformation that CODIT include, but not limited to, include scoping (example 7 in Table 3), adding/deleting method parameters (example 3 in Table 3), changing method/variable access modifiers (example 9, 10 in Table 3), etc.

Result 1: CODIT suggests 15.94 percent correct patches for Code-Change-Data and 28.87 percent for Pull-Request-Data within top 5 and outperforms best baseline by 44.37 and 9.26 percent respectively.

Next, we evaluate CODIT's sub-components.

RQ2. How do different design choices affect CODIT's performance?

This RQ is essentially an ablation study where we investigate in three parts: (i) the token generation model (M_{token}), (ii) the tree translation model (M_{tree}), and (iii) evaluate the combined model, i.e., CODIT, under different design choices. We further show the ablation study on the data collection hyper-parameters (i.e., max_change_size , and max_tree_size) and investigate the cross project generalization.

Evaluating Token Generation Model. Here we compare M_{token} with the baseline SequenceR model. Note that M_{token} generates a token given its structure. Thus, for evaluating the standalone token generation model in CODIT's framework, we assume that the true structure is given (emulating a scenario where a developer knows the kind of structural change they want to apply). Table 5 presents the results.

While the baseline Seq2Seq with copy-attention (SequenceR) generates 9.76 percent (502 out of 5,143) and 26.43 percent (162 out of 613) correct patches for *Code-Change-Data* and *Pull-Request-Data* respectively at top 5, Table 5 shows that if the change structure (i.e., t_n) is known, the standalone M_{token} model of CODIT can generate 39.57 percent (2,035 out of 6,832) and 61.66 percent (378 out of 613) for *Code-Change-Data* and *Pull-Request-Data* respectively. This result empirically shows that if the tree structure is known, NMT-based code change prediction significantly improves. In fact, this observation led us build CODIT as a two-stage model.

Evaluating Tree Translation Model. Here we evaluate how accurately M_{tree} predicts the structure of a change — shown in Table 6. M_{tree} can predict 56.78 and 55.79 percent of the structural changes in *Code-Change-Data* and *Pull-Request-Data* respectively. Note that, the outputs that are generated by M_{tree} are not concrete code, rather structural

TABLE 5
Correct Patches Generated by the Standalone Token Generation Model When the True Tree Structure is Known

Dataset	Total Correct Patches	
	SequenceR	standalone \mathcal{M}_{token}
Code-Change-Data	502 (9.76%)	2035 (39.57%)
Pull-Request-Data	162 (26.43%)	378 (61.66%)

abstractions. In contrast, Tufano *et al.*'s [27] predicts 38.56 and 39.31 percent correct patches in *Code-Change-Data* and *Pull-Request-Data* respectively. Note that, their abstraction and our abstraction method is completely different. In their case, for some of the identifiers (those already exist in the symbol table), they have a deterministic way to concretize the code. In our case, \mathcal{M}_{token} in CODIT augments \mathcal{M}_{tree} by providing a stochastic way to concretize every identifier by using NMT.

Note that, not all patches contain structural changes (e.g., when a single token, such as a method name, is changed). For example, 3,050 test patches of *Code-Change-Data*, and 225 test patches of *Pull-Request-Data* do not have structural changes. When we use these patches to train \mathcal{M}_{tree} , we essentially train the model to sometimes copy the input to the output and rewarding the loss function for predicting no transformation. Thus, to report the capability of \mathcal{M}_{tree} to predict structural changes, we also train a separate version of \mathcal{M}_{tree} using only the training patches with at least 1 node differing between t_n and t_p . We also remove examples with no structural changes from the test set. This is our filtered dataset. In the filtered dataset, \mathcal{M}_{tree} predicts 33.61 and 30.73 percent edited structures from *Code-Change-Data* and *Pull-Request-Data* respectively. This gives us an estimate of how well \mathcal{M}_{tree} can predict structural changes.

Evaluating CODIT. Having \mathcal{M}_{tree} and \mathcal{M}_{token} evaluated separately, we will now evaluate our end-to-end combined model ($\mathcal{M}_{tree} + \mathcal{M}_{token}$) focusing on two aspects: (i) effect of attention-based copy mechanism, (ii) effect of beam size.

First, we evaluate contribution of CODIT's attention-based copy mechanism as described in Section 4.2. Table 7 shows the results. Note that, unlike SequenceR, CODIT is not trained for learning to copy. Our copy mechanism is an auxiliary step in the beam search that prohibits occurrence of `<unknown>` token in the generated code.

Recall that \mathcal{M}_{token} generates a probability distribution over the vocabulary (Section 4.2). Since the vocabulary is generated using the training data, any unseen tokens in the test patches are replaced by a special `<unknown>` token. In our experiment, we found that a large number (about 3 percent is

TABLE 7
CODIT Performance w.r.t. to the Attention Based Copy Mechanism @top-5 ($K_{tree} = 2$, $K_{token} = 5$)

Dataset	lower bound	upper bound	CODIT
Code-Change-Data	742 (14.42%)	898 (17.46%)	820 (15.94%)
Pull-Request-Data	163 (26.59%)	191 (31.16%)	177 (28.87%)

Lower bound is without copy. The upper bound evaluates with oracle copying predictions for `<unknown>`. For CODIT each `<unknown>` token is replaced by the source token with the highest attention.

Code-Change-Data and about 4 percent is *Pull-Request-Data*) of patches contain `<unknown>` tokens; this is undesirable since the generated code will not compile. When we do not replace `<unknown>` tokens, CODIT can predict 742 (14.42 percent), and 163 (26.59 percent) correct patches in *Code-Change-Data* and *Pull-Request-Data* respectively. However, if all the `<unknown>` tokens could be replaced perfectly with the intended token, i.e., upper bound of the number of correct patches goes up to 898 (17.46 percent) and 191 (31.16 percent) correct patches in *Code-Change-Data* and *Pull-Request-Data* respectively. This shows the need for tackling the `<unknown>` token problem. To solve this, we replace `<unknown>` tokens predicted by \mathcal{M}_{token} with the source token with the highest attention probability following Section 4.2. With this, CODIT generates 820 (15.94 percent), and 177 (28.87 percent) correct patches from *Code-Change-Data* and *Pull-Request-Data* respectively (Table 7).

Second, we test two configuration parameters related to the beam size, K_{tree} and K_{token} i.e., the number of trees generated by \mathcal{M}_{tree} and number of concrete token sequences generated by \mathcal{M}_{token} per tree (Section 5). While Table 2 shows the effect of different values of K_{token} (e.g., 1, 2, 5), here we investigate the effect of K_{tree} for a given K_{token} . Fig. 3 shows the parameter sensitivity of K_{tree} when K_{token} is set 5. Recall from Section 5, CODIT first generates K_{tree} number of trees, and then generates K_{token} number of code per tree. Among those $K_{tree} * K_{token}$ generated code, CODIT chooses top K_{token} number of code to as final output. In both *Code-Change-Data* (CC data in Fig. 3), and *Pull-Request-Data* (PR data in Fig. 3), CODIT performs best when $K_{tree} = 2$. When $K_{tree} = 2$, total generated code is 10, among which CODIT chooses top 5. With increasing number of K_{tree} , CODIT has to choose from increasing number of generated code, eventually hurting the performance of CODIT.

Next, we move onto ablation study of our data collection hyper-parameters (i.e., `max_change_size`, and `max_tree_size`). For *Code-Change-Data*, we only collected examples patches where `max_change_size = 10`, and `max_tree_size = 20`. For this ablation study, we created 6 different version *Pull-Request-Data* with respective data parameters (see Table 8 for details). As we increase the `max_tree_size` parameter, the length of the code increases causing the performance to decrease. With the increment of `max_change_size`, CODIT's performance also decreases. Furthermore, Fig. 4 shows histogram of percentage of correctly predicted examples by CODIT w.r.t. size of the tree (in terms of nodes). While CODIT performs well in predicting smaller trees (≤ 10 nodes), CODIT also works comparably well in larger tree sizes. In fact, CODIT produces 21.97 percent correct code in *Pull-Request-Data*, and 16.48 percent correct code in *Code-Change-Data* where the tree size is larger than 30 nodes.

TABLE 6
 \mathcal{M}_{tree} Top-5 Performance for Different Settings

Dataset	# Correct Abstract Patches*	
	Tufano <i>et al.</i>	CODIT
Code-Change-Data	1983 / 5143 (38.56%)	2920 / 5143 (56.78%)
Pull-Request-Data	241 / 613 (39.31%)	342 / 613 (55.79%)

* Each cell represents correctly predicted patches / total patches (percentage of correct patch) in the corresponding setting.

TABLE 8
Ablation Study of $\max_change_size(max_c)$, and
 $\max_tree_size(max_t)$

# Train	# Valid	# Test	max_c	max_t	% correct
3,988	593	593	10	10	34.06
4,320	613	613	10	20	32.63
4,559	622	627	10	30	28.07
7,069	948	932	20	20	31.55
7,340	993	948	20	30	29.32
9,361	1,227	1,213	30	30	24.48

To understand how CODIT generalizes beyond a project, we do a cross-project generalization test. Instead of chronological split of examples (see Section 6), we split the examples based on projects, i.e., all the examples that belongs to a project falls into only one split (train/validation/test). We then train and test CODIT based on this data split. Table 9 shows the result in this settings *w.r.t.* to intra-project split. While \mathcal{M}_{tree} in intra-project and cross-project evaluation setting achieves similar performance, full CODIT performance deteriorate by 68 percent. The main reason behind such deterioration is diverse choice of token name across different projects. Developers tend to use project specific naming convention, api etc. This also indicates that the structural change pattern that developers follow are more ubiquitous across different projects than the token changes.

Result 2: CODIT yields the best performance with a copy-based attention mechanism and with tree beam size of 2. \mathcal{M}_{tree} achieves 58.78 and 55.79 percent accuracy and \mathcal{M}_{token} achieves 39.57 and 61.66 percent accuracy in Code-Change-Data and Pull-Request-Data respectively when tested individually.

Finally, we evaluate CODIT’s ability to fix bugs.

RQ3. How accurately CODIT suggests bug-fix patches?

We evaluate this RQ with the state-of-the-art bug-repair dataset, Defects4J [44] using all six projects.

Training. We collect commits from the projects’ original GitHub repositories and preprocess them as described in Section 5. We further remove the Defects4J bug fix patches and use the rest of the patches to train and validate CODIT.

Testing. We extract the methods corresponding to the bug location(s) from the buggy-versions of Defects4J. A bug can have fixes across multiple methods. We consider each

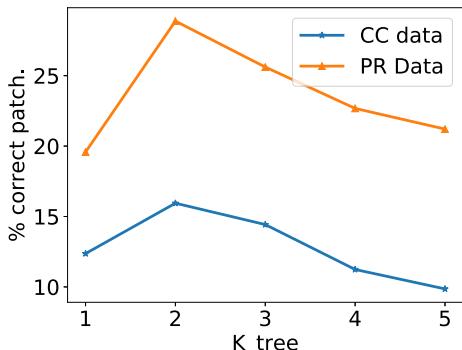


Fig. 3. Performance of CODIT @top-5 ($K_{token} = 5$) for different K_{tree} .

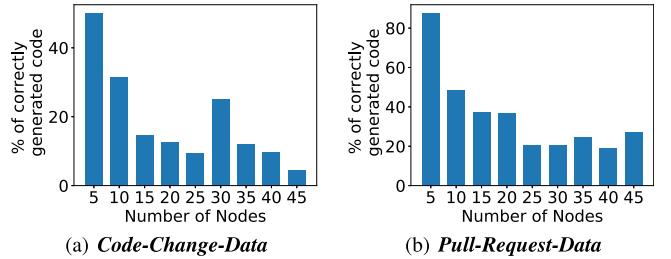


Fig. 4. Percentage of correct prediction by CODIT with respect to number of nodes in the tree.

method as candidates for testing and extract their ASTs. We then filter out the methods that are not within our accepted tree sizes. In this way, we get 117 buggy method ASTs corresponding to 80 bugs. The rest of the bugs are ignored.

Here we assume that a fault-localization technique already localizes the bug [51]. In general, fault-localization is an integral part of program repair. However, in this paper, we focus on evaluating CODIT’s ability to produce patches rather than an end-to-end repair tool. Since fault localization and fixing are methodologically independent, we assume that bug location is given and evaluate whether CODIT can produce the correct patch. Evaluation of CODIT’s promise as a full-fledged bug repair tool remains for future work.

For a buggy method, we extract c_p . Then for a given c_p , we run CODIT and generate a ranked list of generated code fragments (c_n). We then try to patch the buggy code with the generated fragments following the rank order, until the bug-triggering test passes. If the test case passes, we mark it a potential patch and recommend it to developers. We set a specific time budget for the patch generation and testing. For qualitative evaluation, we additionally investigate manually the patches that pass the triggering test cases to evaluate the semantic equivalence with the developer-provided patches. Here we set the maximum time budget for each buggy method to 1 hour. We believe this is a reasonable threshold as previous repair tools (e.g., Elixir [52]) set 90 minutes for generating patches. SimFix [53] set 5 hours as their time out for generating patches and running test cases.

CODIT can successfully generate at least 1 patch that passes the bug-triggering test case for 51 methods out of 117 buggy methods from 30 bugs, i.e., 43.59 percent buggy methods are potentially fixed. Fig. 5 shows the number of patches passing the bug-triggering test case *w.r.t.* time. We see that, 48 out of 51 successful patches are generated within 20 minutes.

We further manually compare the patches with the developer-provided patches: among 51 potential patches, 30 patches are identical and come from 25 different bug ids (See Table 10). The bugs marked in green are completely fixed by CODIT with all their buggy methods being successfully fixed.

TABLE 9
Cross Project Generalization Test for CODIT
(% of Correct at @top-5)

Settings	CODIT full	\mathcal{M}_{tree} only
Intra-Project Split	15.94	56.78
Cross Project Split	9.48	59.65

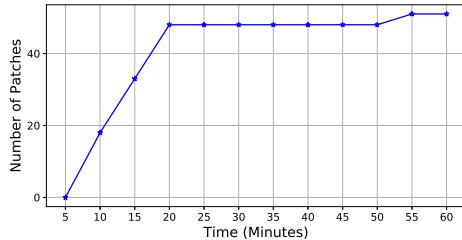


Fig. 5. Patches passing the bug-triggering tests v.s. time.

For example, Math-49 has 4 buggy methods, CODIT fixes all four. For the bugs marked in blue†, CODIT fixes all the methods that are in scope. For example, for Lang-4, there are 2 methods to be fixed, 1 of them are in CODIT’s scope, and CODIT fixes that. However, for two other bugs (marked in orange*), CODIT produces only a partial fix. For example, in the case of Math-46 and Mockito-6, although all the methods are within scope, CODIT could fix 1 out of 2 and 2 out of 20 methods respectively. The ‘Patch Type’ column further shows the type of change patterns.

SequenceR [28] is a notable NMT based program repair tool which takes the advantage of *learning to copy* in NMT. They evaluated on 75 one line bugs in Defects4J dataset and reported 19 plausible and 14 fully correct successful patches. Among those 75 bugs, 38 are in CODIT’s scope. Out of those 38 bugs, CODIT can successfully generate patches for 14 bugs. Note that, we do not present CODIT as full fledged automated program repair tool, rather a tool for guiding developers. Thus, for automatic evaluation, we assumed the correct values of constants (of any data type) given.

One prominent bug repair approach [52], [54], [55] is to transform a suspicious program element following some change patterns until a patch that passes the test cases is found. For instance, Elixir [52] used 8 predefined code transformation patterns and applied those. In fact, CODIT can generate fixes for 8 bugs out of 26 bugs that are fixed by Elixir [52].

Nevertheless, CODIT can be viewed as a transformation schema which automatically learns these patterns without human guidance. We note that CODIT is *not* explicitly focused on bug-fix changes since it is trained with generic changes. Even then, CODIT achieves good performance in Defects4J bugs. Thus, we believe CODIT has the potential to complement existing program repair tools by customizing the training with previous bug-fix patches and allowing to learn from larger change sizes. Note that, current version of CODIT does not handle multi-hunk bugs. Even if a bug is multi-hunk, in current prototype, we consider each of the hunk as separate input to CODIT. For instance, consider Math-46, which is a 2-hunk bug. While all 2 methods are in CODIT’s scope, CODIT can only fix one. Currently we do not consider interaction between multiple hunks [56]. We leave the investigation of NMT in such scenario for future work.

Result 3: CODIT generates complete bug-fix patches for 15 bugs and partial patches for 10 bugs in Defects4J.

8 THREATS TO VALIDITY

External Validity. We built and trained CODIT on real-world changes. Like all machine learning models, our hypothesis

TABLE 10
CODIT’s Performance on Fixing Defects4J [44] Bugs

Project	BugId	# methods to be patched	# methods in scope	# methods CODIT can fix	Patch Type
Chart	8	1	1	1	Api Change
	10	1	1	1	Method-Invocation
	11	1	1	1	Variable-Name-Change
	12	1	1	1	Api-Change
Closure	3†	2†	1†	1†	Method-Invocation†
	75†	2†	1†	1†	Return-Value-Change†
	86	1	1	1	Boolean-Value-Change
	92	1	1	1	Api-Change†
	93	1	1	1	Api-Change
Lang	4†	2†	1†	1†	Method-Invocation†
	6	1	1	1	Method-Parameter-Change
	21	1	1	1	Method-Parameter-Change
	26	1	1	1	Method-Parameter-Add
	30†	5†	1†	1†	Type-Change†
Math	6†	13†	1†	1†	Method-Parameter-Change†
	30	1	1	1	Type-Change
	46*	2*	2*	1*	Ternary-Statement-Change*
	49	4	4	4	Object-Reference-Change
	57	1	1	1	Type-Change
	59	1	1	1	Ternary-Statement-Change
	70	1	1	1	Method-Parameter-Add
Mockito	98	2	2	2	Array-Size-Change
	6*	20*	20*	2*	Api-Change*
	25†	6†	1†	1†	Method-Parameter-Add†
	30†	2†	1†	1†	Method-Parameter-Add†

Green rows are bug ids where CODIT can produce complete patch. Blue† rows are where CODIT can fix all the methods that are in CODIT’s scope. Orange* rows are where CODIT could not fix all that are in CODIT’s scope.

is that the dataset is representative of real code changes. To mitigate this threat, we collected patch data from different repositories and different types of edits collected from real world.

Most NMT based model (or any other text decoder based models) faces the “vocabulary explosion” problem. That problem is even more prominent in code modeling, since possible names of identifiers can be virtually infinite. While this problem is a major bottleneck in DL base code generation, CODIT does not solve this problem. In fact, similar to previous researches (i.e., SequenceR [28]), CODIT cannot generate new identifiers if it is not in the vocabulary or in the input code.

Internal Validity. Similar to other ML techniques, CODIT’s performance depends on hyperparameters. To minimize this threat, we tune the model with a validation set. To check for any unintended implementation bug, we frequently probed our model during experiments and tested for desirable qualities. In our evaluation, we used exact similarity as an evaluation metric. However, a semantically equivalent code may be syntactically different, e.g., refactored code. We will miss such semantically equivalent patches. Thus, we give a lower bound for CODIT’s performance.

9 RELATED WORK

Modeling Source Code. Applying ML to source code has received increasing attention in recent years [26] across many applications such as code completion [11], [31], bug prediction [57], [58], [59], clone detection [60], code search [61], etc.

In these work, code was represented in many form, e.g., token sequences [31], [32], parse-trees [62], [63], graphs [59], [64], embedded distributed vector space [65], etc. In contrast, we aim to model code changes, a problem fundamentally different from modeling code.

Machine Translation (MT) for Source Code. MT is used to translate source code from one programming language into another [66], [67], [68], [69]. These works primarily used Seq2Seq model at different code abstractions. In contrast, we propose a syntactic, tree-based model. More closely to our work, Tufano *et al.* [19], [27], and Chen *et al.* [28] showed promising results using a Seq2Seq model with attention and copy mechanism. Our baseline Seq2Seq model is very similar to these models. However, Tufano *et al.* [19], [27] employed a different form of abstraction: using a heuristic, they replace most of the identifiers including variables, methods and types with abstract names and transform previous and new code fragments to abstracted code templates. This vaguely resembles CODIT's M_{tree} that predicts syntax-based templates. However, the abstraction method is completely different. With their model, we achieved 39 percent accuracy at top 5. As Table 6, our abstract prediction mechanism also predicts 55-56 percent accurately on different datasets. Since, our abstraction mechanism differs significantly, directly comparing these numbers does *not* yield a fair comparison. Gupta *et al.* used Seq2Seq models to fix C syntactic errors in student assignments [70]. However, their approach can fix syntactic errors for 50 percent of the input codes i.e., for rest of the 50 percent generated patches were syntactically incorrect which is never the case for CODIT because of we employ a tree-based approach. Other NMT application in source code and software engineering include program comprehension [46], [49], [71], [72], commit message generation [73], [74], program synthesis [62], [75] etc.

Structure Based Modeling of Code. Code is inherently structured. Many form of structured modeling is used in source code over the years for different tasks. Allamanis *et al.* [76], [77] proposed statistical modeling technique for mining source code idioms, where they leverages probabilistic Tree Substitution Grammar (pTSG) for mining code idioms. CODIT's M_{tree} is based on similar concept, where we model the derivation rule sequence based on a probabilistic Context Free Grammar. Brockschmidt *et al.* [78], Allmanis *et al.* [59] proposed graph neural network for modeling source code. However, their application scenario is different from CODIT's application, i.e., their focus is mainly on generating natural looking code and/or identify bugs in code. Recent researches that are very close to CODIT include Yin *et al.* [79]'s proposed graph neural network-based distributed representation for code edits but their work focused on change representation than generation. Other recent works that focus on program change or program repair include Graph2Diff by Tarlow *et al.* [80], Hoppity by Dinella *et al.* [81]. These research results are promising and may augment or surpass CODIT's performance, but problem formulation between these approach are fundamentally different. While these technique model the change only in the code, we formulate the problem of code change in encoder-decoder fashion, where encoder-decoder implicitly models the changes in code.

Program Repair. Automatic program repair is a well-researched field, and previous researchers proposed many

generic techniques for general software bugs repair [55], [82], [83], [84], [85]. There are two differnt directions in program repair research : generate and validate approach, and sysnthesis bases approach. In generate and validate approaches, candidate patches are first generated and then validated by running test cases [52], [55], [86], [87], [88]. Synthesis based program repair tools synthesizes program elements through symbolic execution of test cases [89], [90]. CODIT can be considered a program generation tool in generate and validate based program-repair direction. Arcuri *et al.* [91], Le Goues *et al.* [86] built their tool for program repair based on this assumption. Both of these works used existing code as the search space of program fixes. Elixir [52] used 8 predefined code transformation patterns and applied those to generate patches. CapGen [88] prioritize operator in expression and fix ingredients based in the context of the fix. They also relied on predefined transformation patterns for program mutation. In contrast, CODIT learns the transformation patterns automatically. Le *et al.* [92] utilized the development history as an effective guide in program fixing. They mined patterns from existing change history and used existing mutation tool to mutate programs. They showed that the mutants that match the mined patterns are likely to be relevant patch. They used this philosophy to guide their search for program fix. The key difference between Le *et al.* and this work, is that we do not just mine change patterns, but learn a probabilistic model that learns to generalize from the limited data.

Automatic Code Changes. Modern IDEs [7], [8] provide support for automatic editings, e.g., refactoring, boilerplate templates (e.g., try-catch block) etc. There are many research on automatic and semi-automatic [93], [94] code changes as well: e.g., given that similar edits are often applied to similar code contexts, Meng *et al.* [20], [22] propose to generate repetitive edits using code clones, sub-graph isomorphisms, and dependency analysis. Other approaches mine past changes from software repositories and suggest edits that were applied previously to similar contexts [2], [3]. In contrast, CODIT generates edits by learning them from the wild—it neither requires similar edit examples nor edit contexts. Romil *et al.* [21] propose a program synthesis-based approach to generate edits where the original and modified code fragments are the input and outputs to the synthesizer. Such patch synthesizers can be thought of as a special kind of model that takes into account additional specifications such as input-output examples or formal constraints. In contrast, CODIT is a statistical model that predicts a piece of code given only historical changes and does not require additional input from the developer. Finally, there are domain-specific approaches, such as error handling code generation [95], [96], API-related changes [9], [10], [11], [12], [13], [14], [97], automatic refactorings [15], [16], [17], [18], etc. Unlike these work, CODIT focuses on general code changes.

10 DISCUSSION AND FUTURE WORK

Search Space for Code Generation. Synthesizing patches (or code in general) is challenging [98]. When we view code generation as a sequence of token generation problem, the space of the possible actions becomes too large. Existing statistical language modeling techniques endorse the action

space with a probability distribution, which effectively reduces the action space significantly since it allows to consider only the subset of probable actions. The action space can be further reduced by relaxing the problem of concrete code generation to some form of abstract code generation, e.g., generating code sketches [97], abstracting token names [27], etc. For example, Tufano *et al.* reduce the effective size of the action space to $3.53 \cdot 10^{10}$ by considering abstract token names [27]. While considering all possible ASTs allowed by the language's grammar, the space size grows to $1.81 \cdot 10^{35}$. In this work, a probabilistic grammar further reduces the effective action space to $3.18 \cdot 10^{10}$, which is significantly lower than previous methods. Such reduction of the action space allows us to search for code more efficiently.

Ensemble Learning for Program Repair. The overall performance of pre-trained deep-learning models may vary due to the different model architectures and hyper-parameters, even if they are trained on the same training corpus. Moreover, bug fixing patterns are numerous and highly dependent on the bug context and the bug type, so a single pre-trained model may only have the power to fix certain kinds of bugs and miss the others. To overcome this limitation, ensemble learning can be a potential approach to leverage the capacities of different models and learn the fixing patterns in multiple aspects [99]. In future work, we plan to conduct researches on exploring the potentials of ensemble model to improve the performance of CODIT.

Larger Code Edits. Our work has focused on generating small code changes (single-line or single-hunk) since such changes take a non-trivial part of software evolution. However, predicting larger (multi-line and multi-hunk) code changes is important and always regarded as a harder task for current automated program repair techniques. Generating larger code snippets will significantly increase the difficulty of repairing bugs for pure sequence-to-sequence model, since any wrongly predicted token along the code sequence will lead to meaningless patches. CODIT can address this problem as it takes language grammar into consideration. Specifically, the tree translation model could maintain its power when dealing with larger code changes, because the structural changes are much simpler and more predictable than token-level code changes. Given the tree structure of the patch, CODIT will not widely search for tokens in the whole vocabulary, but rather, only possible candidates corresponding to a node type will be considered. Therefore, such hierarchical model may have potential to generate larger edits. We plan to investigate CODIT's ability to multi-line and multi-hunk code edits in the future.

11 CONCLUSION

In this paper, we proposed and evaluated CODIT, a tree-based hierarchical model for suggesting eminent source code changes. CODIT's objective is to suggest changes that are similar to change patterns observed in the wild. We evaluate our work against a large number of real-world patches. The results indicate that tree-based models are a promising tool for generating code patches and they can outperform popular seq2seq alternatives. We also apply our model to program repair tasks, and the experiments show that CODIT is capable of predicting bug fixes as well.

ACKNOWLEDGMENTS

This work was supported in part by National Science Foundation Grant CCF1845893, CCF 1822965, and CNS 1842456.

REFERENCES

- [1] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. Al-Kofahi, and T. N. Nguyen, "Recurring bug fixes in object-oriented programs," in *Proc. 32nd ACM/IEEE Int. Conf. Softw. Eng.*, 2010, pp. 315–324.
- [2] H. A. Nguyen, A. T. Nguyen, T. T. Nguyen, T. N. Nguyen, and H. Rajan, "A study of repetitiveness of code changes in software evolution," in *Proc. 28th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2013, pp. 180–190.
- [3] B. Ray, M. Nagappan, C. Bird, N. Nagappan, and T. Zimmermann, "The uniqueness of changes: Characteristics and applications," in *Proc. IEEE/ACM 12th Work. Conf. Mining Softw. Repositories*, 2015, pp. 34–44.
- [4] M. Martinez, W. Weimer, and M. Monperrus, "Do the fix ingredients already exist? An empirical inquiry into the redundancy assumptions of program repair approaches," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 492–495.
- [5] E. T. Barr, Y. Brun, P. Devanbu, M. Harman, and F. Sarro, "The plastic surgery hypothesis," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 306–317.
- [6] B. Ray, M. Kim, S. Person, and N. Rungta, "Detecting and characterizing semantic inconsistencies in ported code," in *Proc. IEEE/ACM 28th Int. Conf. Autom. Softw. Eng.*, 2013, pp. 367–377.
- [7] Microsoft, "Visual studio (<https://visualstudio.microsoft.com/>)", 2018. [Online]. Available: <https://visualstudio.microsoft.com>
- [8] E. Foundation, "Eclipse IDE (<https://www.eclipse.org/>)", 2018. [Online]. Available: <https://www.eclipse.org>
- [9] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr, A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to API usage adaptation," *ACM SIGPLAN Notices*, vol. 45, no. 10, pp. 302–321, 2010.
- [10] W. Tansey and E. Tilevich, "Annotation refactoring: Inferring upgrade transformations for legacy applications," *ACM SIGPLAN Notices*, vol. 43, no. 10, pp. 295–312, 2008.
- [11] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 419–428, 2014.
- [12] J. Andersen, A. C. Nguyen, D. Lo, J. L. Lawall, and S.-C. Khoo, "Semantic patch inference," in *Proc. 27th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2012, pp. 382–385.
- [13] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller, "Documenting and automating collateral evolutions in linux device drivers," *ACM SIGOPS Operating Syst. Rev.*, vol. 42, no. 4, pp. 247–260, 2008.
- [14] A. T. Nguyen *et al.*, "API code recommendation using statistical learning from fine-grained changes," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 511–522.
- [15] S. R. Foster, W. G. Griswold, and S. Lerner, "WitchDoctor: IDE support for real-time auto-completion of refactorings," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 222–232.
- [16] V. Raychev, M. Schäfer, M. Sridharan, and M. Vechev, "Refactoring with synthesis," *ACM SIGPLAN Notices*, vol. 48, no. 10, pp. 339–354, 2013.
- [17] X. Ge, Q. L. DuBose, and E. Murphy-Hill, "Reconciling manual and automatic refactoring," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 211–221.
- [18] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?" in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 392–402.
- [19] M. Tufano, J. Pantiuchina, C. Watson, G. Bavota, and D. Poshyvanyk, "On learning meaningful code changes via neural machine translation," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 25–36, doi: [10.1109/ICSE.2019.00021](https://doi.org/10.1109/ICSE.2019.00021).
- [20] N. Meng, M. Kim, and K. S. McKinley, "Systematic editing: Generating program transformations from an example," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 329–342, 2011.
- [21] R. Rolim *et al.*, "Learning syntactic program transformations from examples," in *Proc. 39th Int. Conf. Softw. Eng.*, 2017, pp. 404–415.
- [22] N. Meng, M. Kim, and K. S. McKinley, "LASE: Locating and applying systematic edits by learning from examples," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 502–511.
- [23] A. T. Ying, G. C. Murphy, R. Ng, and M. C. Chu-Carroll, "Predicting source code changes by mining change history," *IEEE Trans. Softw. Eng.*, vol. 30, no. 9, pp. 574–586, Sep. 2004.

- [24] T. Zimmermann, A. Zeller, P. Weissgerber, and S. Diehl, "Mining version histories to guide software changes," *IEEE Trans. Softw. Eng.*, vol. 31, no. 6, pp. 429–445, Jun. 2005.
- [25] M. Kim, D. Notkin, and D. Grossman, "Automatic inference of structural changes for matching across program versions," in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 333–343.
- [26] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, 2018, Art. no. 81.
- [27] M. Tufano, C. Watson, G. Bavota, M. Di Penta, M. White, and D. Poshyvanyk, "An empirical investigation into learning bug-fixing patches in the wild via neural machine translation," in *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 832–837.
- [28] Z. Chen, S. J. Kommrusch, M. Tufano, L.-N. Pouchet, D. Poshyvanyk, and M. Monperrus, "SEQUENCER: Sequence-to-sequence learning for end-to-end program repair," in *Proc. IEEE Trans. Softw. Eng.*, to be published, doi: 10.1109/TSE.2019.2940179.
- [29] R. M. Karampatsis and C. Sutton, "How often do single-statement bugs occur? The ManySStuBs4J dataset," 2019, *arXiv: 1905.13334*.
- [30] M. Beller, G. Gousios, and A. Zaidman, "TravisTorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration," in *Proc. 14th Work. Conf. Mining Softw. Repositories*, 2017, pp. 447–450.
- [31] A. Hindle, E. T. Barr, Z. Su, M. Gabel, and P. Devanbu, "On the naturalness of software," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 837–847.
- [32] Z. Tu, Z. Su, and P. Devanbu, "On the localness of software," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 269–280.
- [33] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 763–773.
- [34] M. R. Parvez, S. Chakraborty, B. Ray, and K.-W. Chang, "Building language models for text with named entities," in *Proc. 56th Annu. Meeting Assoc. Comput. Linguistics*, 2018, pp. 2373–2383.
- [35] J. Mallinson, R. Sennrich, and M. Lapata, "Paraphrasing revisited with neural machine translation," in *Proc. 15th Conf. Eur. Chapter Assoc. Comput. Linguistics*, 2017, vol. 1, pp. 881–893.
- [36] D. E. Knuth, "Semantics of context-free languages," *Math. Syst. Theory*, vol. 2, no. 2, pp. 127–145, 1968.
- [37] N. Chomsky, "Three models for the description of language," *IRE Trans. Inf. Theory*, vol. 2, no. 3, pp. 113–124, 1956.
- [38] J. E. Hopcroft, *Introduction to Automata Theory, Languages, and Computation*. Noida, Uttar Pradesh: Pearson Education India, 2008.
- [39] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," in *Proc. 3rd Int. Conf. Learn. Representations*, May 2015, pp. 7–9. [Online]. Available: <http://arxiv.org/abs/1409.0473>
- [40] M. Allamanis, H. Peng, and C. Sutton, "A convolutional attention network for extreme summarization of source code," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 2091–2100.
- [41] G. Klein, Y. Kim, Y. Deng, J. Senellart, and A. M. Rush, "OpenNMT: Open-source toolkit for neural machine translation," 2017, *arXiv:1701.02810*.
- [42] J.-R. Falleri, F. Morandat, X. Blanc, M. Martinez, and M. Monperrus, "Fine-grained and accurate source code differencing," in *Proc. 29th ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2014, pp. 313–324.
- [43] D. R. Reddy *et al.*, "Speech understanding systems: A summary of results of the five-year research effort," Dept. Compu. Sci., Carnegie-Mell University, Pittsburgh, PA, vol. 17, 1977.
- [44] R. Just, D. Jalali, and M. D. Ernst, "Defects4J: A database of existing faults to enable controlled testing studies for Java programs," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 437–440.
- [45] Gerrit code review database, 2018. Accessed: Aug. 18, 2018. [Online]. Available: <https://www.gerritcodereview.com/>
- [46] X. Hu, G. Li, X. Xia, D. Lo, and Z. Jin, "Deep code comment generation," in *Proc. IEEE/ACM 26th Int. Conf. Program Comprehension*, 2018, pp. 200–20 010.
- [47] S. Kommrusch, T. Barollet, and L.-N. Pouchet, "Equivalence of dataflow graphs via rewrite rules using a graph-to-sequence neural model," 2020, *arXiv: 2002.06799*.
- [48] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, and C. V. Lopes, "SourcererCC: Scaling code clone detection to big-code," in *Proc. IEEE/ACM 38th Int. Conf. Softw. Eng.*, 2016, pp. 1157–1168.
- [49] U. Alon, S. Brody, O. Levy, and E. Yahav, "code2seq: Generating sequences from structured representations of code," 2018, *arXiv: 1808.01400*.
- [50] W. J. Masek and M. S. Paterson, "A faster algorithm computing string edit distances," *J. Comput. Syst. Sci.*, vol. 20, no. 1, pp. 18–31, 1980.
- [51] R. Abreu, P. Zoetewij, and A. J. Van Gemund, "On the accuracy of spectrum-based fault localization," in *Proc. Testing: Academic Ind. Conf. Pract. Res. Techn.-Mutation*, 2007, pp. 89–98.
- [52] R. K. Saha, Y. Lyu, H. Yoshida, and M. R. Prasad, "ELIXIR: Effective object-oriented program repair," in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 648–659.
- [53] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 298–309.
- [54] Z. Qi, F. Long, S. Achour, and M. Rinard, "An analysis of patch plausibility and correctness for generate-and-validate patch generation systems," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 24–36.
- [55] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 802–811.
- [56] S. Saha, R. K. Saha, and M. R. Prasad, "Harnessing evolution for multi-hunk program repair," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 13–24.
- [57] B. Ray, V. Hellendoorn, S. Godhane, Z. Tu, A. Bacchelli, and P. Devanbu, "On the naturalness of buggy code," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 428–439.
- [58] S. Wang, D. Chollak, D. Movshovitz-Attias, and L. Tan, "Bugram: Bug detection with n-gram language models," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 708–719.
- [59] M. Allamanis, M. Brockschmidt, and M. Khademi, "Learning to represent programs with graphs," 2017, *arXiv: 1711.00740*.
- [60] M. White, M. Tufano, C. Vendome, and D. Poshyvanyk, "Deep learning code fragments for code clone detection," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 87–98.
- [61] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 933–944.
- [62] P. Yin and G. Neubig, "A syntactic neural model for general-purpose code generation," in *Proc. 55th Annu. Meeting Assoc. Comput. Linguistics*, 2017, vol. 1, pp. 440–450.
- [63] C. Maddison and D. Tarlow, "Structured generative models of natural source code," in *Proc. Int. Conf. Mach. Learn.*, 2014, pp. 649–657.
- [64] A. T. Nguyen and T. N. Nguyen, "Graph-based statistical language model for code," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, 2015, vol. 1, pp. 858–868.
- [65] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, 2019, Art. no. 40.
- [66] S. Karaivanov, V. Raychev, and M. Vechev, "Phrase-based statistical translation of programming languages," in *Proc. ACM Int. Symp. New Ideas New Paradigms Reflections Program. Softw.*, 2014, pp. 173–184.
- [67] A. T. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Divide-and-conquer approach for multi-phase statistical migration for source code," in *Proc. 30th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2015, pp. 585–596.
- [68] A. T. Nguyen, H. A. Nguyen, T. T. Nguyen, and T. N. Nguyen, "Statistical learning approach for mining API usage mappings for code migration," in *Proc. 29th ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2014, pp. 457–468.
- [69] X. Chen, C. Liu, and D. Song, "Tree-to-tree neural networks for program translation," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 2547–2557.
- [70] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "DeepFix: Fixing common C language errors by deep learning," in *Proc. 31st AAAI Conf. Artif. Intell.*, 2017, pp. 1345–1351.
- [71] B. Wei, G. Li, X. Xia, Z. Fu, and Z. Jin, "Code generation as a dual task of code summarization," in *Proc. Int. Conf. Neural Inf. Process. Syst.*, 2019, pp. 6563–6573.
- [72] W. U. Ahmad, S. Chakraborty, B. Ray, and K.-W. Chang, "A transformer-based approach for source code summarization," in *Proc. 58th Annu. Meeting Assoc. Comput. Linguistics*, 2020, pp. 4998–5007.
- [73] S. Xu, Y. Yao, F. Xu, T. Gu, H. Tong, and J. Lu, "Commit message generation for source code changes," in *Proc. 28th Int. Joint Conf. Artif. Intell.*, 2019, pp. 3975–3981.
- [74] Z. Liu, X. Xia, A. E. Hassan, D. Lo, Z. Xing, and X. Wang, "Neural-machine-translation-based commit message generation: How far are we?" in *Proc. 33rd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2018, pp. 373–384.

- [75] I. Polosukhin and A. Skidanov, "Neural program search: Solving programming tasks from description and examples," 2018, *arXiv: 1802.04335*.
- [76] M. Allamanis and C. Sutton, "Mining idioms from source code," in *Proc. 22nd ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2014, pp. 472–483.
- [77] M. Allamanis, E. T. Barr, C. Bird, P. Devanbu, M. Marron, and C. Sutton, "Mining semantic loop idioms," *IEEE Trans. Softw. Eng.*, vol. 44, no. 7, pp. 651–668, Jul. 2018.
- [78] M. Brockschmidt, M. Allamanis, A. L. Gaunt, and O. Polozov, "Generative code modeling with graphs," 2018, *arXiv: 1805.08490*.
- [79] P. Yin, G. Neubig, M. Allamanis, M. Brockschmidt, and A. L. Gaunt, "Learning to represent edits," 2018, *arXiv: 1810.13337*.
- [80] D. Tarlow *et al.*, "Learning to fix build errors with graph2diff neural networks," 2019, *arXiv: 1911.01205*.
- [81] E. Dinella, H. Dai, Z. Li, M. Naik, L. Song, and K. Wang, "Hoppity: Learning graph transformations to detect and fix bugs in programs," in *Proc. Int. Conf. Learn. Representations*, 2019. [Online]. Available: <https://openreview.net/pdf?id=SJeqs6EFvB>
- [82] S. Kaleeswaran, V. Tulsian, A. Kanade, and A. Orso, "MintHint: Automated synthesis of repair hints," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 266–276.
- [83] C. Le Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for \$8 each," in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 3–13.
- [84] P. Liu, O. Tripp, and X. Zhang, "Flint: Fixing linearizability violations," *ACM SIGPLAN Notices*, vol. 49, no. 10, pp. 543–560, 2014.
- [85] F. Logozzo and T. Ball, "Modular and verified automatic program repair," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 133–146, 2012.
- [86] C. Le Goues, T. Nguyen, S. Forrest, and W. Weimer, "GenProg: A generic method for automatic software repair," *IEEE Trans. Softw. Eng.*, vol. 38, no. 1, pp. 54–72, Jan./Feb. 2012.
- [87] F. Long and M. Rinard, "Staged program repair with condition synthesis," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 166–178.
- [88] M. Wen, J. Chen, R. Wu, D. Hao, and S.-C. Cheung, "Context-aware patch generation for better automated program repair," in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 1–11.
- [89] H. D. T. Nguyen, D. Qi, A. Roychoudhury, and S. Chandra, "SemFix: Program repair via semantic analysis," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 772–781.
- [90] S. Mehtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proc. 38th Int. Conf. Softw. Eng.*, 2016, pp. 691–701.
- [91] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *Proc. IEEE Congress Evol. Comput.*, 2008, pp. 162–168.
- [92] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven automated program repair," in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reengineering*, 2016, pp. 213–224.
- [93] M. Boshernitsan, S. L. Graham, and M. A. Hearst, "Aligning development tools with the way programmers think about code changes," in *Proc. SIGCHI Conf. Hum. Factors Comput. Syst.*, 2007, pp. 567–576.
- [94] R. Robbes and M. Lanza, "Example-based program transformation," in *Proc. Int. Conf. Model Driven Eng. Lang. Syst.*, 2008, pp. 174–188.
- [95] Y. Tian and B. Ray, "Automatically diagnosing and repairing error handling bugs in C," in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 752–762.
- [96] B. Hartmann, D. MacDougall, J. Brandt, and S. R. Klemmer, "What would other programmers do: Suggesting solutions to error messages," in *Proc. SIGCHI Conf. Hum. Factors Comput. Syst.*, 2010, pp. 1019–1028.
- [97] V. Murali, L. Qi, S. Chaudhuri, and C. Jermaine, "Neural sketch learning for conditional program generation," 2017, *arXiv: 1703.05698*.
- [98] P. Flener, *Logic Program Synthesis From Incomplete Information*, vol. 295. Berlin, Germany: Springer, 2012.
- [99] T. Lutellier, L. Pang, V. H. Pham, M. Wei, and L. Tan, "ENCORE: Ensemble learning using convolution neural machine translation for automatic program repair," 2019, *arXiv: 1906.08691*.



Saikat Chakraborty received the BSc degree from the Bangladesh University of Engineering and Technology, Bangladesh. He is currently working toward the PhD degree at Columbia University, New York. His research interest is machine learning for reliable softwares. He is interested in combining traditional program analysis techniques with robust machine learning to develop better Bug Detection and Program Repair tool. For more information please visit <http://saikatc.info>.



Yangruibo Ding received the BE degree in software engineering from the University of Electronic Science and Technology of China, China, and the MS degree in computer science from Columbia University, New York. He is currently working toward the PhD degree at Columbia University, New York. His research interest is machine learning (ML) for software engineering, including ML for automated program repair and ML for program analysis.



Miltiadis (Miltos) Allamanis received the PhD degree from the University of Edinburgh, United Kingdom. He is a senior researcher at Microsoft Research, Cambridge, United Kingdom. His research is at the intersection of machine learning, natural language processing and software engineering and researches machine learning methods that use the rich structural aspects of programming languages towards creating better coding tools for end-users and developers. For more information please visit <https://miltos.allamanis.com>.



Baishakhi Ray received the PhD degree from the University of Texas, Austin, Austin, Texas. She is an assistant professor with the Department of Computer Science, Columbia University, New York. Her research interest is in the intersection of Software Engineering and Machine Learning. She has received best paper awards at FASE 2020, FSE 2017, MSR 2017, IEEE Symposium on Security and Privacy (Oakland), 2014. Her research has also been published in CACM Research Highlights and has been widely covered in trade media. She is a recipient of the NSF CAREER Award, VMware Early Career Faculty Award, and IBM Faculty Award.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

Watch Out for Extrinsic Bugs! A Case Study of Their Impact in Just-In-Time Bug Prediction Models on the OpenStack Project

Gema Rodríguez-Pérez^{ID}, Member, IEEE, Meiyappan Nagappan^{ID}, Member, IEEE, and Gregorio Robles^{ID}, Senior Member, IEEE

Abstract—Intrinsic bugs are bugs for which a bug-introducing change can be identified in the version control system of a software. In contrast, extrinsic bugs are caused by external changes to a software, such as errors in external APIs; thereby they do not have an explicit bug-introducing change in the version control system. Although most previous research literature has assumed that all bugs are of *intrinsic* nature, in a previous study, we show that not all bugs are intrinsic. This paper shows an example of how considering extrinsic bugs can affect software engineering research. Specifically, we study the impact of extrinsic bugs in Just-In-Time bug prediction by partially replicating a recent study by McIntosh and Kamei on JIT models. These models are trained using properties of earlier bug-introducing changes. Since extrinsic bugs do not have bug-introducing changes in the version control system, we manually curate McIntosh and Kamei's dataset to distinguish between intrinsic and extrinsic bugs. Then, we address their original research questions, this time removing extrinsic bugs, to study whether bug-introducing changes are a moving target in Just-In-Time bug prediction. Finally, we study whether characteristics of intrinsic and extrinsic bugs are different. Our results show that intrinsic and extrinsic bugs are of different nature. When removing extrinsic bugs the performance is different up to 16 percent Area Under the Curve points. This indicates that our JIT models obtain a more accurate representation of the real world. We conclude that extrinsic bugs negatively impact Just-In-Time models. Furthermore, we offer evidence that extrinsic bugs should be further investigated, as they can significantly impact how software engineers understand bugs.

Index Terms—Bugs, extrinsic bugs, intrinsic bugs, mislabeled bugs, bug-introducing changes, just-in-time, bug prediction

1 INTRODUCTION

RECENT studies show that bugs do not have the same origin [1], [2]. While some bugs have their origin in explicit changes recorded in the version control system (VCS) of the software, other bugs are due to external changes that are not recorded in the VCS, e.g., changes in external APIs, compatibility changes or even changes in the specifications.

Rodríguez-Pérez *et al.* distinguish between *intrinsic bugs* and *extrinsic bugs*. *Intrinsic bugs* are those bugs that have an explicit *bug-introducing change* (BIC) in the VCS. On the other hand, *extrinsic bugs* do not have a BIC recorded in the VCS because there is no explicit change in the VCS of the project that introduced the bug [2]. This may be because the bug was caused (i) by a change in the environment where the software is used, (ii) because requirements changed, (iii) in an external library used by the project, or (iv) by an external change to the VCS of the project, among other reasons.

In the case of extrinsic bugs, it is not possible to identify a BIC in the VCS for a given bug; therefore we cannot link the *bug-fixing change* (BFC) to a BIC. This finding can put in jeopardy the results of previous studies as software engineering research has always considered all bugs to be intrinsic. For instance, *Just-In-Time* (JIT) bug prediction models [3], which are built at change-level, can be affected as they are built with the assumption that for each bug there is always a BFC and a BIC in the VCS [4], [5], [6].

Researchers have proposed different bug prediction techniques [7], [8], [9], [10], [11]. But, JIT bug prediction has many advantages over other bug predictions techniques [12]. For example, JIT models allow developers to review risky changes at the time of being produced and they are built at a finer-granularity as changes are often smaller than modules.

To build JIT models and predict bugs before they are discovered in a software component, it is necessary to train these models using historical data of that software component and learn when a bug occurred in the past. During the training phase, JIT models use datasets that connect bug reports with the code changes that fixed the bug (the BFC), and with previous code changes that introduced the bug in the software (the BIC).

Then, to predict future bugs, JIT models use code change properties of BICs and BFCs, such as the size of the change, the number of files modified by the change, or the experience of the developer. Since extrinsic bugs cannot be linked to a BIC, we hypothesize that in JIT models the incorrect

• Gema Rodriguez-Perez and Meiyappan Nagappan are with the University of Waterloo, Waterloo, ON N2L 3G1, Canada.
E-mail: {gema.rodriguez-perez, mei.nagappan}@uwaterloo.ca.

• Gregorio Robles is with the Universidad Rey Juan Carlos, Fuenlabrada, Madrid, Spain. E-mail: grex@gsyc.urjc.es.

Manuscript received 10 Dec. 2019; revised 22 July 2020; accepted 31 Aug. 2020.
Date of publication 2 Sept. 2020; date of current version 18 Apr. 2022.
(Corresponding author: Gema Rodriguez-Pérez.)
Recommended for acceptance by G. Canfora.
Digital Object Identifier no. 10.1109/TSE.2020.3021380

identification of BICs in extrinsic bugs may impact the quality of datasets used to train JIT models, and ultimately may impact JIT models themselves.

JIT models must be trained using reliable datasets to improve their performance and increase their trustworthiness [11], [13]. To do so, we need to identify extrinsic bugs and remove them from the dataset. Therefore, we can obtain reliable dataset that only contains intrinsic bugs, i.e., bugs for which we can identify a BIC.

To study the impact of extrinsic bugs in JIT models, we partially replicated a recent paper by McIntosh and Kamei that analyzed the performance of JIT models [14]. Through this paper, we will refer to McIntosh and Kamei's paper as *Mc&K's paper* to improve readability. As in previous research, Mc&K's paper considered all bugs to be intrinsic. To quantify the impact of extrinsic bugs on JIT models, we removed extrinsic bugs from their dataset.

Methodologically, to classify bugs as intrinsic or extrinsic, we followed the approach proposed in [1], [2], which requires manually analyzing the bugs and their textual information. As this is a very labor-intensive task, we have focused only on one of the projects used as case study in Mc&K's paper: OpenStack. We first manually curated their dataset and classified 1,880 bugs as intrinsic or extrinsic. We then used this curated dataset to train JIT models removing extrinsic bugs and computed their performance when identifying future BICs. Finally, we compared Mc&K's results with our results and deepened in the differences regarding intrinsic and extrinsic bugs.

Thus, we analyze whether our manually curated dataset is different from Mc&K's dataset (RQ1). We have therefore added a constraint (i.e., "when extrinsic bugs are removed") to Mc&K original research questions to study the impact of extrinsic bugs in JIT models, (RQ2-RQ4). As we found a significant share of mislabeled bugs in Mc&K's dataset, we also analyze what their impact is as well (RQ5). Mislabeled bugs refer to issue reports that have been considered as bug reports when, in fact, they are not reporting a bug but another software maintenance activities, e.g., enhancements or refactoring. Finally, we study if intrinsic, extrinsic, and mislabeled bugs have different characteristics (RQ6).

Our results indicate that (1) intrinsic and extrinsic bugs are different, (2) our manually curated dataset differs, in terms of number of bugs, over 40 percent from an automatic extracted dataset, (3) JIT models obtain different performance in terms of Area Under the Curve (AUC) (up to 16 percent AUC points) when they consider only intrinsic bugs, and (4) AUC scores are more stable after removing extrinsic bugs.

The remainder of this paper is organized as follows. Section 2 presents the research questions. Section 3 discusses related work. Section 4 describes the design of our case study, and Section 5 presents how the model is constructed and analyzed. Section 6 presents the results. Section 7 discusses the findings, while Section 8 contains the threats to their validity. Finally, Section 9 draws conclusions.

2 RESEARCH QUESTIONS

The research questions addressed in this work are:

- RQ1: How does our manually curated dataset differ from the one by McIntosh and Kamei?

Motivation: JIT models should use as input intrinsic bugs, as BICs of extrinsic bugs cannot be identified in the VCS. Thus, we are interested in studying how different our manually curated dataset is compared to the dataset obtained automatically and used in McIntosh and Kamei.

Results: Over 40 percent of bugs could not be linked to a BIC: 11.3 percent of the bugs in McIntosh and Kamei's dataset were classified as extrinsic bugs and 29.1 percent as mislabeled issues.

- RQ2: Do JIT models lose predictive power over time when extrinsic bugs are removed?

Motivation: McIntosh and Kamei found that JIT models that were trained with old source code properties of BICs lose predictive power. With this question we want to see how only considering intrinsic bugs affects the predictive power of the models.

Results: JIT models also lose predictive power after one year of being trained when only intrinsic bugs are considered. However, our JIT models obtained a different performance in terms of AUC values (up to 16 percent AUC points) and a minor loss of predictive power for each period (up to 15 percent AUC points).

- RQ3: How does the relationship between code change properties and the likelihood of BICs evolve in terms of time when extrinsic bugs are removed?

Motivation: If code change properties¹ of BICs change over time, the properties of prior and future BICs are different. McIntosh and Kamei studied this relationship and found that properties of BICs change through the evolution of project. However, as the dataset they used contained extrinsic bugs as well, we think that prior and future events may not have similar properties. Thereby the impact of code change properties might fluctuate.

Results: We have found that the impact of code change properties is indeed different than the one reported in McIntosh and Kamei. When extrinsic bugs are removed, the code change properties related to the magnitude of the change (*Size*) increase up to 18 percent AUC points and the code changes properties related to the code review process (*Review*) decrease up to 36 percent AUC points.

- RQ4: How accurately do current importance scores of code change properties represent future ones when extrinsic bugs are removed?

Motivation: McIntosh and Kamei found that the importance scores for some of the most impactful code change properties are consistently under/overestimated. However, we think that the importance score of some properties might change over time when removing extrinsic bugs.

Results: We found that the importance scores for some of the most impactful code change properties are consistently under/overestimated as well. However, the stability of property importance score remains similar in both short and long JIT period

¹ Code change properties are described in Table 3.

models – in McIntosh and Kamei this only applies to short-term models.

- RQ5: How do mislabeled bugs affect JIT models?

Motivation: While manually curating the dataset in McIntosh and Kamei, we found a considerable share of mislabeled bugs. In RQ2-RQ4, we considered them as part of the input data for the JIT models. With this RQ we want to quantify how they affect the JIT models. As reported in recent research literature [15], we expect mislabeled bugs to have a low effect on the results.

Results: Contrary to our expectations, we have found that mislabeled bugs also impact JIT models. When comparing the results of including mislabeled and intrinsic bugs in the dataset that fed JIT models with the best results that these models can obtain, we saw that they lose up to 4 percent AUC points.

- RQ6: Are the properties of BFCs and BICs linked to extrinsic, intrinsic, and mislabeled bugs different?

Motivation: The results from RQ2-RQ5 show an improvement in terms of AUC in JIT models when removing extrinsic bugs. To ensure that these results can be considered statistically significant, we need to analyze how different code change properties of BFCs and BICs linked to extrinsic and intrinsic bugs are.

Results: Intrinsic and Extrinsic bugs present statistically significant differences in the code change properties of BFCs and BICs linked to them. Furthermore, these properties are more similar between extrinsic and mislabeled than between intrinsic and mislabeled bugs.

3 RELATED WORK

In this section, we contextualize our work with past studies on bug origins, JIT bug prediction models and mislabeling issues.

3.1 Origin of Bugs

JIT bug prediction models need to identify *bug-fixing changes* (BFCs) and *bug-introducing changes* (BICs) from historical data, and then use the code change properties of those BFCs and BICs to train the JIT bug prediction models. That way JIT models can point out buggy changes before they are discovered in the software.

Traditionally, JIT bug prediction models use the algorithm proposed by Sliwersky, Zimmermann, and Zeller (SZZ) [4] to identify past BFCs and BICs. SZZ is a popular algorithm in bug prediction [16]. It assumes the last change that *touched* the line(s) fixed in a BFC introduced the bug [4], [5], [6], [17]. In short, SZZ is an algorithm that links the VCS and the issue tracking (ITS) system of a project to identify the BFCs and their associated BICs.

Some authors have highlighted the limitations of linking BFCs with BICs, since the origin of some bugs might not be related to the lines modified in the BFC that fix the bug. German *et al.* investigated bugs that manifest themselves in unchanged parts of the software and their impact across the whole system [18]. Chen *et al.* studied the impact of dormant bugs (i.e., bugs that were introduced in a version of the software system, but they were not found until much

later) on bug localization [19]. Prechelt and Pepper observed that BFCs may touch non-buggy lines, and even when they touched those lines, the actual BIC may have been made earlier [20]. Ahluwalia *et al.* investigated the extend to which defect datasets ignore some defects because they have not been fixed [21].

Recently, Rodríguez-Pérez *et al.* have analyzed in detail the origin of bugs and found that some BICs cannot be identified in the VCS of a project because the change that caused the bug was not recorded in the VCS. The authors identified two types of bugs: (1) intrinsic bugs, i.e., bugs caused by explicit changes recorded in the VCS, and (2) extrinsic bugs, i.e., bugs caused by external factors or changes to the software, as for instance changes in an external API, or changes in the requirements [1], [2].

3.2 Just In Time Bug Prediction Models

JIT bug prediction models identify risky software changes instead of risky files or packages. Kamei *et al.* proposed for the first time the JIT quality assurance technique that predicts defects at change-level [3]. Recent studies have demonstrated that JIT models obtain sufficient prediction accuracy to be applied in practice [22], [23].

JIT bug prediction models assume that code change properties of past BICs are similar to code properties of future BICs. Therefore, we can use JIT models to learn from the past and predict the future. To achieve good prediction accuracy in JIT models, researchers rely on a variety of code changes properties to predict future BICs. These properties can be derived from the changes themselves [13], [24], from VCSs and ITSs [3], [25], [26], or from code review systems [27], [28].

These properties have been used in previous studies [3], [26], [28] and can be grouped into six families of code change properties according to McIntosh and Kamei [14]: (i) the *Size* family measures the magnitude of the change, (ii) the *Diffusion* family measures the dispersion of the changes across each modified file, (iii) the *History* family measures the bug proneness of prior changes to the modified files, (iv) the *Author Experience* family measures the experience of the author of the change, (v) the *Reviewer Experience* family measures the experience of the code reviewer(s) of the change, and (vi) the *Review* family measures characteristics of change in the code review process.

We decided to study JIT bug prediction models because (1) they present many advantages over other bug prediction techniques [12], (2) they perform with high prediction accuracy [22], and (3) they are a more practical alternative to traditional bug prediction techniques [29]. Nowadays, JIT bug prediction models are the best models yielding actionable results in the current state of the art.

3.3 Mislabeling Issues

As far as we know, there are two approaches to distinguish mislabeled bugs from real bugs. The first one is a manual analysis using the classification rules proposed by Herzig *et al.* [30]. The second one is an automatic approach that uses regular expressions to identify (real) bugs from the commit messages of the BFCs. The SZZ algorithm implements this approach, so it has been widely used in previous research [5], [14], [31].

Although the automatic approach can be quicker and easier than the manual analysis, it may lead to noise in the dataset as some issue reports can be mislabeled, i.e., issue reports that describe defects but were not classified as such (or vice versa). Previous studies have shown the importance of correctly collecting data from VCS. Aranda and Venolia found that VCS and ITS hold incomplete or incorrect data [32], which cause mislabeling data. Some studies have demonstrated that 33.8 [30] to 40 percent [33] of the bugs in the ITS are mislabeled.

This mislabeling might impact the performance of bug prediction models. Kim *et al.* found that bug prediction models are considerably less accurate when they are trained using datasets that have a 20-35 percent mislabeling rate [34]. Herzog *et al.* observed that 33.8 percent of all issue reports were mislabeled, and that this impacted the prioritization of files in bug prediction [30]. Seiffert *et al.* carried out a comprehensive study [35] that confirms Kim *et al.*'s findings [34].

More recent studies suggest that mislabeled issues might not be a severe threat in bug prediction models since the mislabeling may not be necessarily random. For example, it is more likely that a novice developer mislabels an issue than an experienced developer. Tantithamthavorn *et al.* found that precision is rarely impacted by mislabeled issues but recall is often impacted [15]. They claim that the differences with Herzog *et al.* [30] may be explained by the differences in their defect prediction experiments. Rahman *et al.* found that the number of buggy modules has a higher impact on bug model performance than the mislabeling data [36]. However, both studies agree that cleaning the data before training the models allows to achieve a better identification of indeed buggy modules.

Thus, to shed some light on this topic, we decided to study how mislabeled bugs impact JIT bug prediction models. While previous studies looked at mislabeling in bug prediction models at the *file or module* level, our study focuses on the *change level*.

As far as we know, all previous studies about mislabeling have not differentiated between extrinsic and intrinsic bugs, considering them together. Therefore, there is no overlapping between what they considered as mislabeled bugs and what we refer to as extrinsic bugs in this work.

4 CASE STUDY AND METHOD

In this section, we describe our rationale for selecting the studied system and the data extraction process.

4.1 Studied System: OpenStack

A qualitative analysis is required to ensure the correct identification of extrinsic bugs. The output of this analysis is a manually curated dataset. Creating this dataset is very labor intensive, since for every issue it is necessary to understand either the textual information in the issue report and the source code in the bug-fixing change, if not both. Given this considerable effort, we selected one of the two case studies from Mc&K's paper to partially replicate their study and understand the impact that extrinsic bugs have on JIT bug prediction models.

We chose OpenStack because we are more familiar with OpenStack –in our previous study [1] we investigate Nova,

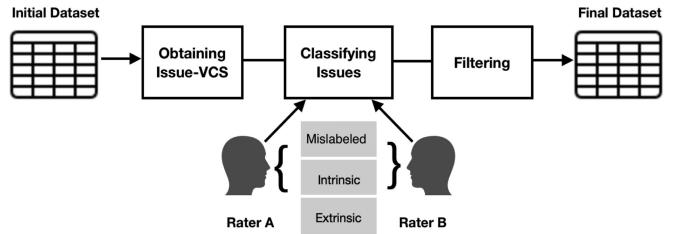


Fig. 1. Overview of the steps followed to curate the dataset.

a component of OpenStack – than with Qt. Furthermore, we believe that OpenStack is an interesting and worthwhile project to study the impact of extrinsic bugs in JIT bug prediction models because it has more than 10,300 contributors with significant industrial support from several major IT companies such as Red Hat, Huawei, and IBM. Currently, OpenStack has more than 330K commits with more than 48M lines of code and around 8,400 active developers.² All its history is available and saved in a VCS (git), an ITS (Launchpad³), and a source code review system (Gerrit⁴).

4.2 Data Extraction

To study the impact of extrinsic bugs on JIT bug prediction models, we used the replication package⁵ provided by Mc&K's paper [14]. With the information of the issues in the ITS and the VCS, we were able to manually annotated bugs on whether they were intrinsic or extrinsic. We also found many issues that were wrongly considered bugs.

Fig. 1 provides an overview of the phases followed to obtain our final dataset. In the remainder of this section, we describe each phase in detail.

4.2.1 Obtaining the Issue-VCS Dataset

The replication package only provides the final dataset to feed the JIT models studied in Mc&K's paper. We identified which of these changes were related to intrinsic or extrinsic bugs, and then removed the property of being a BIC when the change was an extrinsic bug. For that, we required access to the issues in the ITS and their links to the changes in the VCS. To ensure that we could address our research questions with the same dataset as Mc&K, we asked them for the Issue-VCS dataset and obtained it.

The Issue-VCS dataset contains unique identifiers (*issueIDs*) of the issues and the timestamp when they were reported. The *issueIDs* were used to link issues to code changes (*changeIDs*). Thus, for each *issueID* there is one BFC and one or more *changeIDs* flagged as BICs. In total, the Issue-VCS dataset contains 1,880 *issueIDs* linked to 1,904 *changeIDs* identified as BFCs, and 3,486 *changeIDs* identified as possible BICs. Note that *issueIDs* and BFCs do not have to be related one-to-one, since an *issueID* can be fixed by more than one *changeID*.

4.2.2 Classifying Issues

During the manual analysis, we noticed that the data used by Mc&K not only contained extrinsic bugs, but issues that

2. <http://stackalytics.com>

3. <https://launchpad.net/openstack>

4. <https://review.openstack.org/>

5. <https://github.com/software-rebels/JITMovingTarget>

TABLE 1
First Step: Classification Rules for Classifying Issues as Bug Report or *not a Bug Report (Mislabeled)*

An issue is classified as not a bug report (<i>Mislabeled</i>) if ...
(1) It reports a bug in test files. We assume that these bugs are caused by how developers understand and test the code.
Thus, there is no change introducing buggy code to source code of the project.
(2) It reports a clean up in the source code that does not interfere with the performance of the software.
(3) It reports a misspelling or typo in the inline comments.
(4) It reports a change in the source code to prevent future bugs.
(5) The report has discordance in the comments between developers.
(6) The report does not have a BFC.

An issue is classified as bug report if ...
(1) It reports a misspelling or typo in the source code.
(2) It reports that a previous change to the source code caused the bug.
(3) It reports a buggy functionality implemented that should be known at the time of coding.
(4) It reports an omission in the original code that should be considered at the time of coding.

in fact were not bugs (e.g., other kind of issues such as request for new features or maintenance activities [30]).

Thus, first, following the guidelines provided by Herzig *et al.* [30], we manually classified the 1,880 issueIDs into *bug report*, or *not a bug report* (i.e., *mislabeled*). Table 1 shows the rules used in this first step. Then, we manually classified the issues identified as bug reports as *intrinsic* or *extrinsic* bugs. For doing so, we used the approach by Rodríguez-Pérez [1]. Table 2 offers the specific rules used in this second step.

To remove subjectivity and bias in the classification, two raters having at least a master's degree in Computer Science manually classified the issueIDs. The raters were individually trained in different stages, in each of them analyzing 100 random issueIDs from the data until they reached a near perfect agreement (0.81 - 1). The ratio agreement between both raters was computed using Krippendorff's alpha, and the disagreements were resolved with online meetings. After each stage, the raters discussed the discordance and added additional rationale to the guidelines.

Once the raters reached a near perfect agreement, they individually analyzed 25 percent (470) of the issueIDs. At this point, the raters obtained a Krippendorff's alpha of 0.974 classifying issueIDs as a bug or not a bug, and a Krippendorff's alpha of 0.823 classifying bugs reports as extrinsic or intrinsic. We considered that the raters' agreement was high enough to analyze the remaining 1,410 issueIDs only by one rater (i.e., each rater classified 705 of the issueIDs).

The result of the classification procedure is a dataset where issues are labeled as (1) intrinsic bug, (2) extrinsic bug, or (3) not a bug (*mislabeled*).

4.2.3 Characteristics of the Changes

Mc&K extracted several code and review properties for each change from the VCS of OpenStack. The properties were grouped in six families: *Size*, *Diffusion*, *History*, *Author Experience*, *Reviewer Experience*, and *Review*. We use this information "as is". The complete list of properties can be found in Table 3.

TABLE 2
Classification Rules for Classifying Bug Reports as Intrinsic or Extrinsic

A bug report is classified as extrinsic if ...
(1) It reports a bug caused by a change in the environment where the software is used.
(2) It reports a bug because requirements have changed.
(3) It reports a bug caused by an external change to the VCS of the project.
(4) It reports a bug in an external library used by the project.

A bug report is classified as intrinsic if ...
(1) There is no evidence to be classified as an extrinsic bug.

4.2.4 Final Dataset

To obtain the final dataset that fed the JIT bug prediction models, Mc&K merged the Issue-VCS dataset using changeID and issueID. This merging filtered the dataset and mitigated false positives. In addition, the dataset provided by Mc&K (1) ignores potential BICs that only updated code comments or white spaces (an improvement to SZZ by Kim *et al.* [5]); (2) filters out potential BICs that appear after the date that the implicated bug was reported [4]; and (3) ignores suspicious BFCs and suspicious BICs using the framework proposed by da Costa *et al.* [31].

Our goal is to study the impact of extrinsic bugs. Thus, we removed the changeIDs that were not BICs. Since extrinsic bugs do not have BICs, we removed the link between issueIDs classified as extrinsic bugs and their BICs following the recommendation of Rodríguez-Pérez *et al.* [2]. Besides, we ignored changes that modified either at least 10,000 lines ("too much churn") or 100 files ("too many files") as they were likely no BICs. The dataset obtained at this point is what we have called the *final dataset*. Finally, to study whether properties of BICs are consistent, we stratified the final dataset into periods of three and six months as Mc&K's paper recommend [14]. Table 4 shows the number of issues and BICs after each filtering phase.

Furthermore, in RQ5 (see Section 6.5) we discuss what the impact of removing mislabeled issues is. Thus, we added an additional filter to remove the links between issueIDs that were classified as mislabeled with their changeIDs identified as BFCs and BICs.

5 MODEL CONSTRUCTION AND ANALYSIS

In this section, we describe the model construction and analysis approach. Since we were partially replicating Mc&K's paper, we exactly followed their model construction procedure. Thus, we used Mc&K's design decisions with our final dataset, i.e., we did not modify any design decision from Mc&K for the construction or analysis of the model.

5.1 Model Construction

5.1.1 Handling Collinear Properties

Before constructing JIT models, we removed collinear code change properties to avoid distorting the modeled relationship between these code change properties and the likelihood of introducing bugs.

We used the Spearman rank correlation tests ρ to remove code change properties that were highly correlated with one

TABLE 3
Taxonomy of Changes Provided by McIntosh and Kamei [14]

	Property	Description	Acron.
Size	Lines added	Number of lines added by the change.	la
	Lines deleted	Number of lines deleted by the change.	ld
Diff.	Subsystems	Number of modified subsystems.	ns
	Directories	Number of modified directories.	nd
	Files Entropy	Number of modified files. Spread of modified lines across files.	nf ent
Hist.	Unique Developers	Number of prior changes to the modified files. Number of developers who have modified the file in the past.	nuc ndev
	Age	Time interval between the last and the current changes.	age
	Prior Changes	The number of prior changes that an actor ⁶ has participated ⁷ in.	aexp
Author/ Reviewer Exp.	Recent Changes	The number of <i>aexp</i> weighted by the age of the changes.	arexp
	Subs.Changes	Number of prior changes to the <i>ns</i> that an actor has participated in.	asexp
	Awareness	Proportion of <i>aexp</i> to <i>ns</i> that an actor has participated in.	asawr
Review	Iterations	Number of times that a change was revised before integration.	nrev
	Reviewers	Number of reviewers who have voted on integrating a change.	app
	Comments	Number of non-automated, non-owner comments during the review of a change.	hcmt
	Review Window	Time length between creation of a request and its final approval for integration.	rtime

another. For code change properties with correlation $|\rho| > 0.7$, we only included one of the properties in the models.

Then, we fit preliminary models that explain each property using the others to remove redundant code change properties. For that purpose, we used the *redund* function available in the *rms* R package.

5.1.2 Fitting Regression Model

Software Engineering researchers often use a nonlinear variant of multiple regression modeling to understand the relationship between software quality and software development practices [37], [38]. We fit JIT models using this technique as it relaxes the assumption of a linear relationship between the likelihood of introducing bugs and the code change properties; thus we can achieve a more accurate fit of the data. We used restricted cubic splines, which

6. Either the author or reviewer of a change.

7. Either authored or reviewed.

TABLE 4
Number of Unique Issues and Unique BICs That Survive Each Step of the Filtering Process

#	Filter	Issues	BICs
F_0	Issue-VCS dataset	1,880	3,486
F_1	Extrinsic Bugs	1,668	2,925
F_2	Too much Churn	1,668	2,920
F_3	To many Files	1,668	2,911
F_4	No lines added	1,668	2,907
F_5	Period	1,668	2,506

fit smooth transitions at the points where curves change in direction, to fit our curves.

5.2 Model Analysis

To answer our research questions, we analyzed the output of the JIT models using the different datasets.

5.2.1 Analyzing the Performance of the Models (RQ2)

The performance of JIT prediction models was assessed using two metrics: the Area Under Curve (*AUC*) and the Brier score.

The *AUC* is an evaluation metric for assessing the discriminatory power of a model, i.e., in our case its ability to differentiate between a BIC and not a BIC. *AUC* is calculated by measuring the area under the curve that plots the true positive rate of BICs against the false positive rate of BICs. Its values range from 0 to 1; thus, the higher the *AUC*, the better the model is at predicting a BIC or not a BIC. When *AUC* is approximately 0.5, the model has no discrimination capacity, and it performs as random guessing.

The Brier score measures the calibration of the model. It is computed by measuring the mean squared difference between the predicted probability assigned to the possible outcomes (being a BIC or not) for a change and its actual outcome. The Brier score can range from 0 to 1; 0 indicates a perfect calibrated model, while 1 indicates the worst possible calibration for a model.

5.2.2 Analyzing Property Importance (RQ3)

Each of the six change properties families is comprised of several properties, and each property has been allocated with three degrees of freedom. A model term represents each degree of freedom. Thus, to estimate the impact that each family has on the explanatory power of the JIT models we jointly tested the set of model terms for each family using the Wald χ^2 maximum likelihood tests [38]. We normalized the Wald χ^2 values by the total Wald χ^2 score of the JIT model to compare multiple models. The larger the normalized Wald χ^2 score, the more significant the impact a particular family of code change properties has on the explanatory power of our JIT models.

5.2.3 Analyzing Property Stability (RQ4)

To evaluate the stability of the importance scores for each family of code change properties f over time, we calculated the difference between the importance scores of f in a model that is trained using a period n and a future model that is trained using a period $n + \chi$ where $\chi > 0$.

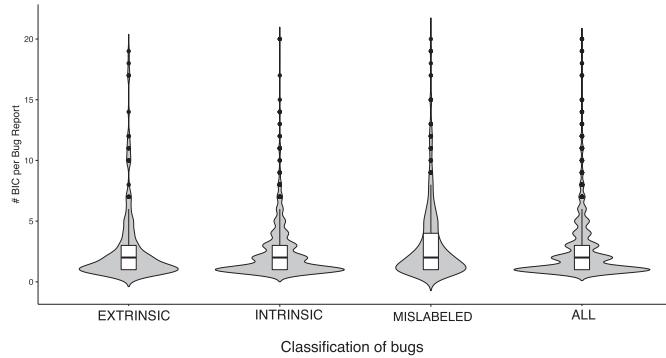


Fig. 2. Distributions of the commits identified as BICs per bug report for each category of bug.

6 RESULTS

6.1 RQ1: How Does Our Manually Curated Dataset Differ From the One by McIntosh and Kamei?

Approach. Our manually curated dataset distinguishes among intrinsic bugs, extrinsic bugs, and mislabeled bugs, but Mc&K's dataset does not. Thus, to further understand the differences between our manually curated dataset and an automatically extracted dataset (Mc&K's paper), we computed the distributions and the probability density of the 1,880 issues in the two datasets. We used a kernel plot to present the distribution shape of the datasets. In kernel plots, wider sections represent a higher probability that members of the population will take on the given value; skinnier sections represent a lower probability.

Results. While manually curating the dataset, we identified 1,120 intrinsic bugs, 212 (11.3 percent) extrinsic bugs, and 548 (29.1 percent) mislabeled bugs. We found that in Mc&K's dataset, there are 1,413 BFCs linked to extrinsic bugs, 3,690 BFCs linked to intrinsic bugs, and 3,147 BFCs linked to mislabeled bugs.

Fig. 2 shows a violin plot with the distribution of the number of commits identified as BICs for each category. This figure offers evidence that (1) extrinsic, intrinsic, and mislabeled bugs have different distribution shapes; (2) Mc&K's bugs (All) and intrinsic bugs have similar distribution shapes; and (3) the distribution shapes of extrinsic and mislabeled bugs differ from the one of intrinsic bugs.

Answer to RQ1: Over 40 percent of the McIntosh and Kamei [14]'s dataset are not intrinsic bugs. Extrinsic and mislabeled bugs show different distribution shapes than intrinsic bugs.

6.2 RQ2: Do JIT Models Lose Predictive Power Over Time When Extrinsic Bugs are Removed?

To study how quick JIT models lose their predictive power we follow the same methodology as McIntosh and Kamei [14]. We split the data into periods, i.e., three-month and six-month periods of data. Then, we train the JIT models for each period and measure their performance on future periods.

Approach. Since older changes may have different characteristics than more recent ones, we used a short period model to train each period. Short period models are JIT models trained only using changes that occurred during

one time period, the latest one before the test period. Since some studies suggest that the more training data, the better the results in bug detection models [36], [39], we also used long period models to train each period. These long period models are JIT models trained using all the changes that occurred during or prior to the test period.

After training our JIT models in short and long periods, we measured their performance when they were applied in the test period. The performance of our JIT models was measured using the AUC and the Brier score, as explained in Section 5.2. For example, for training period 4, the short period model was trained using the changes in this period, and was tested using changes from period 5 onward; while the long period model was trained using changes in periods 1, 2, 3, and 4 and tested using period 5. In both cases, the AUC and Brier measures were computed for each testing period individually.

Results. Fig. 3 shows heat-maps with the trend in AUC and Brier performance scores for each period tested for our short and long period JIT models. The shade of the box indicates the performance value (from 0 to 1): Blue colors stand for strong performance, white colors for random guessing performance, and red colors for weak performance.

The columns of Fig. 3a show that the values tend to improve as the training period increases. For instance, column 4 of the long period model has 0.66 of AUC score when the model was trained using period 1. However, the AUC score is 0.71, an improvement of 5 percent points, when it was trained using period 3. All in all, the long period model in Fig. 3a presents a steady AUC score improvement of 5-9 percent points when we trained the models using the most recent data instead of data from period 1. While the short period model presents a AUC score improvement of 6-10 percent points. The columns in Fig. 3c also show a rise in Brier scores of 1-5 percent points for the long period and 1-4 percent points for the short period. The six-month period models have almost the same performance, Figs. 3b and 3d show AUC and Brier improvements that reach 7-8 and 3-8 percent points for the long period, respectively.

The columns of Fig. 3a show as well an improving trend in AUC scores that is more stable in long than in short period models. For instance, columns 5, 6, and 7 show that the AUC improvement gained by adding the most recent period to the long period is 2 percentage point in column 5 (0.72 and 0.74 for training periods 3 and 4), 1 in column 6 (0.70 and 0.71 for training periods 4 and 5) and 0 in column 7 (0.71 and 0.71 for training periods 5 and 6). While the improvement gained by adding the most recent period to periods 5, 6, and 7 in the short period models is 0, 2 and -1 respectively. Fig. 3b shows a similar tendency for six-month periods. Figs. 3c and 3d indicate that the improving trend in the Brier score is stable in both, short and long period models.

When comparing these results with Mc&K's paper, we noticed a considerable increase in the blue shades, which points out that our models perform stronger in terms of AUC scores. For three-month periods, JIT models without extrinsic bugs improved the AUC score from 1-16 percent AUC points for testing periods 3-9 in the short and long periods. This improvement is for example noticeable in testing periods 3, 4, and 5 with training periods 1 and 2. While Mc&K's models obtain almost the performance of a random

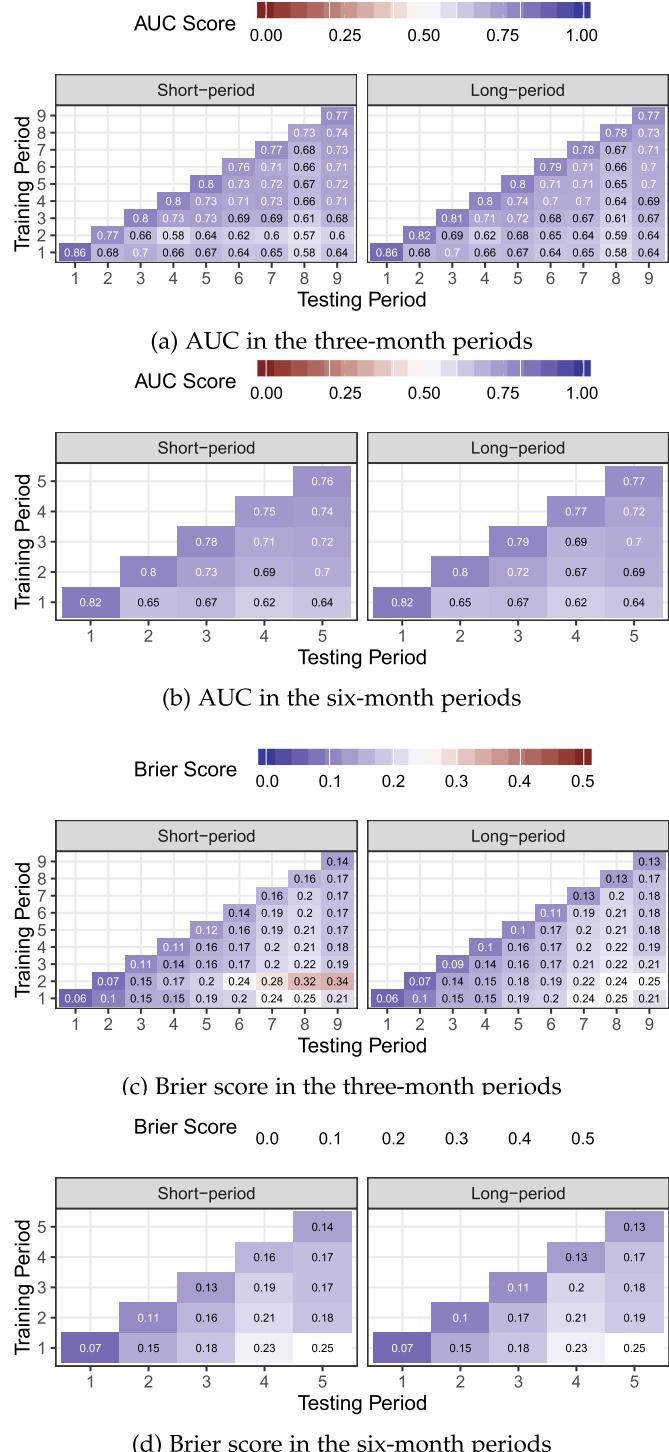


Fig. 3. The predictive performance of JIT models as the studied system age.

guess, our models obtain an AUC score improvement of 6-16 percent points for both short and long periods.

Furthermore, after removing extrinsic bugs, our JIT models increase their stability by reducing two points in the short and long period (after period 5), while Mc&K's models obtain a stability of -5.2 and -1.2 percent points for the short and long period, respectively; our models obtain -2.1 and 0.2 percent points for both periods respectively.

Fig. 4 shows a heat-maps of the difference in AUC and Brier performance scores between training and testing

periods over time of our short and long period JIT models. The shade of the box offers information about the difference: blue colors stand for improvements performance, white colors for unchanged performance, and red colors for drops in performance in the testing period.

The analysis of the rows in Fig. 4 offers evidence that our models lose predictive power after 12 months of being trained. Figs. 4a and 4b show that our short and long period models lose 8-19 and 10-19 percent AUC points 12 months after being trained (i.e., testing period = training period + 4) respectively. Both figures show that after 12 months there is (often) a drop in the AUC. At the same time we can observe a boost in Brier scores (see Figs. 4c and 4d respectively).

Thus, similar to Mc&K's results, we lose predictive power in our JIT models after one year of being trained. However, our models lose less amount of predictive power in each period when using testing periods 1, 2, 3, and 4. Also, AUC scores are more stable after removing extrinsic bugs. For example, Mc&K's models lose 3-34 percent AUC points in short period models after one year, while our models only lose 8-19 percent AUC points. Thus, our models lose 15 percent AUC points less and gained stability up to 20 percent AUC points.

To observe the predictive power of long and short period JIT models, we focus on the data from period 2 and later since the AUC and Brier values of period 1 are identical in both periods. This is because there is no additional data added when training the long period model. The rows of Figs. 4a and 4b show that the short period models of periods 3 and later retain more predictive power than their long period counterparts in terms of AUC, i.e., the drop in the AUC values is smaller since these values are close to 0. Fig. 4a shows that when the long period model is trained using period 3, it drops 10 percent AUC points when it is tested in period 4, while it only drops 7 percent AUC points in the short period model under the same circumstances.

Fig. 4b offers evidence that with six-month periods, both models retain similar predictive power; period 5 drops 8 percent AUC points in both models. Figs. 4c and 4d show that there is also an improvement in the retention of Brier score in short period models. Furthermore, Fig. 4 indicates that short period JIT models retain more predictive power than long periods.

Answer to RQ2: When removing extrinsic bugs, JIT models obtain better performance in terms of AUC (up to 16 percent AUC points). Models that only consider intrinsic bugs also lose predictive power 12 months after being trained, but they lose up to 15 percent AUC points less and are up to 20 percent AUC points more stable.

6.3 RQ3: How Does the Relationship Between Code Change Properties and the Likelihood of BICs Evolve in Terms of Time When Extrinsic Bugs are Removed?

Approach. To answer this question, we followed Mc&K's approach [14] and computed the normalized Wald χ^2 importance score (see Section 5.2) for each family of code change properties, and for short and long period JIT models. Furthermore, we computed the p -values associated with these scores.

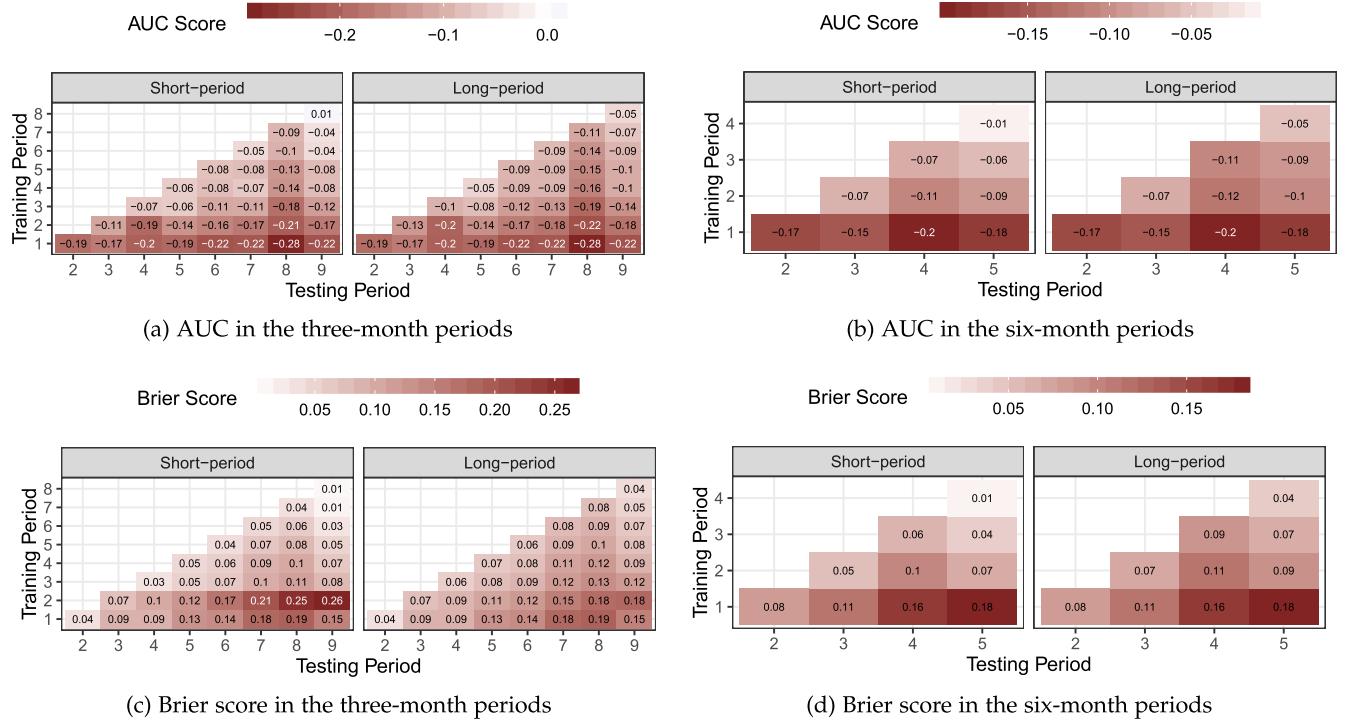


Fig. 4. The delta in the estimate performance of JIT models as the studied system age.

Results. Fig. 5 offers a series of heat-maps with the importance score of the six code change property families. The darker the shade of the box, the more important the family is to our model.

Fig. 5a shows that in both short and long period models of three-month periods, the families of code changes that contribute the most are *Size*, *Diffusion*, and *Review* for the last periods. *Size* accounts for 20-49 percent, *Diffusion* for 0-39 percent, and *Review* for 6-26 percent of the explanatory power in the short period models. In the long period models, *Size* accounts for 21-33 percent, *Diffusion* for 3-18 percent, and *Review* for 10-25 percent of the explanatory power.

The six-month period models present similar results. Fig. 5b shows that the *Size* and *Review* families account for more of the explanatory power in both short and long period models. *Size* accounts for 21-43 and 22-33 percent, and *Review* for 9-24 and 14-21 percent.

For the six-month period models, Fig. 5 shows that the *Size* family is the top contributor in all periods of both short and long period models. For the three-month period models, Fig. 5 shows that, in both short and long period models, the *Size* family is the top contributor in 8 out of 9 periods. The *Review* family is the top contributor in the remaining periods.

The contributed explanatory power of the *Size* family is statistically significant ($\rho < 0.01$, $\rho < 0.001$) in all of the periods for our long and short period models in the three-month and six-month periods. The *Review* family's explanatory power is also statistically significant ($\rho < 0.01$, $\rho < 0.001$) in all of the periods in the long period model of the three-month periods and for both models of the six-month periods. However, in the short period models of the three-month periods, the *Review* family's explanatory power is statistically significant only in 6 out of 9 periods.

Compared to Mc&K's paper, when removing extrinsic bugs in both short period models of three- and six-month

periods, the explanatory power of the *Size* family increases from 3-37 to 20-49 percent, and from 16-25 to 24-43 percent. However, the explanatory power of the *Review* family decreases considerably from 2-59 to 6-26 percent, and from 8-38 to 9-24 percent. In both long period models of three- and six-month periods, when removing extrinsic bugs, the explanatory power of the *Size* family also increases from 11-37 to 21-33 percent, and from 15-19 to 22-33 percent, but the explanatory power of the *Review* family decreases considerably from 15-43 to 10-25 percent, and from 24-37 to 14-21 percent. This may indicate that extrinsic bugs have different characteristics affecting the *Review* family. Moreover, the statistical significance power of the *Review* and *Size* families also increases when removing extrinsic bugs. Our models increase the number of periods with statistical significance of the *Diffusion* family in both long and short periods of the three-months periods and six-months periods.

Furthermore, our models increase the number of the significant periods in *Diffusion* for both long and short six-month and three-month period models. However, the number of significant periods of the *History* family decreases for both long and short three- and six-month period models.

Finally, fluctuations of the properties of BICs are more stable in our JIT models. This suggests that although properties of intrinsic bugs tend to evolve as projects age, the properties of extrinsic bugs fluctuate more drastically from period to period.

Answer to RQ3: When removing extrinsic bugs, the importance of the *Size* family increases (up to 18 percent AUC points), but the importance of the *Review* family decreases (up to 36 percent AUC points). Furthermore, the importance of most families of code changes are more stable through periods, suggesting that the properties of BICs tend to evolve less drastically with the project over time.

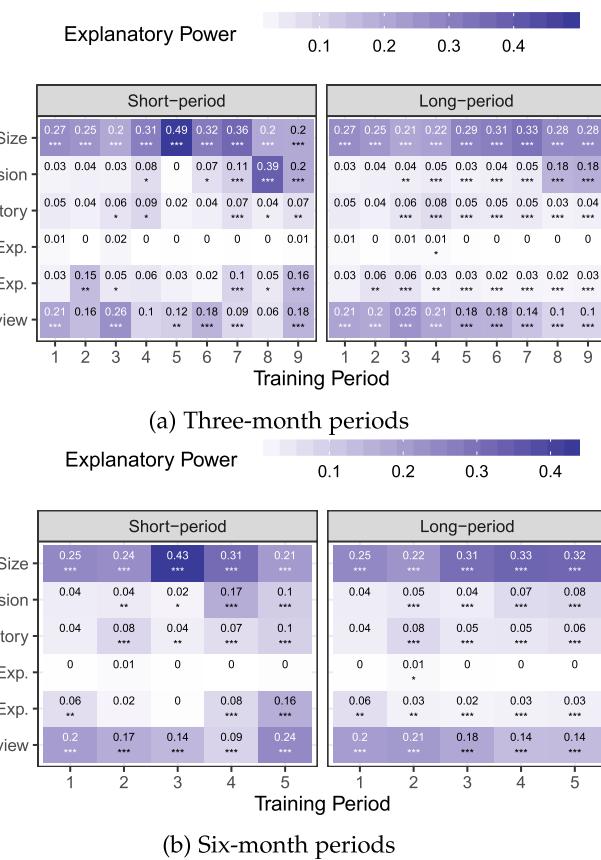


Fig. 5. Evolution of the importance scores of the six studied families of code change properties over time. Shaded indicates magnitude while asterisks indicate significance according to Wald χ^2 test, where: * $p < 0.05$; ** $p < 0.01$; *** $p < 0.001$.

6.4 RQ4: How Accurately do Current Importance Scores of Code Change Properties Represent Future Ones When Extrinsic Bugs are Removed?

Approach. As Mc&K's paper, we used the Family Importance Score (*FIS*) metric to study the stability of the importance scores of each family of code change properties. $FIS(f, n)$ is the jointly tested model terms for all metrics belonging to a family f in the model of period n .

These periods can be the training periods which are represented by i , or the testing periods –or future periods– which are represented by j . Thus, for each one of the JIT models (short and long period) and for each family f , we computed the differences between the importance scores of each family in the training periods i and future periods j using $FISDiff(f, i, j) = FIS(f, i) - FIS(f, j)$.

When the difference between the importance scores of a family f in periods i and j is higher than 0, this family has a larger importance in period i (training) than in period j (future). In such cases, the JIT model (trained using period i) *overestimates* the future importance of family f . On the contrary, when that difference is lower than 0, it indicates that family f has smaller importance in period i (training) than in period j (future). If this occurs, the JIT model (trained using period i) *underestimates* the future importance of family f .

When the model overestimates the future importance of a family f , the impact of that family at the end of the period

might be smaller than anticipated. On the other hand, when the model underestimates the future importance of a family f , the impact of that family at the end of the period might be bigger than anticipated. Software Quality Assurance (SQA) teams can use these importance scores to estimate quality improvements for future periods.

Results. Fig. 6 presents a series of heat-maps with the differences between the importance score in period i and j for each of the six code change property families. Furthermore, each cell reports the statistical significance of the importance score.

In the three-month period models, Fig. 5a shows that the *Size* family spikes in period 5 with a score of 0.49. Training periods 1, 2, 3, and 4 in Fig. 6a show that the importance of *Size* is underestimated by 22, 24, 29, and 18 percent AUC points respectively for testing period 5 in short periods. In the long period models, the underestimation of the importance of *Size* for testing period 5 has similar values. When period 5 becomes the training period in Fig. 6a, the importance of the *Size* family is overestimated in the short period model by up to 29 percent AUC points. However, in the long period model, the maximum overestimation is significantly smaller: 9 percent AUC points.

The short period models of Fig. 6a shows several fluctuations in the importance score of each family over the periods. In the six-month period models, Fig. 6b shows the same trend for the *Size* family but with less severe overestimation or underestimation.

Thus, similar to Mc&K's paper, the importance of the *Size* family is underestimated while the *Review* family is overestimated when removing extrinsic bugs, especially in training periods 1, 2, and 3. However, we found that either long or short period models perform similar. The fluctuations in importance in long period models are not smoother than the fluctuations in short periods.

Answer to RQ4: When removing extrinsic bugs, long-period models do not outperform short periods when analyzing the stability of the importance scores. Larger amounts of training data will not smooth the impact or fluctuations between periods.

6.5 RQ5: How do Mislabeled Bugs Affect JIT Models?

Approach. Although we manually identified mislabeled bugs, we decided to include them into the dataset that fed JIT bug prediction models, i.e., we just removed extrinsic bugs in answers RQ2-RQ4. The reason for doing this is because Tantithamthavorn *et al.* recently found that mislabeled bugs do not have much impact in defect prediction when analyzing whether a file will be buggy or not [15], so we expected it to be the same for JIT models. With RQ5, we want to evaluate if this is true.

We created a ground truth dataset by removing extrinsic and mislabeled bugs from Mc&K's dataset. Since this dataset only contains intrinsic bugs, the most accurate JIT bug prediction models are to be obtained when using this dataset for training the models. Therefore, to study the impact of mislabeled bugs on JIT bug prediction models, we compared the results obtained after training JIT models using

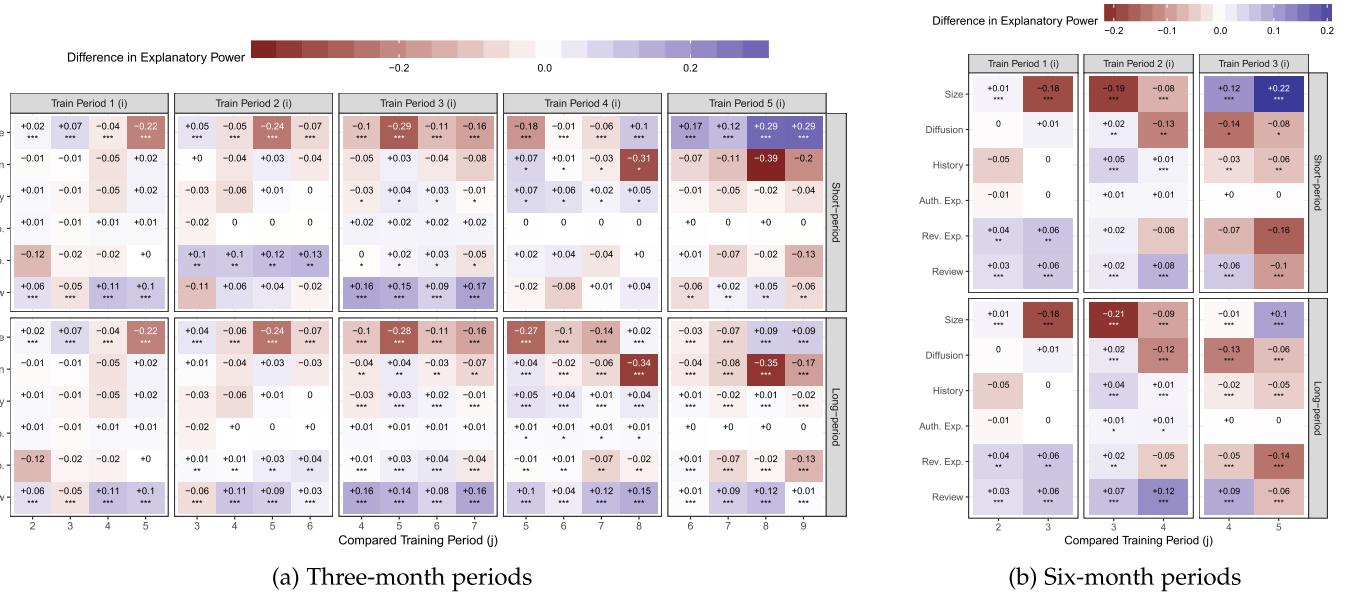


Fig. 6. The stability of the importance scores of the studied families of code change properties ($\text{FISDiff}(f, i, j)$).

the ground truth dataset with the results obtained after training JIT models using intrinsic and mislabeled bugs.

Finally, to obtain a complete picture, we compared the results using the ground truth dataset (i.e., only intrinsic bugs) with the results of (1) intrinsic and mislabeled bugs, (2) intrinsic and extrinsic bugs, and (3) intrinsic, extrinsic, and mislabeled bugs (i.e., Mc&K's results).

Results. While manually curating the dataset, we have identified 548 mislabeled bugs (29.1 percent) and 212 extrinsic bugs (11 percent) in Mc&K's dataset. The percentage of mislabeled bugs is similar to the percentage reported in previous studies, ranging from 33 to 40 percent [30], [33]. Furthermore, the percentage of extrinsic bugs is also similar to the one reported in previous studies (9-21 percent) [2].

Extrinsic bugs were removed in RQ2 and RQ3. To obtain the ground truth dataset, we removed the 548 mislabeled bugs from our dataset. Since mislabeled bugs are not bugs, they do not have a BIC. But, they have a BFC. So, we removed the link between issueIDs classified as mislabeled bugs and their BICs, and we trained again the JIT models, this time using this new dataset (i.e., using the ground truth dataset, composed of 1,120 issues and 1,571 BICs).

We followed the procedures described in RQ2 and RQ3 to analyze the most accurate performance that JIT models can have using the ground truth dataset. We compared these results with (1) the performance of JIT models when mislabeled bugs are included in the dataset; (2) the performance of JIT models when extrinsic bugs are included in the dataset; and (3) the performance of JIT models when mislabeled and extrinsic bugs are included in the dataset. We will report the results of this RQ in textual form, due to space constraints. All figures corresponding to the ones in RQ2-RQ4 for the scenarios under study in RQ5 can be found in the online Appendix.⁸

Table 5 shows the delta comparison between the ideal results (only intrinsic bugs) and the results of the different

JIT models implemented for this RQ. We obtain a complete picture of how extrinsic bugs and mislabeled bugs affect the performance of JIT bug prediction models. A score of 0 in the table means that for that particular case, the JIT model performs as good as the ideal JIT model.

Training JIT Models With Intrinsic and Mislabeled Bugs. when the datasets contain intrinsic bugs and mislabeled bugs, the performance of the models decrease up to 4 percent AUC points for both short and long periods of the three month period models. Furthermore, the performance also decreases 2 percent AUC points for both short and long periods of the six month period models. These models are almost as stable as the models trained with only intrinsic bugs.

The importance of the studied families differ from the ideal scenario. Although the importance of the *Size* family is slightly overestimated, the importance of the *Diffusion* and the *History* families are overestimated up to 14 percent AUC and 12 percent AUC points for the three month short periods. Moreover, the *History* family is underestimated up to 13 percent AUC points for the three month long periods.

Training JIT Models With Intrinsic and Extrinsic Bugs. the performance of these models decreases up to 3 percent AUC points for both short and long periods of the three month period models. However, the performance of both long periods of the three and six month period models increases up to 3 percent AUC points. This means that these models are over-fitted. These models are as stable as the models trained with only intrinsic bugs for the three month long periods and the six month short periods.

The importance of the *Rev.Exp.* family is overestimated up to 12 percent AUC points, but underestimated up to 10 percent AUC points for the three month long periods and six month short periods, respectively. There are slightly no differences in the importance of the remaining families.

Training JIT Models With Intrinsic, Mislabeled, and Extrinsic Bugs. the performance of these models increases up to 15 percent AUC points and 9 percent AUC points for both short and long periods of the three- and six-month period models respectively.

8. <http://gemarodri.github.io/2019-Study-of-Extrinsic-Bugs/>

TABLE 5
Comparison of the Results of the Different JIT Prediction Models Implemented in This Study
With Respect to the Ideal JIT Bug Prediction Model (Only Intrinsic Bugs)

Issues Total BICs	[Intrinsic + Mislabeled] Bugs				[Intrinsic + Extrinsic] Bugs				[Intrinsic + Mislabeled + Extrinsic] Bugs			
	1,668 2,506		1,332 2,132		1,880 3,067							
	3 months		6 months		3 months		6 months		3 months		6 months	
	Short	Long	Short	Long	Short	Long	Short	Long	Short	Long	Short	Long
Δ % AUC	[-4,3]	[-4,1]	-2	-2	[-3,8]	[-3,3]	[-1,3]	[-1,3]	[-2,15]	[2,15]	[2,9]	[3,9]
Δ Stability	[-1,1]	1	1	1	[-3,-1]	0	0	2	[1,3]	-1	-1	-1
Δ % Size Fam.	[2,3]	[1,-2]	[4,5]	2	[-2,-1]	[-2,2]	[-2,5]	[-4,2]	[-10]	[-9,6]	[-13,-6]	[-5,12]
Δ % Diffusion Fam.	[14]	4	[1,10]	[1,2]	[1,2]	[2,3]	[2,4]	[-3,-2]	16	[-1,14]	10	2
Δ % History Fam.	[1,12]	[-13,-3]	[-3,-2]	[-1,2]	[2,3]	[-2,3]	[-4,2]	[1]	[2]	[-4,4]	[-4,6]	[-1,10]
Δ % Auth.Exp. Fam.	0	0	0	1	3	1	1	-2	3	3	1	2
Δ % Rev.Exp. Fam.	[-7,1]	-2	-7	-2	-2	[1,12]	[-5,1]	[-10,-2]	6	[-4,-3]	6	[3,4]
Δ % Review Fam.	[-3,-2]	-10	-9	[1,-8]	[3,7]	[-3,-1]	[3,-1]	[1,3]	[-2,20]	[5,8]	[1,3]	[2,11]

"[Intrinsic+Mislabeled] Bugs" stands for the models after removing extrinsic bugs. "[Intrinsic+Extrinsic] Bugs" stands for the models after removing mislabeled bugs. "[Intrinsic+Mislabeled+ Extrinsic] Bugs" stands for McIntosh and Kamei's models [14].

Therefore, these models are over-fitted, which may cause a poor predictive performance.

The importance of the *Size*, *Diffusion*, and *History* families is either overestimated or underestimated for the three and six month periods. The importance of the *Review* family is overestimated up to 20 percent AUC points for the three short period models.

Answer to RQ5: Mislabeled bugs affect JIT models reducing their performance up to 4 percent AUC points and underestimating the importance of the *History* family. Extrinsic bugs affect JIT models reducing their performance up to 3 percent AUC points and overestimating the importance of the *Rev.Exp.* family.

6.6 RQ6: Are the Properties BFCs and BICs Linked to Extrinsic, Intrinsic, and Mislabeled Bugs Different?

Approach. During the manual classification, we found that Mc&K's dataset not only contained extrinsic bugs, but also mislabeled bugs. Thus, to further understand whether code change properties are different among these categories, we analyzed the distributions and probability density of the six families of code change properties (see Table 3) of (1) the commit identified as BFCs and (2) the commits identified as BICs. Notice that, although extrinsic bugs and mislabeled bugs cannot be linked to a BIC, in this RQ we analyze whether there are differences between the BICs linked to intrinsic bugs and those BICs (incorrectly) linked to extrinsic bugs and mislabeled bugs.

We then compute whether the differences between these three groups were statistically significant across the six families of code change properties for BFCs and BICs. For that, we used the Kruskal-Wallis test [40]. This test is a non-parametric statistical test that assesses the differences among three or more independently sampled groups on a single, non-normally distributed continuous variable.⁹

Finally, we analyze how different these three groups are when they are paired in two groups i.e., Extrinsic-Intrinsic,

9. We found that the final dataset contained skewed data using the function *skewness* with the *e1071* package in R.

Extrinsic-Mislabeled, and Intrinsic-Mislabeled. For that, we used the Wilcoxon Signed Rank test [41] which is a non-parametric test that statistically compares the average of two dependent samples and assesses for significant differences.

Results a) Code Change Properties of BICs for intrinsic, extrinsic, and mislabeled bugs

Fig. 7 shows violin plots with the distribution for each family of code change properties of the manually classified intrinsic, extrinsic, and mislabeled issues. The kernel plot indicates the distribution shape of the data. Wider sections represent a higher probability that members of the population will take on the given value; skinnier sections represent a lower probability.

Fig. 7 shows the distribution shape among the three groups per family of code change properties. Fig. 7a shows a bimodal distribution for extrinsic bugs. The distribution shape of intrinsic and mislabeled bug is however unimodal. Besides, Fig. 7b shows that the distribution frequency of intrinsic bugs are concentrated in lower values while the distribution frequency for extrinsic and mislabeled bugs is more uniform.

Fig. 7 offers evidence that (1) intrinsic and extrinsic bugs have different distributions and medians for all the six families; (2) intrinsic and mislabeled bugs also have a different distribution and medians; and (3) extrinsic and mislabeled bugs are more similar than intrinsic and mislabeled bugs in terms of distributions shape and median.

After computing the Kruskal-Wallis test for the six families of code change properties, we obtained $p - values < 0.05$ in five of them. Thus, the *Size* (1.9E.-014), *Diffusion* (2.2E.-16), *Reviewer* (1.6E.-06), *Author* (2.6E.-05), and *Review* (0.0005) families can be considered different for BICs linked to extrinsic, intrinsic and mislabeled bugs.

Table 6 shows which pairs of groups are different for the six families of BIC code change properties. This table offers evidence that the differences between intrinsic bugs and extrinsic bugs or mislabeled bugs are statistically significant for five out of six families. Furthermore, this table also points out that extrinsic bugs and mislabeled bugs are similar in four out of six families, i.e., *Author*, *Reviewer*, *History*, and *Review*. This finding illustrates that (1) intrinsic, extrinsic, and mislabeled bugs are not the same; and (2) extrinsic bugs and mislabeled bugs have code change properties that are very similar.

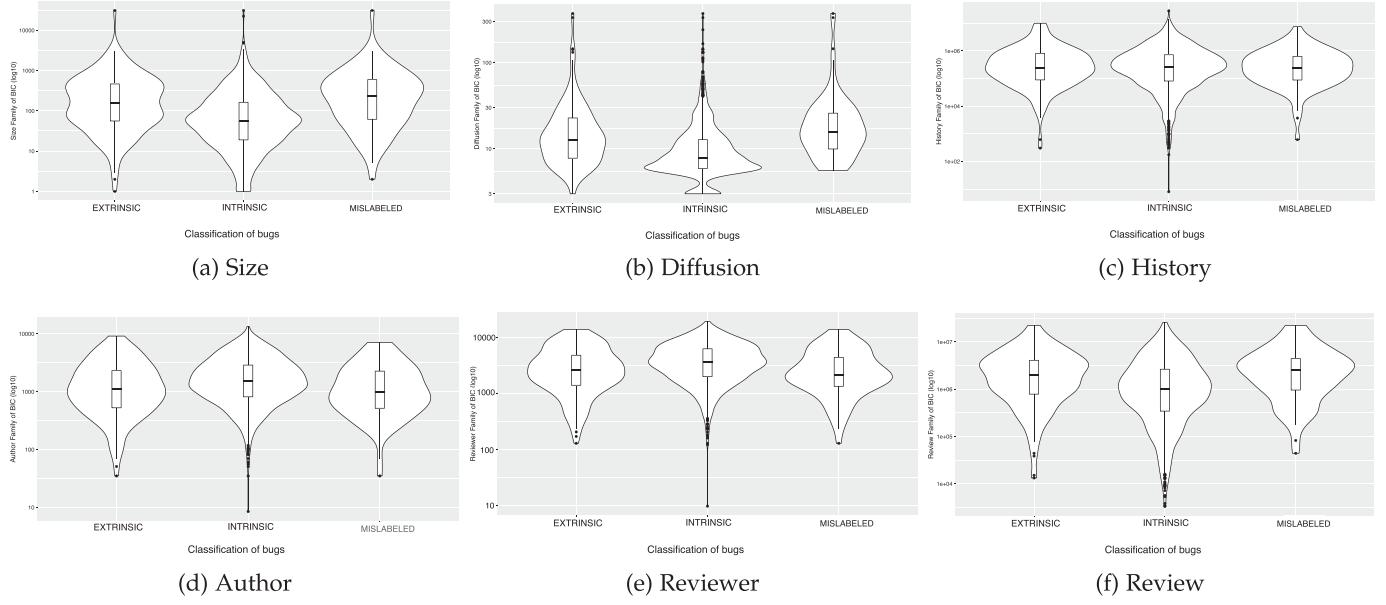


Fig. 7. Distribution of intrinsic bugs, extrinsic bugs, mislabeled bugs and all bugs for the six families of code change properties. The families of code change properties are shown in Table 3.

Results b) Code Change Properties of BFCs for intrinsic, extrinsic, and mislabeled bugs

After computing the Kruskal-Wallis test for the six families code change properties of BFCs, we obtained $p - \text{values} < 0.05$ in three of them. Thus, the *Size* (4.8⁻⁰¹⁶), *Reviewer* (0.0003), and *Author* (0.002) families can be considered different for BFCs linked to extrinsic, intrinsic and mislabeled bugs.

Table 7 shows which pairs of groups are different for the six families of BIC code change properties. This table offers evidence that (1) the differences between intrinsic and extrinsic bugs are statistically significant for *Size* and *Author*, (2) the differences between intrinsic and mislabeled bugs are statistically significant for *Size*, *Author*, *Reviewer* and *Review*; and (3) the differences between extrinsic bugs and mislabeled bugs are statistically significant for the *Size* family.

Answer to RQ6: Intrinsic and extrinsic bugs have different code change properties. When analyzing mislabeled bugs as well, we have found that the *nature* of extrinsic bugs is closer to them than to intrinsic bugs. These differences are statistically significant in five out of six families for BICs. For BFCs, half of code change families are statistically different.

7 DISCUSSION AND FURTHER RESEARCH

In this section, we discuss the impact of our results first on JIT models and then on software engineering practice in

general. We also discuss the implications of our results for researchers and practitioners.

7.1 Impact on JIT

Our results show that JIT models fed exclusively with intrinsic bugs obtain a more accurate representation of the real world; issues that are mislabeled bugs and bug reports that are due to extrinsic bugs should be removed.

The impact of this finding is significant, as over the past 15 years many studies have used automatic techniques to collect bug datasets which are formed by bug reports, bug-fixing commits, and bug-introducing changes. These dataset are then used to train bug prediction models [9], [26], [29], [42], [43], [44], [45].

Hence, the results of hundreds of studies on bug prediction [11] may be not as accurate as they could, as they have not discriminated between intrinsic and extrinsic bugs when training their models.

On the other hand, our results support some of Mc&K's results for JIT models. When JIT models are trained without extrinsic bugs, we found that (1) they lose a large amount of predictive power one year after being trained; (2) when trained using periods that are closer to the testing period tend to outperform models that are trained using older periods; (3) long period JIT models do not always retain more predictive power for longer than short period JIT models; and (4) the *Size* family is consistently the top contributor in our JIT models, and fluctuations in short period JIT models are more common than in long period JIT models.

TABLE 6
p-Values Between Bug-Introducing Changes Linked to Different Kind of Bugs (Wilcoxon Rank Sum Test)

	Size	Diffusion	Author	Reviewer	History	Review
Int.-Ext.	2.4E.-09	1.4E.-13	0.001	0.0002	0.9	0.005
Int.-Mis.	4.8E.-08	3.1E.-02	0.002	0.002	0.9	0.012
Ext.-Mis.	2.7E.-01	4.8E.-14	0.6	0.5	0.9	0.7

TABLE 7
p-Values Between Bug-Fixing Changes Linked to Different Kind of Bugs (Wilcoxon Rank Sum Test)

	Size	Diffusion	Author	Reviewer	History	Review
Int.-Ext.	0.02	0.15	0.043	0.05	0.15	0.113
Int.-Mis.	4.9E.-16	0.71	0.005	0.0004	0.17	0.03
Ext.-Mis.	0.005	0.15	0.95	0.61	0.38	0.96

As already mentioned, there is a debate in the research literature on the impact of mislabeled bugs. Some authors found that they introduce noise in the results of prediction models [30], [34], while others contradict this finding [15]. Our paper sheds more light on this topic, as it offers evidence that JIT models that used only intrinsic bugs obtained a more accurate representation of bugs, as intrinsic code change metrics fit better in JIT models. We believe that researchers should be aware of the noise that mislabeled bugs introduce in their dataset. Furthermore, we believe that a necessary criteria to assess the quality of the dataset is to select projects based on their policies to distinguish between bugs and non-bugs.

While mislabeled bugs have been widely studied in previous works [30], [33], [46], extrinsic bugs have been recently discovered [2] and we still do not understand them fully. In our opinion, further research should be devoted to them. It should be noted that in our case study the effect on 212 extrinsic bugs is similar to the one of 568 mislabeled bugs. Future research lines should continue studying how the characteristics of extrinsic bugs impact not just JIT bug prediction models, but also other bug prediction techniques.

7.2 Impact on Software Engineering

We knew that not all the bugs are the same; they could be intrinsic or extrinsic depending of their origin [1], [2]. In this paper, we offer evidence that intrinsic and extrinsic have different code change properties.

We have also found more similarity between extrinsic bugs and mislabeled bugs in the patterns shown in Fig. 7. This result is also supported by Tables 6 and 7 which indicate similar code changes properties between mislabeled bugs and extrinsic when analyzing BFCs and BICs linked to these bugs. Furthermore, we have observed that some code change properties, i.e., *Author*, *History*, *Review*, and *Reviewer* of the BFC linked to extrinsic bugs and mislabeled bugs are similar. This finding might suggest that fixing a bug which does not have a BIC in the VCS can be compared to developing other kind of issues such as a mislabeled bug. In short, extrinsic bugs have similar characteristics than non-buggy changes. We find this evidence worth further research in order to understand the different *natures* of bugs, and in particular extrinsic bugs.

We think our findings might have a broader impact than just improving bug prediction models.

Practices and Processes. In the paper we have seen that, when removing extrinsic bugs, the explanatory power of the *Size* family increases from 11-43 percent to 20-49 percent, but the explanatory power of the *Review* family decreases considerably from 2-59 percent to 6-21 percent (see RQ3). This points out that review practices may affect extrinsic and intrinsic bugs in a different manner, and thus should be addressed differently. In this regard, it would be interesting to see if there are practices that minimize the number (or at least the effect) of extrinsic bugs. We imagine as well that some software architectures could be more robust than others.

Education. We believe that there is currently a strong bias towards training future software engineers exclusively on intrinsic bugs when identifying the origin of bugs as previous studies do not consider the extrinsic nature of bugs [1], [2]. Our findings suggest that we should educate students

in the fact that software bugs do not always have their origin in a change in the VCS. If tools and practices to support bug fixing of extrinsic bugs appear, we should incorporate them to the curriculum.

7.3 Implications

Besides the impact that our results have on JIT models and Software Engineering, we discuss the implications for developers, researchers, and practitioners.

Data Awareness. If researchers include all bugs in their datasets, they are using a dataset which has not been conveniently prepared, and the results could differ from reality. Thus, if developers are aware of the type of bug they are fixing and start labeling them accordingly in bug tracking systems or commit messages, researchers could obtain better datasets for bug prediction models and foster research on this issue. We hypothesize that software projects will benefit from this as well in the long run.

In the past, we had a similar situation when developers started to indicate the ITS bug id in the BFC; this helped considerably in the improvement of the SZZ algorithm [47]. ITSs also offer the possibility to categorize issues as mislabeled bugs. At this point, we do not if our results may lead to a drastic changes for developers because with one case study we are not able generalize. But, in the case of OpenStack the models without extrinsic bugs perform usually slightly better, sometimes much better.

Furthermore, researchers should be aware of their data and put more attention in the data collection process. They must ensure that when gathering data the ITSs selected for the study distinguish between bug reports and other kind of issues. Therefore, data validation is recommended [30].

Tools. The curation of bugs is a labor-intensive task that requires expert knowledge of the software system, which makes it a very costly process. Thus, the development of tools that help in the classification of bugs might be useful for researchers. In the same manner as tools have been developed that help to lower mislabeling [47], [48], [49], new tools could automatically detect intrinsic and extrinsic bugs. These tools can help practitioners and researchers to ensure the maintainability of software systems, nonetheless the quality of the datasets used to train bug prediction models. For example, a new search could be how to use natural language processing techniques in combination with deep learning techniques to classify bugs as extrinsic or intrinsic based on the textual information from the bug reports. Also, another research line could study different techniques to automate as much as possible the theoretical model proposed by Rodríguez-Pérez *et al.* [2] to identify extrinsic and intrinsic bugs. We envision that these tools might be of benefit in other fields of software engineering such as testing/verification, software analytics and software maintenance and evolution.

Research. The different nature of extrinsic bugs compared to intrinsic ones demands as well further research; based on our previous work [1], [2], we conjecture that previous studies have focused much on the latter, but there is a lack of understanding, research and tools on the former. We call for more investigations on the topic. We need to know more about extrinsic bugs. We know very little about them. Are there different types of extrinsic bugs? Are they more costly

than intrinsic bugs? Are they re-opened more often? Can we write software that is less prone to contain extrinsic bugs? Our aim with this paper has been not to focus only on the impact on extrinsic bugs on JIT bug prediction, but to draw attention to the fact that there is a new field of research in knowing more about extrinsic bugs.

8 THREATS TO VALIDITY

The validity of this study is described in terms of the three main threats to validity that affect empirical software engineering research: construct, internal, and external [50].

Construct Validity. Since we are using the replication package provided in Mc&K's paper [14], this study suffers from the same construct validity threats reported in Mc&K's study. We have attempted to mitigate some of these threats. For example, they used the SZZ algorithm to identify BICs without further refinement. SZZ is widely used algorithm in bug prediction research [3], [26], [42], but it is well-known that it suffers from several limitations [16], [31]. In this work, we manually identify those issues that are not related to a bug and then discriminate between extrinsic and intrinsic bugs. Only for the latter the use of SZZ makes sense. The classification of 705 issues by only a single rater can be a threat to the validity. However, we tried to minimize the impact of this threat by training the two raters until they achieved a near perfect agreement before they starting classifying the 705 issues. This training include the analysis of 470 issues (25 percent).

Internal Validity. Although we have experience in OpenStack from investigating it for several years, we have no advanced development expertise in this system. This fact may have influenced the manual classification of bugs into the different types. To mitigate this threat, we discussed the unclear cases, and when no agreement was reached, we treated these cases as Mc&K's paper did (i.e., we considered that these bug reports were "true" bug reports and not other kind of issues).

External Validity. A notable difference between Mc&K's study and ours is that they had two case studies (OpenStack and Qt), while we have only one (OpenStack). The rationale for this is that our study is very labor-intensive; while Mc&K apply directly SZZ to the dataset of 1,880 issues, we have curated them manually. The curation procedure requires to understand the bug in its very detail, which is a non-trivial task. In total, raters have devoted over 250 hours carrying out the task of classifying these 1,880 issues. The study of just one case study prevents us to generalize our findings to other systems. However, our goal was not claim that our results would stand to all systems, but rather to show the exception, we have found that extrinsic bugs can have a significant impact on bug prediction models, at least in one project. We think that our research is successful in this regard, as we demonstrate that intrinsic and extrinsic bugs show different characteristics. In the particular case of JIT models, we offer sufficient evidence that researchers and practitioners should be aware of extrinsic bugs (in addition to mislabeled bugs). Case studies contribute to increase knowledge and gain a deep understanding of particular phenomenon [51]. Also, some theorist argument that case studies help to draw attention to things that need change [52].

9 CONCLUSION

Previous studies on Just In Time (JIT) bug prediction have not only assumed that future BICs are similar to past ones, but also that all bugs from the project can be linked to explicit BICs. As the research literature has shown [1], this does not always happen. Often it is not possible to find a BIC for a bug fix. Those bugs are referred to as extrinsic bugs, and are mainly caused by external factors to the project, such as changes to APIs or changes in the requirements.

Through a case study of the OpenStack system, we have investigated whether extrinsic bugs have an impact on JIT models. Our results indicate the negative role that extrinsic bugs have on the performance of JIT approaches. When removing extrinsic bugs from the trained data used in OpenStack, JIT models obtain a more accurate representation of the real world as indicated by their different (often higher) AUC values in their performance. These models capture change properties better. Therefore, JIT models that are fitted only with intrinsic bugs obtain more stable AUC scores and lose less predictive power.

Our findings also support in part McIntosh and Kamei's results [14]. We found that after removing extrinsic bugs, the values of the importance score of the six source code change families fluctuate as the system evolves and that these fluctuations can lead to underestimate or overestimate the future impact of those families.

Researchers and practitioners should be aware of the data that feed JIT bug prediction models. They should perform data validation to ensure that only intrinsic bugs are considered when training their models. Although with the current state of the art data validation might be tedious and very labor-intensive to achieve, at least researchers should be aware that considering extrinsic bugs during the training of the models might impact bug prediction results.

All in all, we show evidence that extrinsic bugs are of different *nature* than intrinsic bugs. Actually, they are more similar to issues that are not bugs than to intrinsic bugs. We think that this finding is not only relevant for JIT bug prediction models, but that it may impact many other areas of software engineering practice and research, and would like to call for further research on extrinsic bugs.

A future line of research will be the semi-automation of the process to identify extrinsic bugs. Our experience shows that this will not be an easy process because researchers have to understand at least the bug description (in natural language) and the change (code). We envision that a semi-automated process will require the combination of different techniques and tools. For example, to understand the bug description researchers can implement natural language processing; and to understand the source code they can use tools that help researchers to backtrack the evolution of source code lines from their introduction in the file until their modification in the bug fixing commit. More details can be found in our replication package.

Replication Package. We have set up a replication package¹⁰ including data sources, intermediate data, and scripts.

10. <http://gemarodri.github.io/2019-Study-of-Extrinsic-Bugs/>

ACKNOWLEDGMENTS

We acknowledge the support of the Government of Spain through project “BugBirth” (RTI2018-101963-B-100). Many thanks to JJ Merchant, who devoted tons of hours to classify bugs. We would like to thank as well McIntosh and Kamei for their invaluable support, and J.M. Gonzalez-Barahona, A. Serebrenik, A. Zaidman, and D.M. German for their invaluable feedback and discussions.

REFERENCES

- [1] G. Rodríguez-Pérez, A. Zaidman, A. Serebrenik, G. Robles, and J. M. Gonzalez-Barahona, “What if a bug has a different origin? making sense of bugs without an explicit bug introducing change,” in *Proc. 12th Int. Symp. Empir. Softw. Eng. Meas.*, 2018, Art. no. 52.
- [2] G. Rodríguez-Pérez, G. Robles, A. Serebrenik, A. Zaidman, D. M. German, and J. M. Gonzalez-Barahona, “How bugs are born: A model to identify how bugs are introduced in software components,” *Empir. Softw. Eng.*, vol. 25, pp. 1294–1340, 2019.
- [3] Y. Kamei *et al.*, “A large-scale empirical study of just-in-time quality assurance,” *IEEE Trans. Softw. Eng.*, vol. 39, no. 6, pp. 757–773, Jun. 2013.
- [4] J. Śliwerski, T. Zimmermann, and A. Zeller, “When do changes induce fixes?,” in *Proc. Int. Workshop Mining Softw. Repositories*, 2005, pp. 1–5.
- [5] S. Kim, T. Zimmermann, K. Pan, and E. J. Whitehead Jr., “Automatic identification of bug-introducing changes,” in *Proc. 21st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2006, pp. 81–90.
- [6] C. Williams and J. Spacco, “SZZ revisited: Verifying when changes induce fixes,” in *Proc. Workshop Defects Large Softw. Syst.*, 2008, pp. 32–36.
- [7] E. Giger, M. D’Ambros, M. Pinzger, and H. C. Gall, “Method-level bug prediction,” in *Proc. ACM-IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2012, pp. 171–180.
- [8] H. Hata, O. Mizuno, and T. Kikuno, “Bug prediction based on fine-grained module histories,” in *Proc. 34th Int. Conf. Softw. Eng.*, 2012, pp. 200–210.
- [9] T. Zimmermann, R. Premraj, and A. Zeller, “Predicting defects for eclipse,” in *Proc. Int. Workshop Predictor Models Softw. Eng.*, 2007, pp. 9–9.
- [10] N. Nagappan and T. Ball, “Use of relative code churn measures to predict system defect density,” in *Proc. 27th Int. Conf. Softw. Eng.*, 2005, pp. 284–292.
- [11] T. Hall, S. Beecham, D. Bowes, D. Gray, and S. Counsell, “A systematic literature review on fault prediction performance in software engineering,” *IEEE Trans. Softw. Eng.*, vol. 38, no. 6, pp. 1276–1304, Nov./Dec. 2012.
- [12] E. Shihab, A. E. Hassan, B. Adams, and Z. M. Jiang, “An industrial study on the risk of software changes,” in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, Art. no. 62.
- [13] A. Mockus, “Missing data in software engineering,” in *Guide to Advanced Empirical Software Engineering*, Berlin, Germany: Springer, 2008, pp. 185–200.
- [14] S. McIntosh and Y. Kamei, “Are fix-inducing changes a moving target? a longitudinal case study of just-in-time defect prediction,” *IEEE Trans. Softw. Eng.*, vol. 44, no. 5, pp. 412–428, May 2018.
- [15] C. Tantithamthavorn, S. McIntosh, A. E. Hassan, A. Ihara, and K. Matsumoto, “The impact of mislabelling on the performance and interpretation of defect prediction models,” in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 812–823.
- [16] G. Rodríguez-Pérez, G. Robles, and J. M. Gonzalez-Barahona, “Reproducibility and credibility in empirical software engineering: A case study based on a systematic literature review of the use of the SZZ algorithm,” *Inf. Softw. Technol.*, vol. 99, pp. 164–176, 2018.
- [17] A. Zeller *et al.*, “Causes and effects in computer programs,” in *Proc. 5th Int. Workshop Comput.*, 2011, pp. 482–508.
- [18] D. M. German, A. E. Hassan, and G. Robles, “Change impact graphs: Determining the impact of prior codechanges,” *Inf. Softw. Technol.*, vol. 51, no. 10, pp. 1394–1408, 2009.
- [19] T.-H. Chen, M. Nagappan, E. Shihab, and A. E. Hassan, “An empirical study of dormant bugs,” in *Proc. 11th Working Conf. Mining Softw. Repositories*, 2014, pp. 82–91.
- [20] L. Prechelt and A. Pepper, “Why software repositories are not used for defect-insertion circumstance analysis more often: A case study,” *Inf. Softw. Technol.*, vol. 56, no. 10, pp. 1377–1389, 2014.
- [21] A. Ahluwalia, D. Falessi, and M. Di Penta, “Snoring: A noise in defect prediction datasets,” in *Proc. 16th Int. Conf. Mining Softw. Repositories*, 2019, pp. 63–67.
- [22] M. Nayrolles and A. Hamou-Lhadj, “Clever: Combining code metrics with clone detection for just-in-time fault prevention and resolution in large industrial projects,” in *Proc. 15th Int. Conf. Mining Softw. Repositories*, 2018, pp. 153–164.
- [23] M. Tan, L. Tan, S. Dara, and C. Mayeux, “Online defect prediction for imbalanced data,” in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 99–108.
- [24] N. Nagappan, T. Ball, and A. Zeller, “Mining metrics to predict component failures,” in *Proc. 28th Int. Conf. Softw. Eng.*, 2006, pp. 452–461.
- [25] T. L. Graves, A. F. Karr, J. S. Marron, and H. Siy, “Predicting fault incidence using software change history,” *IEEE Trans. Softw. Eng.*, vol. 26, no. 7, pp. 653–661, Jul. 2000.
- [26] S. Kim, E. J. Whitehead Jr., and Y. Zhang, “Classifying software changes: Clean or buggy?,” *IEEE Trans. Softw. Eng.*, vol. 34, no. 2, pp. 181–196, Mar./Apr. 2008.
- [27] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, “The impact of code review coverage and code review participation on software quality: A case study of the Qt, VTK, and ITK projects,” in *Proc. 11th Working Conf. Mining Softw. Repositories*, 2014, pp. 192–201.
- [28] O. Kononenko, O. Baysal, L. Guerrouj, Y. Cao, and M. W. Godfrey, “Investigating code review quality: Do people and participation matter?,” in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2015, pp. 111–120.
- [29] Y. Kamei, T. Fukushima, S. McIntosh, K. Yamashita, N. Ubayashi, and A. E. Hassan, “Studying just-in-time defect prediction using cross-project models,” *Empir. Softw. Eng.*, vol. 21, no. 5, pp. 2072–2106, 2016.
- [30] K. Herzig, S. Just, and A. Zeller, “It’s not a bug, it’s a feature: how misclassification impacts bug prediction,” in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 392–401.
- [31] D. A. da Costa, S. McIntosh, W. Shang, U. Kulesza, R. Coelho, and A. E. Hassan, “A framework for evaluating the results of the SZZ approach for identifying bug-introducing changes,” *IEEE Trans. Softw. Eng.*, vol. 43, no. 7, pp. 641–657, Jul. 2017.
- [32] J. Aranda and G. Venolia, “The secret life of bugs: Going past the errors and omissions in software repositories,” in *Proc. 31st Int. Conf. Softw. Eng.*, 2009, pp. 298–308.
- [33] G. Rodríguez-Pérez, J. M. Gonzalez-Barahona, G. Robles, D. Dalipaj, and N. Sekitoleko, “Bugtracking: A tool to assist in the identification of bug reports,” in *Proc. IFIP Int. Conf. Open Source Syst.*, 2016, pp. 192–198.
- [34] S. Kim, H. Zhang, R. Wu, and L. Gong, “Dealing with noise in defect prediction,” in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 481–490.
- [35] C. Seiffert, T. M. Khoshgoftaar, J. Van Hulse, and A. Folleco, “An empirical study of the classification performance of learners on imbalanced and noisy software quality data,” *Inf. Sci.*, vol. 259, pp. 571–595, 2014.
- [36] F. Rahman, D. Posnett, I. Herranz, and P. Devanbu, “Sample size versus bias in defect prediction,” in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, pp. 147–157.
- [37] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, “An empirical study of the impact of modern code review practices on software quality,” *Empir. Softw. Eng.*, vol. 21, no. 5, pp. 2146–2189, 2016.
- [38] M. Zhou and A. Mockus, “Does the initial environment impact the future of developers?,” in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 271–280.
- [39] F. Zhang, A. Mockus, I. Keivanloo, and Y. Zou, “Towards building a universal defect prediction model with rank transformed predictors,” *Empir. Softw. Eng.*, vol. 21, no. 5, pp. 2107–2145, 2016.
- [40] P. E. McKnight and J. Najab, “Kruskal-wallis test,” *Corsini Encyclopedia Psychol.*, pp. 1–1, 2010.
- [41] E. Whitley and J. Ball, “Statistics review 6: Nonparametric methods,” *Critical Care*, vol. 6, no. 6, 2002, Art. no. 509.
- [42] S. Kim, T. Zimmermann, E. J. Whitehead Jr., and A. Zeller, “Predicting faults from cached history,” in *Proc. 29th Int. Conf. Softw. Eng.*, 2007, pp. 489–498.
- [43] D. Di Nucci, F. Palomba, G. De Rosa, G. Bavota, R. Oliveto, and A. De Lucia, “A developer centered bug prediction model,” *IEEE Trans. Softw. Eng.*, vol. 44, no. 1, pp. 5–24, Jan. 2018.

- [44] Q. Huang, X. Xia, and D. Lo, "Revisiting supervised and unsupervised models for effort-aware just-in-time defect prediction," *Empir. Softw. Eng.*, vol. 24, pp. 2823–2862, 2019.
- [45] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *J. Syst. Softw.*, vol. 150, pp. 22–36, 2019.
- [46] W. Maalej and H. Nabil, "Bug report, feature request, or simply praise? on automatically classifying app reviews," in *Proc. IEEE 23rd Int. Requirements Eng. Conf.*, 2015, pp. 116–125.
- [47] A. Bachmann, C. Bird, F. Rahman, P. Devanbu, and A. Bernstein, "The missing links: bugs and bug-fix commits," in *Proc. 18th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2010, pp. 97–106.
- [48] R. Wu, H. Zhang, S. Kim, and S.-C. Cheung, "Relink: recovering links between bugs and changes," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 15–25.
- [49] M. Wen, R. Wu, and S.-C. Cheung, "Locus: Locating bugs from software changes," in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 262–273.
- [50] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, and A. Wesslén, *Experimentation in Software Engineering*. Berlin, Germany: Springer, 2012.
- [51] P. Runeson, M. Host, A. Rainer, and B. Regnell, *Case Study Research in Software Engineering: Guidelines and Examples*. Hoboken, NJ, USA: Wiley, 2012.
- [52] S. Easterbrook, J. Singer, M.-A. Storey, and D. Damian, "Selecting empirical methods for software engineering research," in *Guide to Advanced Empirical Software Engineering*, Hoboken, NJ, USA: Springer, 2008, pp. 285–311.



Gema Rodríguez-Pérez (Member, IEEE) received the PhD degree in software engineering from the University Rey Juan Carlos, Madrid, Spain, in 2018. She is a postdoctoral fellow with the David R. Cheriton School of Computer Science, University of Waterloo. Her research focuses on identifying and detecting the origin of software bugs. For more information, please visit <http://gamarodri.github.io/PersonalWeb/>



Meiyappan Nagappan (Member, IEEE) received the PhD degree in computer science from North Carolina State University. He is an assistant professor with the David R. Cheriton School of Computer Science, University of Waterloo. His research is centered around the use of large-scale Software Engineering (SE) data to address the concerns of the various stakeholders (e.g., developers, operators, and managers). He has published in various top SE venues such as the *IEEE Transactions on Software Engineering*, FSE, the *Empirical Software Engineering*, and the *IEEE Software*. He has also received best paper awards at the International Working Conference on Mining Software Repositories (MSR 12, 15). For more information, please visit mei-nagappan.com.



Gregorio Robles (Senior Member, IEEE) is an associate professor with the Universidad Rey Juan Carlos, Madrid, Spain. He is specialized in free/open source software research. He is one of the founders of Bitergia, a software development analytics company.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

Efficient Summary Reuse for Software Regression Verification

Fei He[✉], Qianshan Yu, and Liming Cai

Abstract—Software systems evolve throughout their life cycles. Many revisions are produced over time. Verifying each revision of the software is impractical. Regression verification suggests reusing intermediate results from the previous verification runs. This paper studies regression verification via summary reuse. Not only procedure summaries, but also loop summaries are proposed to be reused. This paper proposes a fully automatic regression verification technique in the context of CEGAR. A lazy counterexample analysis technique is developed to improve the efficiency of summary reuse. We performed extensive experiments on two large sets of industrial programs (3,675 revisions of 488 Linux kernel device drivers). Results show that our summary reuse technique saves 84 to 93 percent analysis time of the regression verification.

Index Terms—Regression verification, program verification, abstraction refinement, summary reuse

1 INTRODUCTION

ALONG with the widespread use of software in our daily life, there is a growing concern for software reliability. At the same time, market pressure demands quick product introductions. The software companies are required to introduce new features to their software products in shorter release cycles. Since errors may be introduced with new features, the new products must be reverified to ensure their correctness.

Software verification [1] has made great success in recent years. However, it is still very time-consuming. Verifying every revision of the software is impractical. Inspired by the success of regression testing [2], [3], researchers in formal verification community proposed the technique of regression verification [4], [5], [6], [7], [8]. Taking into consideration that many intermediate results are produced during the verification, and the computation of these results is costly, regression verification aims to make use of these intermediate results in the verification of new program revisions.

Different intermediate results have been proposed for reuse, including abstract precisions, state-space graphs, constraint solver solutions, and interpolation-based procedure summaries. Beyer *et al.* [7] proposed to record the final abstract precision in the previous verification run, and reuse it in the current verification. Henzinger *et al.* [9] proposed to reuse the state-space graph for incremental checking of temporal safety properties. Visser *et al.* [10] noticed the important role of constraint solving in software verification, and proposed to reuse the constraints solving results.

Procedure summaries, representing input/output behaviors of procedures, have been proposed in [11] to be reused in incremental upgrade checking. Note that procedure summaries are reasonably small to store, technically easy to process, and do not require much extra computation effort to be reused. Therefore, reusing procedure summaries is a good choice for regression verification.

Inspired by [11], this paper studies the summary-based regression verification for predicate analysis. In [11], the procedure summaries are mainly constructed by interpolations. In this paper, we consider the summaries constructed using abstract states of predicate analysis. Note that these abstract states are by-products of program analysis [12], [13], [14]. Thus, it does not require additional computational effort to generate these summaries. Moreover, different from existing techniques, our approach considers the reuse of not only procedure summaries, but also loop summaries. We build a unified framework for reusing both of them.

Moreover, we consider regression verification in the context of counter-example guided abstraction refinement (CEGAR) [15]. Summary reuse techniques need to be adapted to the CEGAR framework (see Section 5). A lazy counterexample analysis technique is further proposed to address the effectiveness issue of summary reuse (see Section 5.3). Considering that CEGAR is a widely-adopted technique in software verification [16], [17], [18], [19], [20], our approach can be applied to most state-of-the-art software verifiers. To the best of our knowledge, our approach represents a novel attempt to the regression verification with CEGAR.

We implemented our approach on top of CPAchecker [17]. We have performed extensive experiments on two large sets of industrial programs. The first set of programs contains 1,119 real-world program revisions of 62 Linux device drivers, and the second contains 2,556 artificial program revisions (by mutation) of 426 Linux device drivers. In total, there are 6,749 verification tasks, among which 6,064 are regression verification tasks. Experimental results show a very promising performance of our approach. With the set

• The authors are with the School of Software, Tsinghua University, Beijing 100084, China. E-mail: hefei@tsinghua.edu.cn, yuqianshan@foxmail.com, limingcai0101@yeah.net.

Manuscript received 16 Mar. 2020; accepted 21 Aug. 2020. Date of publication 3 Sept. 2020; date of current version 18 Apr. 2022.

(Corresponding author: Fei He.)

Recommended for acceptance by C. Wang.

Digital Object Identifier no. 10.1109/TSE.2020.3021477

of real-world programs, in comparison to the standalone verification without reuse, our approach solves 216 more regression verification tasks and saves 93.1 percent of analysis time. With the set of artificial programs, our approach solves 10 more regression verification tasks and saves 84.2 percent of analysis time.

The main technical contributions of this paper are summarized as follows:

- We propose a unified framework for reusing both procedure summaries and loop summaries.
- We propose a fully automatic regression verification technique in the context of counterexample-guided abstraction refinement. A novel lazy counterexample analysis technique is developed to improve the efficiency of summary reuse.
- We implement our approach in the software verification tool CPAchecker. Experimental results show the promising performance of our approach.

The remainder of this paper is organized as follows. Section 2 introduces the necessary backgrounds. Section 3 motivates our approach using a simple example. Section 4 reviews the CEGAR-based program verification and the definition of program summaries. Section 5 presents our CEGAR-based regression verification framework. Section 6 reports evaluation results on our approach. Section 7 discusses related work and Section 8 concludes this paper.

2 BACKGROUNDS

2.1 Abstraction and Refinement

Abstraction plays a central role in software verification. Abstraction omits details of the system behaviors, resulting in a simpler model. We call the model before and after abstraction the *concrete* and the *abstract* model, respectively. An abstraction is *conservative* [21] iff it does not omit any behavior of the concrete model. Conservative abstraction guarantees that the properties (more precisely, the *ACTL** properties [21]) established on the abstract system also hold on the concrete system. The reverse, however, is not guaranteed: if the abstract model falsifies the property, the concrete model does not necessarily falsify this property.

The *abstract precision* [7] (for short, *precision*) defines the level of abstraction of an abstract model. The precision must be at a proper level. A too-coarse precision may fail to verify the property; a too-fine precision, however, may lead to state space explosion. Finding a proper precision appears to require ingenuity.

Counterexample-guided abstraction refinement [15] provides a framework for automatically finding proper precisions. Starting from an initial abstract precision, it iteratively checks if the corresponding abstract model satisfies the desired property. If the property is satisfied, it must also hold on the concrete model, the algorithm terminates and reports “correct”. Otherwise, the checker returns a path on the abstract model that falsifies the desired property. The algorithm then checks if the returned path is valid on the concrete model or not. If it is, the algorithm finds a real bug, it thus terminates and reports “incorrect”. Otherwise, the precision is too coarse, and needs to be refined with the counterexample. Then the above process repeats, until either “correct” or “incorrect” is reported.

The abstract precision does not necessarily keep the same throughout the program [22]. To simplify the discussion, we assume in this paper that the abstract precisions are defined at the level of procedures, i.e., each procedure is associated with a unique abstract precision.

2.2 Software Verification

Model checking and program analysis are two major approaches for software verification. Comparing these two techniques, model checking is more precise with fewer false positives produced, while program analysis is comparatively more efficient and can be applied to more programs. An increasing tendency to software verification is to integrate these two techniques together [23], to get a good balance between accuracy and efficiency.

Predicate abstraction [22], [24] is a widely-adapted abstraction technique [1], [19] for software verification. It creates an abstract model with respect to a set of predicates defined on the program variables. This predicate set defines an abstract precision for predicate abstraction. The state space of the abstract model is only related to the number of predicates in the abstract precision. Finding proper predicates is the key problem for predicate abstraction. One popular technique is based on interpolation computation on the counterexamples [25], [26].

Interprocedural analysis deals with programs with multiple procedures. One simple way of interprocedural analysis is to inline a copy of the callee procedure at each of its call sites. The inlining technique is, however, very expensive and may lead to context explosion for recursive procedures. Another interprocedural analysis technique is to use summaries [12], [13]. A procedure summary (or shortly, a summary) describes the input/output behaviors of a procedure. This technique plugs summaries at each call site of the procedure. Re-analysis of the procedure body at each of its call sites can be avoided using this technique, the efficiency is therefore improved.

There are at least two kinds of procedure summaries in literature: the state-based summaries [12], [13], where each summary is a pair of input and output states of the procedure; and the interpolation-based summaries [11], where the summary is an overapproximation of the procedure’s behaviors.

In this paper, we assume a deterministic, single-threaded program and a safety property. To specify the property, a special error location is introduced in the program. We say the program is *correct* if and only if the error location is not reachable.

3 A MOTIVATING EXAMPLE

Fig. 1 shows a simple program that consists of two procedures: *main* and *inc*. A while loop is implemented in the *main* procedure, and in the loop body the *inc* procedure is invoked. The *inc* procedure takes two input parameters: *a* and *sign*, and outputs either *a* + 1 (if *sign!* = 0), or *a* - 1 (if *sign* = 0). We want to verify that the error location (at line 6) is not reachable in any execution of this program.

Consider an invocation to the *inc* procedure (at line 3) with parameters *a* = 0 and *sign* = 1, the returned value is *rv* = 1. The pair of this entry state (i.e., *a* = 0 \wedge *sign* = 1)

```

main()
{
    int i = 0, x = 0;
1:   while(i < 10) {
2:     if(x <= 5)
3:       x = inc(x, 1);
4:     else
5:       x = inc(x, 0);
6:     i = inc(i, 1);
}
7:   if(x < 5) goto error;

inc(int a, int sign)
{
7:   if (sign) return a + 1;
8:   else return a - 1;
}

```

Fig. 1. An example program.

and the exit state (i.e., $rv = 1$) summarizes this execution of the `inc` procedure. Later, when the `inc` procedure is invoked again, if its entry state is again $a = 0 \wedge sign = 1$, then without entering the `inc` procedure, we can immediately determine its exit state as $rv = 1$.

The execution of a loop can also be summarized by a pair of an entry state and an exit state. Consider the `while` loop in the `main` procedure, its entry state (i.e., the state exactly before the program enters the loop at line 1) is $i = 0 \wedge x = 0$, and its exit state (i.e., the state when the program exits the loop at line 6) is $x = 6 \wedge i = 10$. Similar to the procedure summary, the pair of these two states also summarizes an execution of this loop, and is called a *loop summary*.

Assume that the original program evolves to a new revision. Apparently, this new revision needs also to be checked to guarantee its correctness. Assume that in the new revision, the `inc` procedure does not change, then the summaries of this procedure, which were generated in the previous round of verification, can be reused in the new round of verification. Similarly, if the `while` loop does not change in the new revision, the previous-generated summaries for this loop can also be reused. *How to efficiently reuse the previously-generated summaries in regression verification* is the main research problem we want to solve in this paper.

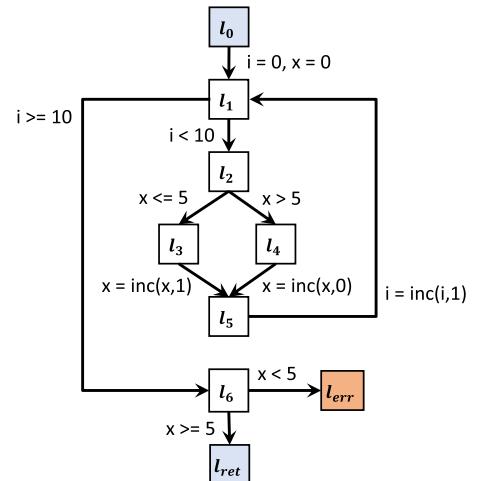
Moreover, in the above discussions, the program is analyzed by tracking its concrete states. The concrete state space of a program is, however, considerably huge and often infinite. A practical verification technique (including the regression verification) needs to be performed on the abstract state space. *How to efficiently combine regression verification and abstraction techniques, especially the counterexample-guided abstraction refinement*, is another research target of this paper.

4 CEGAR-BASED VERIFICATION

In this section, we first review the CEGAR-based program verification, upon which our regression verification scheme is based. We then propose a unified definition for procedure summaries and loop summaries.

4.1 Preliminaries

We begin by introducing the necessary preliminaries for program verification.

Fig. 2. CFA of the `main` procedure in Fig. 1.

Control-Flow Automata (CFA) [22], [23] were adopted in many software verification techniques (for example, BLAST and CPAchecker) for representing programs. Given a program P , let \mathcal{L} be the set of program locations and St be the set of statements of P , respectively. The CFA of P is a pair (\mathcal{L}, G) , where \mathcal{L} is the set of program locations, and $G \subseteq \mathcal{L} \times St \times \mathcal{L}$ is the set of control flow edges. The CFA is different from the control flow graph with program statements labeling the edges rather than the vertices. For example, CFA of the `main` procedure in Fig. 1 is shown in Fig. 2, where l_{err} represents the error location, and l_0, l_{ret} represent the entry and exit locations of the `main` procedure, respectively.

A *state* of a program is a configuration of the program location and the set of facts that we know about the program at that location. Formally, a *concrete state* of the program P is a pair (l, u) , where $l \in \mathcal{L}$ is a program location and u is a full assignment to all variables of P . The assignment u is also called the *concrete data state* of P . Let λ be a set of predicates, representing the current abstract precision. An *abstract state* is a pair (l, s) , where l is a program location, and s is a valuation to all predicates in λ . The valuation s is also called the *abstract data state* of P . In the remainder of the paper, we denote P^λ the abstract model of P with respect to λ .

Consider the CFA of the `main` procedure in Fig. 2, with the abstract precision $\lambda_{main} = \{i < 10, x \leq 5, x < 5\}$. An abstract data state is a valuation to the three predicates in λ_{main} . During the procedure of the analysis, the value of a predicate may be *true* (abbreviated by 0), *false* (abbreviated by 1) or non-deterministic (abbreviated by *). We use a vector to denote an abstract data state. For example, the abstract data state at l_0 is $[*, *, *]$ (for short, written $***$), indicating that all predicates' values are non-deterministic at this location. And when the program transits from l_0 to l_1 , the abstract data state at l_1 is 111, since executing the statements `i=0` and `x=0` can make the three predicates all *true*.

A *path* π of the program is an alternating sequence of states and program statements, i.e.,

$$\pi = (l_0, s_0) \xrightarrow{st_0} (l_1, s_1) \xrightarrow{st_1} \cdots \xrightarrow{st_{n-1}} (l_n, s_n).$$

A path π is a *concrete path* of P (or an *abstract path* of P^λ) iff all states on π are concrete states of P (or abstract states of

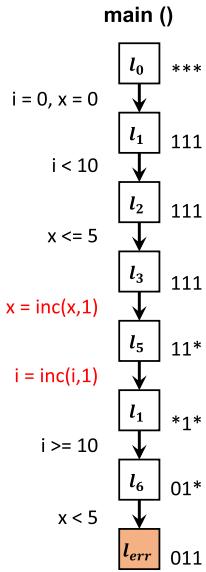


Fig. 3. A counterexample of the program in Fig. 1 with $\lambda_{main} = \{i < 10, x \leq 5, x < 5\}$.

P^λ). A path π is called a *CFA path* if l_0 is the entry location of the program, and for each i with $0 \leq i < n$ there exists a CFA edge $g = (l_i, st_i, l_{i+1})$. In other words, a CFA path represents a syntactical walk through the CFA. A *counterexample* of P (or P^λ) is a CFA path of P (or P^λ) that ends at the error location.

Consider the CFA in Fig. 2 with the abstract precision $\lambda_{main} = \{i < 10, x \leq 5, x < 5\}$. A possible counterexample is shown in Fig. 3, where the abstract data states are labelled beside the corresponding program locations.

We use the *strongest post-condition operator* SP to define the semantics of a CFA path. For a formula φ and a statement st , $SP_{st}(\varphi)$ represents the set of data states that are reachable from any of the states that satisfy φ after the execution of st . Let $st_0, st_1, \dots, st_{n-1}$ be the sequence of program statements passed by the CFA path π . The *semantics* of π is the successive application of the SP operator to each statement of π , i.e., $SP_\pi(\varphi) = SP_{st_{n-1}}(\dots(SP_{st_0}(\varphi)\dots))$.

Definition 1. A CFA path π starting from the abstract state (l, s) is feasible iff $SP_\pi(s)$ is satisfiable.

Note that a feasible path is always a CFA path. Let (l_0, s_0) be the initial state of P^λ . An abstract state (l, s) of P^λ is *reachable* iff there exists a feasible path π of P^λ that ends at the location l such that $s \models SP_\pi(s_0)$.

Definition 2. The abstract model P^λ is correct iff the error location is not reachable in P^λ .

4.2 CEGAR

We next describe the scheme of the standalone program verification (i.e., without reuse) via predicate abstraction and CEGAR [22], [23].

Let λ be an abstract precision, and P^λ be the abstract model with respect to λ . Since the predicate abstraction is conservative [24], to verify the program P , it is sufficient to find a proper abstract precision λ such that P^λ is correct. This can be achieved by the scheme of the counterexample-guided abstraction refinement (Fig. 4). Initially, the abstract

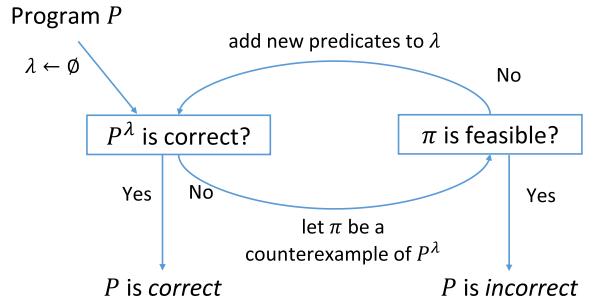


Fig. 4. CEGAR-based verification.

precision is set to empty. The abstract precision is then iteratively refined by adding new predicates, until the program is verified.

Each iteration consists of two phases: a model validation phase and a counterexample analysis phase. During the model validation phase, we check if the abstract model P^λ is correct or not. If P^λ is correct, we immediately conclude that P is also correct. Otherwise, we get a counterexample π that is a CFA path of P^λ ending at the error location.

During the counterexample analysis phase, the counterexample π is semantically analyzed to determine whether it is feasible or not. If it is feasible, we find a real execution of the program P that reaches error, and we thus conclude that P is incorrect. Otherwise, π is a spurious counterexample, and the proof of its infeasibility can be used to refine the abstract precision [22]. The refinement is performed by adding new predicates in the abstract precision, such to eliminate the spurious counterexample from the refined model. After the refinement, the next iteration continues.

4.3 Summaries

We now introduce a unified definition for the procedure and loop summaries.

Let ϱ be a program fragment (either a procedure or a loop). An *entry state* of ϱ is a state at the entry location of ϱ , and an *exit state* of ϱ is a state at its exit location. A pair of an entry state and an exit state summarizes the input/output behavior of one execution of ϱ [12], [13].

Definition 3. A summary of a program fragment ϱ is a triple $\langle \lambda, \phi_{in}, \phi_{out} \rangle$, where λ is an abstract precision, ϕ_{in} and ϕ_{out} are Boolean combinations of predicates in λ , representing an entry state and a set of exit states of ϱ , respectively.

A summary states that if the entry state of ϱ satisfies ϕ_{in} , its exit state must satisfy ϕ_{out} . This definition is particularly suitable for predicate analysis. Let λ be the current abstract precision, and P^λ be the abstract model of P with respect to λ . The model validation is essentially to traverse the state space of P^λ [17] to find that if the error location is reachable or not. Let (l, s) be the current abstract state, and ϱ be the program fragment to be executed, the model validation algorithm needs to traverse all possible paths of ϱ (under the abstract precision λ) to compute the set of exit states. Let ϕ be the formula representing the entry state (l, s) , and ϕ' be the formula representing the set of exit states, the triple $\langle \lambda, \phi, \phi' \rangle$ is a summary of ϱ .

With the above definition, a summary corresponds to a subset of paths in ϱ . The main advantage of using the state-based summaries is the efficiency. Consider a state-based

summary $\langle \lambda, \phi_{in}, \phi_{out} \rangle$, the abstract precision λ is determined at each iteration of CEGAR; the entry state ϕ_{in} and the set ϕ_{out} of exit states are computed in the model validation process. In conclusion, all ingredients of this summary are by-products of CEGAR. There needs no additional computation to generate the state-based summaries.

Note that an abstract precision must be specified in the summary, since the entry state and the exit state must both be defined over the predicates in the abstract precision. Recall that we assume a unique abstract precision throughout a procedure or a loop. Thus only one abstract precision needs to be specified here. Otherwise, if the abstract precision differs in different points of ϱ (for example, as in the lazy abstraction [22]), we need to specify an abstract precision for ϕ_{in} and ϕ_{out} , respectively.

All the generated summaries are maintained in a *summary cache* Ξ . During the program analysis, whenever a program fragment is encountered, the verifier seeks in Ξ for an applicable summary. Let λ_c be the current abstract precision, and s_c be the current abstract data state. A summary $\langle \lambda, \phi_{in}, \phi_{out} \rangle$ of ϱ is called *applicable* iff $\lambda_c \subseteq \lambda$ and $s_c \Rightarrow \phi_{in}$. If any applicable summary exists, the verifier directly uses ϕ_{out} of this summary as the exit state of ϱ . Otherwise, the verifier needs to conduct a heavy fix-point computation [27] on the fragment ϱ to compute its exit state.

5 CEGAR-BASED REGRESSION VERIFICATION

Summaries convey important information about the verification. In this section, we propose some efficient summary reuse techniques for regression verification.

5.1 Overview

An overview of our CEGAR-based regression verification is shown in Fig. 5. Besides the program P , a set Ξ' of the previously-generated summaries is also provided for the regression verification. Note that these summaries are produced by the previous revisions, and may not be applicable to the current revision. Similar to [11], we propose a *summary selection* step to guarantee the safe reuse of summaries (see Section 5.2).

As in a standalone verification, each iteration of CEGAR for a regression verification also consists of two phases: a model validation phase and a counterexample analysis phase. The former phase is exactly the same as in the standalone verification. The *counterexample analysis*, however, requires more careful handling, which will be discussed in Section 5.3.

Note that our summary reuse does not depend on the verification result. A summary here represents an execution of the corresponding program fragment. No matter whether the verification result is “correct” or “incorrect”, as long as the fragment does not change semantically, the summary can be reused. This is very different from the interpolation-based summaries [11], where the summaries are related to the property to be verified, and can only be reused when the verification result is “correct”.

5.2 Summary Selection

Let P' be the old revision of P . For each fragment ϱ (either a procedure or a loop) of P , let ϱ' be its previous version in P' . If ϱ' does not exist in P' , i.e., ϱ is a newly added fragment in P , we simply let $\varrho' = \text{NULL}$.

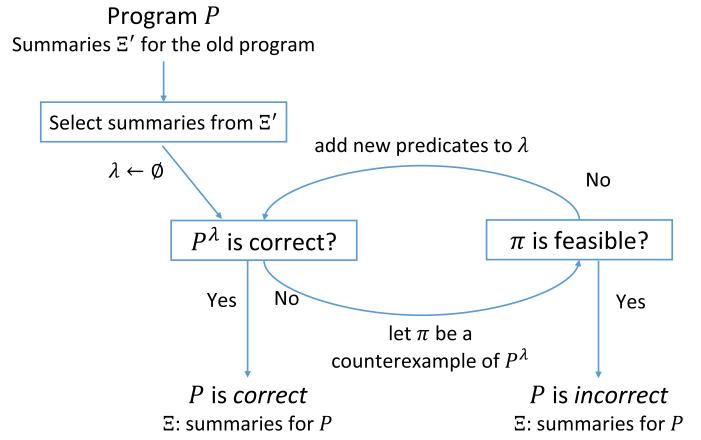


Fig. 5. CEGAR-based regression verification.

In the summary selection (Algorithm 1), we check for each ϱ of P if ϱ is semantically equivalent to ϱ' or not. If it is, the summaries of ϱ' are selected. Otherwise, these summaries are abandoned. These selected summaries are then reused in the regression verification to initialize the summary cache Ξ .

Algorithm 1. Summary Selection

```

forall  $\varrho \in P$  do
  Let  $\varrho'$  be its previous version in  $P'$ ;
  if  $\varrho \equiv \varrho'$  then
    Select summaries of  $\varrho'$ ;
  end
end
Use selected summaries to initialize  $\Xi$ ;
  
```

Note that the semantic equivalence checking $\varrho \equiv \varrho'$ is very expensive in computation, we thus choose to perform syntactic checking instead. The syntactic checking is less precise, i.e., may miss some semantically equivalent fragments, but preserves the soundness, i.e., the fragments that pass the syntactic checking must be semantically equivalent.

5.2.1 Syntactic Checking

In the following, we first discuss two notions, i.e., syntactically unchanged fragments and semantically equivalent fragments, then, we propose our syntactic checking technique.

Definition 4. Let ϱ be a program fragment in P and ϱ' be its previous version in P' , we say ϱ is syntactically unchanged if all statements of ϱ and ϱ' are same, and syntactically changed, otherwise.

In the following, we consider only syntactically unchanged fragments for possible summary reuse. Note that a syntactically changed procedure may still be semantically equivalent to its previous version. We ignore this case since the semantic equivalence checking is too expensive.

On the other hand, a syntactically unchanged fragment is not necessarily to be semantically equivalent. Comparing only statements in a fragment is not enough for checking the semantics of this fragment. For example, the `call` statement in a fragment may lead the execution to a statement outside the fragment (in the called procedure). If the called procedure

changes, even all statements in the fragment under consideration remain the same, its semantics has changed.

Consider the program in Fig. 1, we assume that in the new program revision the main procedure remains the same while the inc procedure changes, then the semantics of the main procedure is regarded as changed.

To summarize, the syntactic changes of a fragment may lead the semantics of another fragment to be changed. We thus have the following definition.

Definition 5. Let ϱ_1, ϱ_2 be two program fragments of P , we say ϱ_1 impacts ϱ_2 , written $\varrho_1 \prec \varrho_2$, if either of the following statements is satisfied:

- ϱ_1 is a procedure and is called in ϱ_2 , or
- ϱ_1 is a loop and is nested in ϱ_2 .

The above *impact* relation extends the call relation over procedures by taking loops and their nesting structures into consideration.

The impact relation is *transitive*, i.e., if $\varrho_1 \prec \varrho_2$ and $\varrho_2 \prec \varrho_3$, then $\varrho_1 \prec \varrho_3$. Let \prec^* be the *transitive closure* of \prec . If $\varrho_1 \prec^* \varrho_2$, i.e., there exist $\varrho_i, \dots, \varrho_{i+k}$ such that $\varrho_1 \prec \varrho_i \prec \dots \prec \varrho_{i+k} \prec \varrho_2$, we call ϱ_2 is *reachable* from ϱ_1 . Moreover, we define the set of *forward reachable* fragments from ϱ_1 as $FReach(\varrho_1) = \{\varrho_2 \mid \varrho_1 \prec^* \varrho_2\}$. Apparently, $FReach(\varrho_1)$ is the *maximal* set of fragments that ϱ_1 can impact. We also define the set of *backward reachable* fragments from ϱ_1 as $BReach(\varrho_1) = \{\varrho_2 \mid \varrho_2 \prec^* \varrho_1\}$, which is the *maximal* set of fragments that have impact on ϱ_1 .

Consider the three fragments in the program in Fig. 1: the main procedure (denoted as ϱ_{main}), the inc procedure (denoted as ϱ_{inc}) and the loop in the main procedure (denoted as ϱ_{loop}). They satisfy: $\varrho_{inc} \prec \varrho_{loop}$, $\varrho_{inc} \prec \varrho_{main}$ and $\varrho_{loop} \prec \varrho_{main}$. Thus we have $FReach(\varrho_{inc}) = \{\varrho_{loop}, \varrho_{main}\}$, $BReach(\varrho_{inc}) = \emptyset$. In other words, any change in ϱ_{inc} can impact the semantics of ϱ_{loop} and ϱ_{main} , and no other fragment can impact the semantics of ϱ_{inc} .

The concepts of forward/backward reachable sets can be lifted to a fragment set. Let τ be a set of fragments, $FReach(\tau) = \bigcup_{\varrho \in \tau} FReach(\varrho)$, and $BReach(\tau) = \bigcup_{\varrho \in \tau} BReach(\varrho)$.

Definition 6. Let ϱ be a program fragment in P and ϱ' its previous version in P' , we say ϱ is *globally syntactically unchanged* if

- 1) ϱ is syntactically unchanged, and
- 2) all fragments in $BReach(\varrho)$ are syntactically unchanged.

Lemma 1. A globally syntactically unchanged fragment is semantically equivalent to its previous version.

Proof. This is a direct conclusion by the definition of globally syntactically unchanged fragment. \square

To find the globally unchanged fragments of P , we syntactically compare each fragment of P to its previous version. According to the comparing results, fragments in P are divided into two parts: the syntactically unchanged set τ_1 and the syntactically changed set τ_2 . Then we have the following lemma.

Lemma 2. Let τ_1 and τ_2 be the set of syntactically unchanged and syntactically changed fragments of P , respectively. Fragments in $\tau_1 \setminus FReach(\tau_2)$ are all globally syntactically unchanged.

Proof. Assume that the lemma does not hold, i.e., there is a fragment $\varrho_1 \in \tau_1 \setminus FReach(\tau_2)$ that is not globally syntactically unchanged. By $\varrho_1 \in \tau_1$ and Definition 6, there must be a fragment $\varrho_2 \in BReach(\varrho_1)$ such that ϱ_2 is syntactically changed. By $\varrho_2 \in BReach(\varrho_1)$, we have $\varrho_1 \in FReach(\varrho_2)$. By ϱ_2 being syntactically changed, we have $\varrho_2 \in \tau_2$, and thus $\varrho_1 \in FReach(\tau_2)$. This is contradicted with the assumption. Thus the assumption does not hold, and the lemma holds. \square

Let $\tau^* = \tau_1 \setminus FReach(\tau_2)$. The semantic equivalence checking $\varrho \equiv \varrho'$ (on Row 4 of Algorithm 1) is implemented as checking whether $\varrho \in \tau^*$. If $\varrho \in \tau^*$, the summaries of ϱ are selected, and otherwise they are abandoned. Note that the computations of τ_1 , τ_2 and τ^* involve only syntactic checking of P and P' . According to Lemmas 1 and 2, all selected summaries are semantically equivalent to its previous version, and thus they can be safely reused in the regression verification.

5.3 Counterexample Analysis

Given a counterexample returned by the model validation process, we need to check if this counterexample corresponds to a real bug or not. Summary reuse makes this process intricate.

Consider the counterexample in Fig. 3 that contains two procedure calls. During the program verification, these two procedure calls are replaced by two *abstract* summaries, which are defined over predicates and may introduce spurious behaviors over the program's concrete semantics. Therefore, to check the feasibility of this counterexample, the inner paths in the inc procedure that correspond to these two summaries must be restored.

Consider the procedure call $inc(x, 1)$ between $(l_3, 111)$ and $(l_5, 11*)$ for example. If the summary for this procedure call is generated in the current verification run, the inner path in inc that leads from $(l_3, 111)$ to $(l_5, 11*)$ is available [17]; otherwise, if the summary is inherited from the previous verification, there is no information for the inner path. Then, we have to rely on the heavy fix-point computation [27] to reproduce this path. In other words, with the counterexample checking, the saved analysis on the inc procedure is getting back. The benefits of summary reuse are thus significantly weakened.

5.3.1 Holes

Definition 7. A summary on a path is called a hole if it is inherited from the previous verification runs.

Let π be a counterexample path with holes. Replacing a hole with the corresponding inner path is called an *expansion*. For example, Fig. 6 shows the expanded version of the counterexample in Fig. 3. We call a path *holeless* if it contains no hole.

Let H be the set of holes on a path π . The path π is split by these holes into $|H| + 1$ path segments. Each of these segments is a *holeless* path. The semantics of a path with holes is defined as the conjunction of the semantics of its holeless segments.

Theorem 1. Let π be a path with holes H , and Π the set of segments of π split by H ,

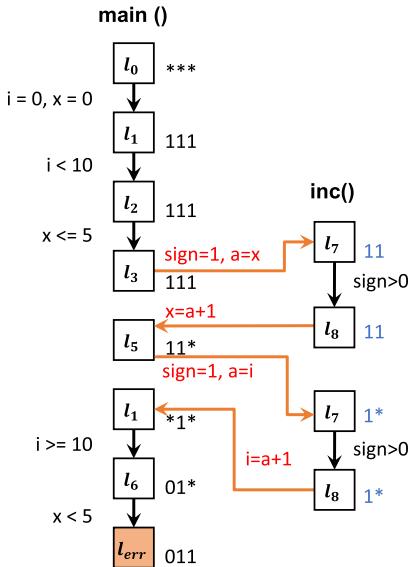


Fig. 6. An expanded version of the counterexample in Fig. 3, with $\lambda_{main} = \{i < 10, x \leq 5, x < 5\}$ and $\lambda_{inc} = \{\text{sign} > 0, a < 5\}$.

- 1) if there exists any infeasible segment in Π , π is infeasible; and
- 2) if all segments in Π are feasible, π is, however, not necessarily feasible.

For the latter case, if all segments of π are feasible, we call the path π *separately feasible*. The above theorem states that the separate feasibility does not imply the feasibility of the whole path. This is obvious since the semantics of holes are not taken into consideration in the separate feasibility.

Consider the counterexample in Fig. 3, the holes at l_4 and l_5 split the path into three segments, i.e.,

$$\begin{aligned}\sigma_1 : (l_0, *** \rightarrow (l_1, 111) \rightarrow (l_2, 111) \rightarrow (l_3, 111), \\ \sigma_2 : (l_5, 11*), \\ \sigma_3 : (l_1, *1*) \rightarrow (l_6, 01*) \rightarrow (l_{err}, 011).\end{aligned}$$

This counterexample is infeasible if any of σ_1 , σ_2 and σ_3 is infeasible. Reversely, even σ_1 , σ_2 and σ_3 are all feasible, the path is not necessarily feasible since the inner paths from l_3 to l_5 and from l_5 to l_1 , that are replaced by summaries, may be infeasible.

5.3.2 Lazy Analysis Algorithm

A *brute-force* algorithm for counterexample analysis is to directly expand all holes of the counterexample, and then check its feasibility. This algorithm is correct but inefficient. Recall that to expand a hole, we need to perform a heavy analysis on the program fragment, which usually costs lots of computation. In the worst case, it needs to traverse all paths of the fragment to reproduce the path from a given input state to a given output state. We therefore propose a technique to avoid unnecessary hole expansions.

Our *lazy analysis* algorithm is shown in Algorithm 2. The basic idea is to expand holes on demand, so as to avoid unnecessary hole expansions. The main body of the algorithm is a *while* loop. At the beginning of each iteration of the loop, the algorithm checks whether the current path is holeless ($isHoleless(\pi)$), and whether the current path is

infeasible ($isInfeasible(\pi)$). If both checks return *false*, the loop continues by expanding one hole in π . Otherwise, the current path π must be either infeasible or holeless. For the former case, the algorithm returns π ; and for the latter case, the algorithm reports “incorrect”.

Algorithm 2. *lazyAnalysis*(π)

```

Input: A finite abstract path  $\pi$ 
Output: The expanded path of  $\pi$  if it is infeasible; or
         “unsafe” if it corresponds to a real path.
while  $\neg(isHoleless(\pi) \vee isInfeasible(\pi))$  do
    let  $h$  be a hole in  $\pi$ ;
     $\pi \leftarrow expandHole(\pi, h)$ ;
end
if  $isInfeasible(\pi)$  then return  $\pi$ ;
else return incorrect;
```

Comparing to the brute-force approach, our lazy algorithm needs more feasibility checking. However, it is still beneficial. First, with the lazy approach, the computational efforts for unnecessary hole expansions (which in many cases are expensive) are saved. Second and more importantly, the returned path by the lazy approach is often much shorter than the fully expanded one. Note that the refinement is a heavy step in CEGAR [26]. With a shorter counterexample, the computation efforts for the refinement (for example, the interpolation-based refinement [26]) can often be significantly reduced.

The lazy analysis algorithm can be easily adapted to the existing CEGAR framework (for example, CPAchecker [17]) in the following way. When the verifier in the existing framework returns a counterexample, our algorithm is applied to check if this counterexample is spurious or not. In case of a spurious counterexample, our algorithm returns a (partially) expanded path and gives it to the existing refiner in the framework. The returned path by our algorithm may contain holes. Treating these holes as value assignments, these paths can be directly processed by most of the existing refinement techniques, for example, the interpolation-based refinement [26].

5.4 Precision Reuse

In the beginning of the regression verification (in Fig. 5), the abstract precision is set to be empty, which is in fact not necessary. Dirk Beyer *et al.* [7] showed that the abstract precision can also be reused in the regression verification. To take this idea, we simply use the final abstract precision λ' in the previous verification to initialize the current abstract precision λ , i.e., replace $\lambda \leftarrow \emptyset$ with $\lambda \leftarrow \lambda'$ in Fig. 5.

With precision reuse, the amount of summaries that need be recorded at the end of the verification run is also reduced. Assume that the abstract precision of the regression verification is initialized as λ' . Then during the regression verification, the abstract precision is iteratively refined by CEGAR. In other words, summaries with a smaller abstract precision than λ' are useless in the regression verification. Thus, we need only to output the summaries with the final precision at the end of each verification run.

6 EXPERIMENTAL EVALUATION

We implemented our regression verification technique on top of CPAchecker [17]. CPAchecker provides a configurable

framework for software verification, with both predicate abstraction and counterexample-guided abstraction refinement supported. To support regression verification, we realized the following functionalities in CPAchecker:

- summary dumping, i.e., dump all summaries to an external file at the end of the verification,
- summary selection, i.e., load and select summaries from an external file at the beginning of the verification (Section 5.2), and
- lazy counterexample analysis (Section 5.3).

Moreover, precision reuse (Section 5.4) was also integrated into our implementation. In the following, we call our enhanced implementation CPAchecker⁺.

Experiments are designed to answer the following research questions:

- RQ1. Is summary reuse efficient enough for regression verification?
- RQ2. What is the impact of precision reuse on the efficiency of our approach?
- RQ3. What is the impact of summary types on the efficiency of our approach?
- RQ4. What is the impact of the lazy counterexample analysis on the efficiency of our approach?

6.1 Experimental Setup

We prepared two industrial benchmarks with 3,675 program revisions of 488 Linux device drivers to evaluate our approach:

- 1) The first benchmark, obtained from [7], consists of 1,119 real-world program revisions of 62 Linux device drivers.
- 2) The second benchmark, prepared by ourselves, consists of 2,556 program revisions of 426 Linux device drivers, where all drivers were collected¹ from the “SystemsDeviceDriversLinux64Reach-Safety” category of the 6th International Competition on Software Verification (SV-COMP’17) [28]. For each driver, we make the program in [28] as the base revision, and use a state-of-the-art mutation tool MiLu [29] to randomly generate 5 artificial new revisions.

Recall that all program revisions in the first benchmark were obtained from the official Linux kernel repositories [7], and they are real revisions implemented by the experienced programmers. We use this benchmark to evaluate our approach on real program changes. In contrast, program revisions in the second benchmark were obtained by adding mutations to the base revision of each device driver. An advantage of the second benchmark is that the mutants generated by random pattern can involve much more *unpredictable modifications* than those written by human programmers. We use the second benchmark to evaluate our approach on a broader range of program changes.

All experiments are performed on a machine with Intel Xeon E5-2620 CPU of 2.4 GHz 24 cores and 32 GB RAM. We

¹ The selection strategy is as similar as in [7]. We limit our selection to drivers of Linux 3.4 kernel and with the mutex lock/unlock specifications, and skipped programs whose total CPU time is less than 0.5s and those that need no refinement.

use Ubuntu 16.04 (64-bit) with Linux 4.4.0 and jdk1.8.0. The CPAchecker is configured using the *predicateAnalysis-ABE* option. Each verification run is limited to 300 seconds (total CPU time), 6 GB of Java heap size and 6 CPU cores.

6.2 Overall Results on Real Revisions (RQ1)

This experiment evaluates our approach on real program revisions. We compare the performance of CPAchecker⁺ with CPAchecker on the first benchmark. All regression verification techniques, including summary reuse, precision reuse, and lazy counterexample analysis are enabled for CPAchecker⁺ in this experiment. For simplicity, in the following, we refer to our approach as “Reuse”, and the standard CPAchecker as “no Reuse”.

In our experiments, a *verification task* is to verify a program revision against a specification. Note that a device driver may have multiple program revisions and also multiple specifications. A pair of a device driver and a specification involves a sequence of verification tasks, where the base revision is verified from scratch, while the other revisions are verification in an incremental way, i.e., as regression verifications. In total, there are 259 driver/specification pairs and 4,193 verification tasks in the first benchmark. Among all tasks, 3,934 are regression verification tasks.

Experimental results are listed in Table 1. Due to page limitation, we restrict this table to the 40 best and 10 worst cases out of the total of 259 driver/specification pairs (sorted by the “Speedup” column). The first two columns (“Driver” and “Specification”) list the device driver name and the specification name, respectively. The third column “LoC” shows the lines of code for the base revision of each device driver. The fourth column (“#T”) shows the number of regression verification tasks (i.e., the number of revisions minus 1) for each driver/specification pair. The fifth column (“ T_{1st} ”) lists the analysis time for verifying the first revision which is not a regression verification task. This “ T_{1st} ” gives us the information on the complexity of verifying each device driver.

The following two column assemblies report the experimental results by “no Reuse” and “Reuse” approaches, respectively. For both approaches, we report the number of successfully verified regression tasks “#solved”; the total number of abstract successor computations “#abs_succ”² and the total analysis time “ T_{rv} ” (in seconds) for each driver/specification pair. To conduct a fair comparison, “ T_{rv} ” and “#abs_succ” are limited to regression verification tasks that are solved by both approaches. The “Speedup” column shows the average speedup of “Reuse” approach over the “no Reuse” approach, calculated by: $1 - T_{rv2}/T_{rv1}$. The last “AvgFSIZE” and “RSR” columns report the average size (in Kilobytes) of summary files, and the average reusable summary ratio, among all revisions of each driver/specification pair, respectively. For each regression verification task, RSR is the proportion of summaries that are kept after the summary selection.

Recall that each row in the table corresponds to a driver/specification pair. The last two rows (“Sum” and “Avg”)

² An abstract successor is a successor of the current state on the abstract model. Abstract successors computation needs to invoke a SMT solver and is considered as the most time-costly operation in predicate abstraction [23].

TABLE 1
Overall Experimental Results on Real Revisions

Device	Run set Spec.	LoC	#T	T_{1st}	#solved	no Reuse #abs_succ	T_{rv1}	#solved	Reuse #abs_succ	T_{rv2}	Speed -up	Avg FSize	RSR
wm831x	39_7a	5183	34	61.9	34	346.7K	144.0	34	0.2K	1.4	99.0%	7.2	0.8
cx231xx	08_1a	8241	13	21.0	11	356.4K	203.9	13	0.1K	1.8	99.0%	1.5	0.8
wm831x	32_7a	4482	34	2.2	34	253.8K	125.1	34	0.2K	1.5	98.8%	5.7	0.8
usb-az6007	08_1a	7912	5	60.6	5	250.4K	94.5	5	0.2K	1.2	98.7%	2.5	0.7
wm831x	08_1a	4579	34	50.2	34	178.1K	103.2	34	0.2K	1.3	98.7%	4.1	0.8
abyss	68_1	5382	2	82.6	2	41.8K	35.6	2	0.0K	0.5	98.7%	2.6	0.8
abyss	32_1	4251	3	65.1	3	46.6K	46.9	3	0.0K	0.6	98.7%	2.3	0.8
wm831x	68_1	6569	3	64.7	3	164.9K	73.0	3	0.1K	1.0	98.6%	4.4	0.9
leds-bd2802	32_1	6346	4	57.0	4	232.1K	84.7	4	0.1K	1.2	98.5%	8.5	0.9
cp210x	68_1	8169	14	44.2	10	234.1K	102.8	14	0.3K	1.1	98.5%	3.3	0.9
wm831x	32_1	5339	3	49.7	3	121.9K	61.2	3	0.1K	0.9	98.5%	4.1	0.9
sppc8x5	68_1	6086	13	68.0	12	156.2K	86.1	13	0.2K	1.3	98.4%	4.6	0.9
mos7840	08_1a	8971	60	81.0	28	270.6K	156.1	60	0.3K	1.3	98.2%	6.1	0.6
sppc8x5	32_1	6002	13	52.5	13	112.8K	72.3	13	0.2K	1.3	98.2%	4.3	0.9
sil164	39_7a	7026	3	54.2	3	74.3K	33.8	3	0.1K	0.6	98.2%	8.7	0.8
cx231xx	39_7a	9959	13	61.5	2	67.3K	29.7	13	0.0K	0.1	98.1%	12.0	0.6
xilinx_uartps	32_7a	5393	3	75.2	3	47.0K	45.6	3	0.1K	1.0	97.9%	3.7	0.8
tcm_loop	39_7a	12515	41	52.4	41	186.9K	72.7	41	0.2K	1.6	97.8%	16.1	0.9
xilinx_uartps	08_1a	5239	3	72.0	3	46.8K	44.3	3	0.1K	1.0	97.7%	3.3	0.8
catc	32_1	6245	10	25.2	10	39.1K	40.0	10	0.1K	0.9	97.6%	3.0	0.9
sil164	32_7a	6805	3	53.6	3	68.3K	43.8	3	0.2K	1.1	97.5%	7.3	0.8
ems_usb	39_7a	9647	21	70.3	21	175.9K	70.2	21	0.2K	1.7	97.5%	11.3	0.7
tdo24m	39_7a	5529	12	71.4	12	220.7K	107.1	12	0.7K	2.7	97.5%	6.5	0.6
i915	68_1	13916	79	37.9	79	26.7K	41.2	79	0.0K	1.1	97.3%	1.8	1.0
cp210x	39_7a	6909	71	79.8	56	300.1K	116.2	71	0.9K	2.5	97.3%	8.2	0.6
ssu100	08_1a	5985	28	41.9	28	54.4K	45.5	28	0.2K	1.4	97.0%	3.8	0.8
i2o_scsi	08_1a	5644	7	42.3	7	27.1K	35.8	7	0.1K	1.1	97.0%	2.1	0.7
i2o_scsi	39_7a	7515	6	65.5	6	80.0K	49.6	6	0.2K	1.5	97.0%	8.5	0.7
i915	32_1	13557	79	30.8	79	18.5K	35.7	79	0.0K	1.1	97.0%	1.6	1.0
cp210x	08_1a	8370	71	183.6	64	167.6K	94.4	71	0.9K	2.6	96.9%	2.9	0.6
budget-patch	39_7a	8446	9	105.8	1	265.9K	105.8	8	0.1K	0.4	96.9%	11.3	0.8
tty-serial	39_7a	6216	9	81.8	9	135.1K	66.8	9	0.2K	2.1	96.9%	17.4	0.8
sppc8x5	08_1a	5751	37	49.8	37	118.0K	62.5	37	0.3K	2.1	96.7%	3.7	0.7
ems_usb	32_1	8646	6	30.7	6	24.0K	26.0	6	0.1K	0.9	96.6%	4.7	1.0
i915	39_7a	14298	79	49.5	79	70.7K	60.9	79	0.3K	2.1	96.6%	8.3	0.9
catc	08_1a	5878	22	23.4	22	51.8K	42.5	22	0.2K	1.5	96.5%	3.6	0.8
cx231xx	32_7a	8393	13	22.5	2	5.8K	12.1	4	0.0K	0.2	96.4%	4.5	0.6
wl12xx_sdio	39_7a	8578	38	67.4	38	120.6K	60.6	38	0.5K	2.2	96.4%	32.9	0.9
abyss	08_1a	4136	4	26.2	4	17.6K	20.5	4	0.0K	0.7	96.4%	1.8	0.6
catc	39_7a	7941	22	46.6	22	537.7K	182.1	22	3.7K	6.8	96.3%	29.8	0.8
⋮				⋮				⋮				⋮	
i2c-algo-pca	32_7a	2879	14	0.8	14	0.4K	1.9	14	0.1K	0.9	50.0%	1.7	0.8
mtd-mtdoops	43_1a	2898	20	1.6	20	0.2K	1.9	20	0.1K	1.0	50.0%	2.9	0.7
mtd-ar7part	32_1	1003	2	1.2	2	0.1K	0.7	2	0.0K	0.3	49.6%	0.7	0.7
wm831x	43_1a	4246	3	46.1	3	64.6K	43.5	3	20.6K	26.0	40.3%	18.8	0.9
farsync	32_7a	6823	9	2.4	9	0.8K	2.9	9	0.4K	1.8	38.0%	5.8	0.7
drbd	08_1a	57282	96	3.4	96	0.1K	5.2	96	0.0K	3.3	35.6%	0.6	1.0
i2c-algo-pca	43_1a	3031	7	0.9	7	0.1K	1.5	7	0.1K	1.2	18.4%	1.0	0.4
vmalloc	32_7a	3885	29	11.0	29	3.4K	9.9	29	0.8K	8.2	17.3%	1548.7	0.6
paride-pt	32_7a	5163	9	21.3	9	13.4K	18.9	9	13.3K	17.4	8.0%	3.2	0.4
mtd-ar7part	43_1a	1000	2	0.5	2	0.0K	0.6	2	0.0K	0.6	-8.9%	0.9	0.5
Sum	-	1.5M	3934	6.0K	3588	228.3M	151.8K	3804	3.2M	10.5K	-	-	-
Avg	-	6128	15.2	23.3	13.9	881.4K	586.0	14.7	12.5K	40.5	93.1%	17.4	0.8

take the total and average amount of all rows in the table, respectively.

From Table 1, we observed that our method outperforms “no Reuse” in vast majority of cases. Considering “Speed-up” column, among 259 driver/specification pairs, only one pair that our method is slower. Comparing “#solved” columns of both approaches, 216 more regression verification tasks were solved with our approach. This witnesses the value of summary reuse for regression verification.

Among the common 3,581 regression verification tasks that both approaches can verify, “no Reuse” takes 151.8 thousand seconds of analysis time while our “Reuse” approach

finishes in 10.5 thousand seconds. The overall time speedup of our approach is 93.1 percent.

Comparing the numbers of abstract successor computations (“#abs_succ.”) required by “Reuse” and “no Reuse” for each spec/driver pair, we found that our method cuts down the amount significantly (about 98 percent reduction), which can explain the reason for the speedup of analysis time.

Let us look at the “AvgFSize” column. The average size of summary files among all regression revisions is 17.4 KB (the median is 3.8 KB). The added overhead by our approach in storage is acceptable.

TABLE 2
Overall Experimental Results on Larger Changes

Revs.	Avg.		no Reuse		Reuse		Speed-up	RSR
	#T	Diff. Lines	#solved	T_{rv1}	#solved	T_{rv2}		
All	3934	511	3588	151.8K	3804	10.5K	93.1%	0.8
4th	898	1242	812	32.7K	870	4.5K	86.1%	0.6
2Revs	259	1562	240	7.7K	245	1.5K	80.8%	0.4

6.2.1 Scaling With Larger Changes

The changes between adjacent revisions may not be very significant. We further use the following settings to evaluate our approach on larger changes:

- *4th*: we set up a regression verification task every 4 revisions of the program, and
- *2Revs*: we set up a regression verification task for the last revision of each program.

Finally, we get respectively 898 and 259 regression verification tasks using the above two settings.

Experimental results are listed in Table 2. The original setting that incrementally verifies each adjacent revision of the program is referred to as *All* in the table. The average numbers of changed lines for regression verification tasks using the above three settings are 511, 1242 and 1562, respectively. The increasing *avg. diff. lines* lead to decreasing *RSR*, which is reasonable since more program changes can of course lead less summaries to be reusable.

Observe that our approach can still get considerable performance improvements (86.1 and 80.8 percent of speedups) with the *4th* and *2Revs* settings, which show the effectiveness of our approach on larger changes.

6.3 Overall Results on Artificial Revisions (RQ1)

The second experiment evaluates our approach on artificial program revisions in the second benchmark. This experiment contains 2,556 verification tasks, involving 426 driver/specification pairs. Among all tasks, 2,130 are regression verification tasks.

Results of this experiment are listed in Table 3. Again, we limit this table to the 40 best and 10 worst cases out of all 426 driver/specification pairs (sorted by the “Speedup” column). Each column is with the same meaning as in Table 1. Note that there is only one specification for each driver in this benchmark, the “Spec.” column is thus skipped.

From this table, we observed similar results as in Table 1. Among all 426 driver/specification pairs, our approach wins on 389 pairs. In total, our approach solved 10 more verification tasks, and the average speedup is 84.2 percent.

6.4 Comparison With Existing Tools (RQ1)

To further demonstrate the efficiency of our approach, two more experiments were conducted to compare CPAchecker⁺ with the existing regression verification tools:

- eVolCheck [30], a regression verification tool that implements the technique of interpolation-based procedure summaries [11], and
- UAutomizer⁺ [31], a regression verification extension of the famous software verification tool UAutomizer [19].

6.4.1 Comparison With eVolCheck

This experiment was conducted on the set of real-world programs. Before this experiment, some of the programs need to be modified to adapt to the input format of eVolCheck, e.g., replacing “ldv_error()” by “assert(0)”.

Note that eVolCheck [30] is just an experimental implementation, and is not fully optimized.³ Among all 4,193 verification tasks, eVolCheck failed on 3,646 tasks due to various parsing and runtime errors. The comparative experiment was conducted on the remaining 547 verification tasks.

The comparison results are listed in Table 4, where the T_{rv1} and T_{rv2} columns report the total regression verification time *without* and *with* reuse, respectively, and the “Speedup” is calculated by $1 - T_{rv2}/T_{rv1}$. Note that eVolCheck employs the bounded model checking technique, and its unwinding factor was set to 15. Among the 547 verification tasks, eVolCheck solved (i.e., the underlying SMT solver returned a result) 143 tasks within the time limit of 300 seconds, whereas our CPAchecker⁺ solved all. In comparison of the efficiency of the employed reuse techniques, the speedups of eVolCheck and CPAchecker⁺ are 75.2 and 89.7 percent, respectively. These results demonstrate the efficiency of our summary reuse technique.

6.4.2 Comparison With UAutomizer

This experiment was conducted on the real-world benchmark, too. Excluding the programs that UAutomizer fails to parse, there are totally 1,177 verification tasks that belong to 90 driver/specification pairs.⁴

Note that the adopted verification techniques are very different in these two tools: UAutomizer⁺ uses the trace abstraction, while our CPAchecker⁺ uses the predicate abstraction. Moreover, the regression verification techniques implemented in these two tools are also different: one attempts to reuse the previously generated Floyd-Hoare automata [31], while another attempts to reuse the previously generated state-based summaries. To compare the efficiency of their adopted regression verification techniques, we compare the speedups of these two tools (with reuse over without reuse).

The comparison results are listed in Table 5. Note that UAutomizer⁺ implements two reuse strategies, i.e., *Eager* and *Lazy*. Results for both strategies are reported. From Table 5, CPAchecker⁺ achieves a speedup of 90.8 percent,

³ A successor version of this tool was recently released at: <http://verify.inf.usi.ch/uprover>

⁴ In their original paper [31], the UAutomizer⁺ was evaluated on the same set of programs.

TABLE 3
Overall Experimental Results on Artificial Revisions

Run set Device	LoC	#T	T_{1st}	no Reuse			Reuse			Speed -up	Avg FSize	RSR
				#solved	#abs_succ	T_{rv1}	#solved	#abs_succ	T_{rv2}			
iuu_phoenix	10258	5	214.5	5	136306	202.2	5	105	2.1	99.0%	4.3	0.9
broadcom1	4931	5	75.6	5	244123	72.0	5	77	1.1	98.5%	2.3	0.9
sungem_phi	5287	5	184.3	5	249324	115.2	5	270	2.0	98.3%	3.6	0.9
paride-bpck	9896	5	89.4	5	85737	89.1	5	106	1.7	98.1%	3.8	0.9
rtl2830	4457	5	111.9	5	16172	107.9	5	69	2.3	97.9%	2.4	0.9
mwifiex_pcie	11367	5	95.1	5	128217	99.0	5	164	2.3	97.6%	7.2	0.9
paride-epat	7695	5	106.1	5	22426	103.4	5	271	2.7	97.4%	2.9	0.8
tty-serial-mfd	10237	5	67.1	5	135512	66.4	5	120	1.9	97.2%	9.3	1.0
paride-kbic	8009	5	77.9	5	88482	74.9	5	190	2.2	97.1%	3.0	0.9
tpm_nsc	4791	5	35.5	5	10807	36.2	5	69	1.1	97.1%	2.7	0.9
mxl111sf-demod	8288	5	32.2	5	24957	31.7	5	41	1.0	96.7%	3.0	0.9
paride-on26	17745	5	134.5	5	80637	137.1	5	443	5.0	96.4%	2.7	0.9
dib3000mb	7868	5	120.3	5	26216	123.1	5	447	4.6	96.2%	4.0	0.9
paride-on20	7747	5	40.9	5	20099	41.3	5	106	1.6	96.1%	1.5	0.9
paride-dstr	6604	5	34.3	5	16655	33.8	5	84	1.4	95.8%	2.4	0.9
cmd640	6406	5	38.5	5	36496	37.7	5	93	1.6	95.6%	4.7	0.9
it913x	5947	5	69.1	5	14337	53.6	5	82	2.4	95.6%	3.0	0.9
serqt_usb21	10282	5	56.8	5	53423	56.4	5	166	2.6	95.3%	7.3	1.0
tuners-qt1010	7043	5	54.1	5	2645	47.9	5	79	2.3	95.3%	2.0	0.8
cfrag12864b	1493	5	15.0	4	5278	14.8	5	13	0.6	95.1%	8.8	1.0
paride-comm	5408	5	24.9	5	13211	23.8	5	77	1.2	94.9%	2.5	0.9
cp210x	7113	5	186.9	5	274622	166.8	4	1526	10.8	94.8%	14.6	0.9
paride-ktti	2886	5	15.1	5	6299	15.3	5	52	0.8	94.7%	1.5	0.8
paride-friq	8979	5	40.8	5	12680	40.1	5	146	2.2	94.6%	2.9	0.9
google-gsmi	6826	5	20.8	5	5294	19.9	5	48	1.1	94.3%	3.8	0.9
spcp8x5	6913	5	29.6	5	17396	23.3	5	50	1.3	94.3%	3.5	1.0
i2c-diolan	5308	5	30.2	5	13085	30.8	5	200	1.8	94.1%	3.4	0.8
sil164	6298	5	16.8	5	6906	16.6	5	66	1.0	94.0%	3.3	0.9
metro-usb	6288	5	25.3	5	20810	25.0	5	73	1.5	94.0%	2.4	0.9
paride-fit3	4459	5	19.5	5	10691	19.7	5	72	1.2	93.9%	1.7	0.9
wm831x-ldo	3720	5	19.9	5	15462	18.7	5	129	1.2	93.3%	4.0	0.9
paride-fit2	2593	5	13.6	5	5301	13.1	5	50	0.9	93.3%	1.5	0.9
sdricooh_cs	1633	5	12.4	5	7725	13.2	5	38	0.9	93.2%	3.5	1.0
tdo24m	5039	5	30.9	5	12666	26.7	5	121	1.9	93.0%	2.5	0.8
hid-zydacron	3211	5	10.6	5	2855	10.2	5	40	0.7	92.9%	2.0	0.9
max8903	1736	5	10.2	5	2984	10.3	5	27	0.7	92.8%	1.4	0.9
paride-epia	5518	5	28.1	5	10999	27.5	5	187	2.0	92.8%	2.8	0.9
wm831x-dcdc	3420	5	21.7	5	18191	22.7	5	149	1.6	92.8%	4.2	1.0
paride-frpw	8046	5	33.2	5	9374	33.4	5	182	2.6	92.2%	3.0	0.9
leds-lp5521	5443	5	6.4	5	526	6.4	5	12	0.6	91.4%	1.9	1.0
⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮	⋮
vp3054-i2c1	5582	5	0.6	5	37	0.6	5	17	0.9	-40.1%	0.5	0.9
atlas_btbs1	3599	5	0.6	5	30	0.6	5	20	0.8	-40.7%	0.5	0.9
girbil-sir1	5260	5	0.8	5	29	0.8	5	25	1.1	-42.1%	0.5	0.9
sbc8360	2052	5	0.6	5	97	0.6	5	29	0.9	-45.5%	1.1	0.9
vivopay	3793	5	0.5	5	11	0.5	5	5	0.8	-49.0%	0.4	0.8
epx_c31	2435	5	0.6	5	115	0.7	5	50	1.0	-49.3%	1.0	0.9
vgg2432a41	4136	5	0.7	5	41	0.6	5	33	0.9	-50.7%	0.5	0.9
siemens_mpi	3788	5	0.5	5	11	0.5	5	5	0.7	-54.6%	0.4	0.8
icplus1	4968	5	0.5	5	20	0.5	5	12	0.8	-76.3%	0.3	1.0
vp7045	8254	5	10.8	5	1407	11.8	5	270	26.2	-121.9%	61.3	0.9
Sum	1828793	2130	4375.5	2119	13.8M	20.9K	2129	247847	3.3K	-	-	-
Avg	4293	5	10.3	5	32.5K	49.1	5	581.8	7.7	84.2%	3.1	0.9

and UAutomizer⁺ gets a speedup of 82.8 percent or 81.4 percent. This result demonstrates the efficiency of our summary reuse technique. Note that we cannot conclude the superiority of our reuse technique over UAutomizer⁺ from this result, since they are used in different verification frameworks.

6.5 Impact of Reuse Strategies (RQ2)

In the former two experiments, both summary reuse and precision reuse were enabled for our approach. In this experiment, we switch off “precision reuse” and “summary

TABLE 4
Comparison of Our Approach With eVolCheck

	#computable	T_{rv1}	T_{rv2}	Speedup
eVolCheck	143	2382.7	591.9	75.2%
CPAchecker ⁺	244	2074.2	213.2	89.7%

TABLE 5
Comparison of Our Approach With UAutomizer⁺

	Speedup
CPAchecker ⁺	90.8%
UAutomizer ⁺ -Eager	82.8%
UAutomizer ⁺ -Lazy	81.4%

TABLE 6
Results on Different Reuse Strategies

	T_{rv}	#solved	Mem
no Reuse	20898.8	2119	589
Precision Reuse	73.5%	2125	151
Summary Reuse	69.5%	2127	180
Both Reuse	84.2%	2129	137

reuse” respectively, and evaluate the efficiency of our approach under different reuse strategies, i.e., “no Reuse”, “Precision Reuse”, “Summary Reuse” and “both Reuse”. This experiment was conducted on the second benchmark.

Table 6 shows the results of this experiment. Every row sums up results of all 2,130 regression verification tasks. For each row, we report the total regression verification time “ T_{rv} ” (in seconds), the total number of solved regression tasks “#solved”, and the average memory usage “Mem” (in gigabytes). Note that we only list the time usage in “no Reuse” row. For the other three rows, we show the speedup against “no Reuse” approach.

From Table 6, we found that every reuse technique outperforms “no Reuse”. They are not surprising since the precision reuse can significantly reduce the CEGAR iterations [7], and thus cuts down the verification time (73.5 percent speedup); and the summary reuse can save the repeated computation of summaries, and thus also reduces the verification time (69.5 percent speedup). Moreover, “Summary Reuse” solves 2 more tasks than “Precision Reuse”, illustrating that the former technique is more robust than the latter one. Precision reuse and summary reuse are two orthogonal techniques. By integrating these two techniques, “both Reuse” get the best performance, not only in analysis time (84.2 percent speedup), but also in the number of verified regression tasks.

The “Mem” column shows that all reuse strategies save the memory usage meetly. Again, “both Reuse” saves the most on memory consumption.

6.6 Impact of Summary Types (RQ3)

This experiment investigates the efficiency of our approach on different types of summaries. We evaluate the efficiency of our approach with loop summaries reused, procedure summaries reused and all summaries reused, respectively. Note that “precision reuse” and “lazy counterexample analysis” are switched off in this experiment.

We accumulate the analysis time on different summary types. Results are illustrated in Fig. 7, where the X-axis

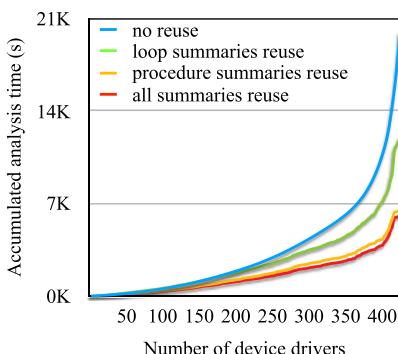
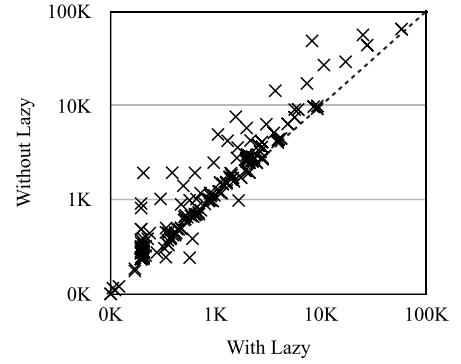
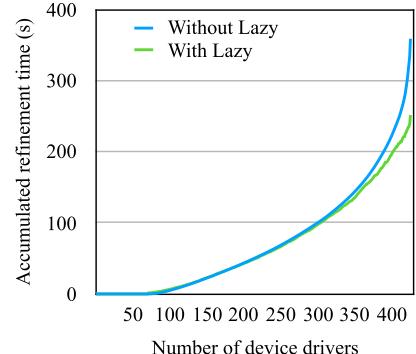


Fig. 7. Accumulating the analysis time on different summary types.



(a) Accumulated counterexample length



(b) Accumulated refinement Time

Fig. 8. Performance of summary reuse with and without *lazy counterexample analysis*.

indicates the number of device drivers, and the Y-axis represents the accumulated analysis time. From Fig. 7, we observed that “Procedure Summaries” outperforms “Loop Summaries”, and “All Summaries Reuse” performs the best. The main reason is that our benchmark contains fewer loop statements than procedures (1,273 versus 11,417).

6.7 Impact of Lazy Counterexample Analysis (RQ4)

The final experiment evaluates the efficiency of lazy counterexample analysis. Note that this technique is mainly relative to the refinement process, we measure the refinement time and counterexample length in this experiment.

Fig. 8a shows the results on counterexample length in a scatter diagram. Note that each device driver may involve several regression verification tasks, and each regression verification task may require many CEGAR iterations. The reported counterexample length is the accumulated length of all counterexamples generated in all CEGAR iterations among all regression verification tasks of a device driver. In Fig. 8a, each point represents the accumulated counterexample length of a device driver, the Y and X axes indicate our approach with and without lazy counterexample analysis, respectively. Both X and Y axes are logarithmic. A point below the reference line indicates a case where the lazy counterexample analysis is beneficial.

Fig. 8b show results on refinement time. Again, the reported refinement time is the accumulation of time spent on all refinement iterations among all regression verification tasks of all the currently-tested device drivers. In Fig. 8b, the X-axis catalogs the number of device drivers and the Y-axis shows the accumulated refinement time.

On the whole, the lazy counterexample analysis performs better on refinement, saving about 29.9 percent of time. It is also observed that for 415 of 426 device drivers, the corresponding data points in Fig. 8a are above the reference line, indicating that the accumulated counterexample lengths were reduced with this technique. These results conform to our algorithmic analysis in Section 5.3. With the lazy analysis technique, the counterexample is not necessarily to be fully expanded. And a shorter counterexample can usually reduce the refinement efforts.

7 RELATED WORK

Regression verification was investigated mainly in two directions, the verification of differences, and the reuse of previously computed results. We also discuss the summarization and symbolic execution techniques in this section.

Verification of Differences. In this line of research, one attempts to establish the correctness of the new program by proving its (conditional) equivalence to an old and verified program.

Many techniques have been proposed in this line of research. The technique for proving conditional equivalence of two programs by abstraction and decomposition of procedures is proposed in [4], [32]. Backes *et al.* [5] proposed to distinguish the program behaviors that are impacted by the changes. Only the impacted program behaviors needed to be considered during the regression verification. Beyer *et al.* [33] proposed the conditional model checking, which outputs a condition such that the program satisfies the specification under this condition. Böhme *et al.* [6] proposed a partition-based regression verification technique. Instead of proving the absence of regression errors for the entire input space, this approach continuously verifies the input space in a gradual manner. Felsing *et al.* [34] reduced the equivalence proving of two related imperative integer programs to Horn constraints over uninterpreted predicates, and then solved the constraints using an Horn solver.

Moreover, Rungta *et al.* [35] presented a technique for interprocedural change impact analysis. Yang *et al.* [36] introduced an incremental approach for checking the conformance of code against different properties. Trostanetski *et al.* [37] analyzed the semantic difference between successive revisions. Mora *et al.* [38] performed modular symbolic execution to prove the equivalence between different versions of libraries with respect to the same parts of codebase (client program).

Reuse of Intermediate Results. In this line of research, one studies the reuse of previously-generated results to the current verification. A variety of information has been proposed for reuse.

Some researchers [9], [39], [40], [41] proposed to keep the reached state space and reuse them in the further verification runs. The rationale of these techniques is that state spaces of consecutive versions tend to be similar. However, recording and reusing reached state space may be costly, and these techniques may not be applicable to large-scale programs. For example, [40] points out 6 times more on memory usage in the worst case. The lines of code of single revision of [40] are less than 1,000.

Visser *et al.* [10] noticed the importance of constraint solving for symbolic execution. They proposed to cache and reuse the results of constraint solving. This approach was further improved in [42], [43] from different aspects. This group of techniques is orthogonal to our approach. These techniques can be applied to enhance our approach.

Beyer *et al.* [7] proposed to use abstract precisions as the intermediate results. An abstract precision defines the level of abstraction, which conveys important information on the current verification. They proposed to record the final abstract precision and to reuse it as the initial abstract precision of the current verification. With this technique, the number of refinements can often be reduced. Note that the precision reuse and our summary reuse are orthogonal to each other. It is possible to combine these two reuse techniques together. We have already combined this technique with ours. The combined technique shows a very promising performance.

Fedyukovich *et al.* [44] offered a regression verification technique for checking property directed equivalence. The safe inductive invariants across program transformations were migrated and established. Rothenberg *et al.* [31] proposed to reuse the sequence of Floyd-Hoare automata learned during the trace abstraction. Two reuse strategies, eagerly and lazily, were developed in this paper. This technique has been realized in UAutomizer, a well-known software verification tool.

The work most relevant to ours is [11], [30], where a regression verification technique by means of interpolation-based procedure summaries was proposed. Our idea of summary reuse was inspired by these two papers. The main difference lies in the way in which the summaries are constructed during the verification. In [11], [30], the authors use a logical formula φ_A to encode the behaviors of the procedure ϱ , and another logical formula φ_B to encode its calling context. Then they compute the interpolation of φ_A and φ_B and use that as the summary of ϱ . In contrast, we use the abstract states in predicate analysis to construct the program summaries. Each summary in our paper consists of an entry state and a set of exit states of ϱ . Our state-based summaries can be completely integrated into the framework of CEGAR. All ingredients of a state-based summary are by-products of CEGAR. There needs no additional computation for generating this kind of summaries. Experimental comparison in Section 6.4 demonstrates the practicability of our approach.

Pastore *et al.* [45] proposed a method to validate that an already tested code has not been broken by an upgrade. It maintains a test suite that can be used to revalidate the software as it evolves. Different from our approach, this technique is respect to regression testing. The verification technique is used there, as an aid, to validate dynamic properties (or invariants). In contrast, we aim to provide a new regression verification technique via reusing summaries.

Summarization. In this line of research, one tries to replace program fragments with summaries. A summary can usually be represented as an input-output pair of a program fragment. Procedure summaries have been long studied and there are also many studies on loop summaries recently.

Many researchers [11], [30], [46], [47] proposed to use interpolation-based method to generate procedure summaries. [47] combines function summaries with the expressiveness of satisfiability modulo theories (SMT), which makes summaries smaller and more human-readable. [11], [30], [46] implement a procedure summarization approach for software bounded model checking, and uses interpolation-based procedure summaries as over-approximation of procedure calls.

Kroening *et al.* [48] proposed the idea to substitute a loop with a conservative abstraction of its behavior, constructing abstract transformers for nested loops starting from the innermost loop. They also applied this method in termination analysis. Seghir *et al.* [49] used various inference rules for deriving summaries based on control structures. However, this approach can only compute precise loop summaries for restricted classes of programs depending on inference rules. Xie *et al.* [50] proposed a general framework for summarizing multi-path loops. It classifies loops according to the patterns of values changes in path conditions and the interleaving of paths within the loop. A disjunctive summarization is constructed for all the feasible executions in the loop. Different from our method, [50] cannot summarize loops containing non-induction variables, array variables, and nested loops. Godefroid *et al.* [51] investigated an alternative approach based on automatic loop-invariant generation. This approach can (partially) summarize a loop body during a single dynamic symbolic execution, which can ease the path explosion in dynamic test generation.

Symbolic Execution. In recent years, a great deal of effort has been focused on regression symbolic execution, which takes advantage of the previous analysis of symbolic execution to speedup the current analysis.

Person *et al.* [52] used a form of overapproximating symbolic execution to skip portions of the program that are provably identical across the versions. In [53], Person *et al.* presented a regression symbolic execution technique for Java programs, based on the Symbolic PathFinder. It analyzes the CFAs of two program versions, computes the locations affected by the program changes, and then applies the symbolic execution to the affected code only. Further more, Guo *et al.* [54] investigated the symbolic execution technique for multi-threaded programs.

8 CONCLUSION

We proposed in this paper a fully automatic regression verification technique in the context of CEGAR. Abstract summaries are reused across different abstract precisions and different program revisions. We proposed a unified framework for reusing both procedure summaries and loop summaries. A lazy counterexample analysis algorithm was further proposed to reduce the unnecessary path expansion efforts. We implemented our approach in the software verification tool CPAchecker. Experimental results show the promising performance of our technique.

ACKNOWLEDGMENT

This work was supported in part by the National Natural Science Foundation of China (61672310, 61527812), the National Key R&D Program of China (2018YFB1308601).

REFERENCES

- [1] V. D’silva, D. Kroening, and G. Weissenbacher, “A survey of automated techniques for formal software verification,” *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 27, no. 7, pp. 1165–1178, Jul. 2008.
- [2] T. L. Graves, M. J. Harrold, J.-M. Kim, A. Porter, and G. Rothermel, “An empirical study of regression test selection techniques,” *ACM Trans. Softw. Eng. Methodol.*, vol. 10, no. 2, pp. 184–208, 2001.
- [3] G. Rothermel and M. J. Harrold, “Analyzing regression test selection techniques,” *IEEE Trans. Softw. Eng.*, vol. 22, no. 8, pp. 529–551, Aug. 1996.
- [4] B. Godlin and O. Strichman, “Regression verification,” in *Proc. 46th Annu. Des. Autom. Conf.*, 2009, pp. 466–471.
- [5] J. Backes, S. Person, N. Rungta, and O. Tkachuk, “Regression verification using impact summaries,” in *Proc. Int. SPIN Workshop Model Checking Softw.*, 2013, pp. 99–116.
- [6] M. Böhme, B. C. D. S. Oliveira, and A. Roychoudhury, “Partition-based regression verification,” in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 302–311.
- [7] D. Beyer, S. Löwe, E. Novikov, A. Stahlbauer, and P. Wendler, “Precision reuse for efficient regression verification,” in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, pp. 389–399.
- [8] S. K. Lahiri, K. L. McMillan, R. Sharma, and C. Hawblitzel, “Differential assertion checking,” in *Proc. 9th Joint Meeting Found. Softw. Eng.*, 2013, pp. 345–355.
- [9] T. A. Henzinger, R. Jhala, R. Majumdar, and M. A. Sanvido, “Extreme model checking,” in *Verification: Theory and Practice*. Berlin, Germany: Springer, 2003, pp. 332–358.
- [10] W. Visser, J. Geldenhuys, and M. B. Dwyer, “Green: Reducing, reusing and recycling constraints in program analysis,” in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, Art. no. 58.
- [11] O. Sery, G. Fedyukovich, and N. Sharygina, “Incremental upgrade checking by means of interpolation-based function summaries,” in *Proc. Formal Methods Comput.-Aided Des.*, 2012, pp. 114–121.
- [12] M. Sharir and A. Pnueli, *Two Approaches to Interprocedural Data Flow Analysis*. New York, NY, USA: New York Univ. Comput. Sci. Dept., 1978.
- [13] T. Reps, S. Horwitz, and M. Sagiv, “Precise interprocedural data-flow analysis via graph reachability,” in *Proc. 22nd ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, 1995, pp. 49–61.
- [14] Y. Xie and A. Aiken, “Saturn: A scalable framework for error detection using boolean satisfiability,” *ACM Trans. Program. Lang. Syst.*, vol. 29, no. 3, 2007, Art. no. 16.
- [15] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, “Counterexample-guided abstraction refinement,” in *Proc. Int. Conf. Comput. Aided Verification*, 2000, pp. 154–169.
- [16] D. Beyer, “Reliable and reproducible competition results with BenchExec and witnesses (report on SV-COMP 2016),” in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2016, pp. 887–904.
- [17] D. Beyer and M. E. Keremoglu, “CPAchecker: A tool for configurable software verification,” in *Proc. Int. Conf. Comput. Aided Verification*, 2011, pp. 184–190.
- [18] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, “The software model checker BLAST,” *Int. J. Softw. Tools Technol. Transfer*, vol. 9, no. 5/6, pp. 505–525, 2007.
- [19] M. Heizmann, J. Hoenicke, and A. Podelski, “Software model checking for people who love automata,” in *Proc. Int. Conf. Comput. Aided Verification*, 2013, pp. 36–52.
- [20] M. Carter, S. He, J. Whitaker, Z. Rakamarić, and M. Emmi, “SMACK software verification toolchain,” in *Proc. 38th IEEE/ACM Int. Conf. Softw. Eng. Companion*, 2016, pp. 589–592.
- [21] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.
- [22] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, “Lazy abstraction,” *ACM SIGPLAN Notices*, vol. 37, no. 1, pp. 58–70, 2002.
- [23] D. Beyer, T. A. Henzinger, and G. Théoduloz, “Configurable software verification: Concretizing the convergence of model checking and program analysis,” in *Proc. Int. Conf. Comput. Aided Verification*, 2007, pp. 504–518.
- [24] S. Graf and H. Saïdi, “Construction of abstract state graphs with PVS,” in *Proc. Int. Conf. Comput. Aided Verification*, 1997, pp. 72–83.
- [25] K. L. McMillan, “Interpolation and SAT-based model checking,” in *Proc. Int. Conf. Comput. Aided Verification*, 2003, pp. 1–13.
- [26] K. L. McMillan, “Lazy abstraction with interpolants,” in *Proc. Int. Conf. Comput. Aided Verification*, 2006, pp. 123–136.

- [27] C. Baier and J.-P. Katoen, *Principles of Model Checking*. Cambridge, MA, USA: MIT Press, 2008.
- [28] D. Beyer, "Software verification with validation of results (report on SV-COMP 2017)," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2017, pp. 331–349.
- [29] Y. Jia and M. Harman, "MILU: A customizable, runtime-optimized higher order mutation testing tool for the full C language," in *Proc. Testing: Academic Ind. Conf. Practice Res. Techn.*, 2008, pp. 94–98.
- [30] G. Fedyukovich, O. Sery, and N. Sharygina, "eVolCheck: Incremental upgrade checker for C," in *Proc. 19th Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2013, pp. 292–307.
- [31] B.-C. Rothenberg, D. Dietsch, and M. Heizmann, "Incremental verification using trace abstraction," in *Proc. Int. Static Anal. Symp.*, 2018, pp. 364–382.
- [32] S. Chaki, A. Gurkinkel, and O. Strichman, "Regression verification for multi-threaded programs," in *Proc. Int. Workshop Verification Model Checking Abstract Interpretation*, 2012, pp. 119–135.
- [33] D. Beyer, T. A. Henzinger, M. E. Keremoglu, and P. Wendler, "Conditional model checking: A technique to pass information between verifiers," in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, Art. no. 57.
- [34] D. Felsing, S. Grebing, V. Klebanov, P. Rümmer, and M. Ulbrich, "Automating regression verification," in *Proc. 29th ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2014, pp. 349–360.
- [35] N. Rungta, S. Person, and J. Branchaud, "A change impact analysis to characterize evolving program behaviors," in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, 2012, pp. 109–118.
- [36] G. Yang, S. Khurshid, S. Person, and N. Rungta, "Property differencing for incremental checking," in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1059–1070.
- [37] A. Trostanetski, O. Grumberg, and D. Kroening, "Modular demand-driven analysis of semantic difference for program versions," in *Proc. Int. Static Anal. Symp.*, 2017, pp. 405–427.
- [38] F. Mora, Y. Li, J. Rubin, and M. Chechik, "Client-specific equivalence checking," in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 441–451.
- [39] S. Lauterburg, A. Sobeih, D. Marinov, and M. Viswanathan, "Incremental state-space exploration for programs with dynamically allocated data," in *Proc. 30th Int. Conf. Softw. Eng.*, 2008, pp. 291–300.
- [40] G. Yang, M. B. Dwyer, and G. Rothermel, "Regression model checking," in *Proc. IEEE Int. Conf. Softw. Maintenance*, 2009, pp. 115–124.
- [41] C. L. Conway, K. S. Namjoshi, D. Dams, and S. A. Edwards, "Incremental algorithms for inter-procedural analysis of safety properties," in *Proc. Int. Conf. Comput. Aided Verification*, 2005, pp. 449–461.
- [42] A. Aquino, F. A. Bianchi, M. Chen, G. Denaro, and M. Pezzè, "Reusing constraint proofs in program analysis," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 305–315.
- [43] X. Jia, C. Ghezzi, and S. Ying, "Enhancing reuse of constraint solutions to improve symbolic execution," in *Proc. Int. Symp. Softw. Testing Anal.*, 2015, pp. 177–187, doi: 10.1145/2771783.2771806.
- [44] G. Fedyukovich, A. Gurkinkel, and N. Sharygina, "Property directed equivalence via abstract simulation," in *Proc. Int. Conf. Comput. Aided Verification*, 2016, pp. 433–453.
- [45] F. Pastore et al., "Verification-aided regression testing," in *Proc. Int. Symp. Softw. Testing Anal.*, 2014, pp. 37–48.
- [46] O. Sery, G. Fedyukovich, and N. Sharygina, "FunFrog: Bounded model checking with interpolation-based function summarization," in *Proc. Int. Symp. Automated Technol. Verification Anal.*, 2012, pp. 203–207.
- [47] L. Alt et al., "HiFrog: SMT-based function summarization for software verification," in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2017, pp. 207–213.
- [48] D. Kroening, N. Sharygina, S. Tonetta, A. Tsitovich, and C. M. Wintersteiger, "Loop summarization using state and transition invariants," *Formal Methods Syst. Des.*, vol. 42, no. 3, pp. 221–261, 2013.
- [49] M. N. Seghir, "A lightweight approach for loop summarization," in *Proc. Int. Symp. Automated Technol. Verification Anal.*, 2011, pp. 351–365.
- [50] X. Xie, B. Chen, Y. Liu, W. Le, and X. Li, "Proteus: Computing disjunctive loop summary via path dependency analysis," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 61–72.
- [51] P. Godefroid and D. Luchaup, "Automatic partial loop summarization in dynamic test generation," in *Proc. Int. Symp. Softw. Testing Anal.*, 2011, pp. 23–33.
- [52] S. Person, M. B. Dwyer, S. Elbaum, and C. S. Psreanu, "Differential symbolic execution," in *Proc. 16th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2008, pp. 226–237.
- [53] S. Person, G. Yang, N. Rungta, and S. Khurshid, "Directed incremental symbolic execution," *ACM SIGPLAN Notices*, vol. 46, no. 6, pp. 504–515, 2011.
- [54] S. Guo, M. Kusano, and C. Wang, "Conc-iSE: Incremental symbolic execution of concurrent software," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2016, pp. 531–542.



Fei He received the BS degree in computer science and technology from the National University of Defense Technology, Changsha, China, in 2002, and the PhD degree in computer science and technology from Tsinghua University, Beijing, China, in 2008. He is currently an associate professor with the School of Software, Tsinghua University, Beijing, China. His research interests include formal verification and program analysis.



Qianshan Yu received the BS degree from Jilin University, Changchun, China, in 2017. He is currently working toward the PhD degree in the School of Software, Tsinghua University, Beijing, China. His research interests include software verification and regression verification.



Liming Cai received the BS degree from Xiamen University, Xiamen, China, in 2013, and the MS degree from Tsinghua University, Beijing, China, in 2016. He is currently an algorithm engineer in kwai tech.co. His research interests include formal methods and program verification.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

The Effectiveness of Supervised Machine Learning Algorithms in Predicting Software Refactoring

Maurício Aniche^{ID}, Erick Maziero, Rafael Durelli^{ID}, and Vinicius H. S. Durelli

Abstract—Refactoring is the process of changing the internal structure of software to improve its quality without modifying its external behavior. Empirical studies have repeatedly shown that refactoring has a positive impact on the understandability and maintainability of software systems. However, before carrying out refactoring activities, developers need to identify refactoring opportunities. Currently, refactoring opportunity identification heavily relies on developers' expertise and intuition. In this paper, we investigate the effectiveness of machine learning algorithms in predicting software refactorings. More specifically, we train six different machine learning algorithms (i.e., Logistic Regression, Naive Bayes, Support Vector Machine, Decision Trees, Random Forest, and Neural Network) with a dataset comprising over two million refactorings from 11,149 real-world projects from the Apache, F-Droid, and GitHub ecosystems. The resulting models predict 20 different refactorings at class, method, and variable-levels with an accuracy often higher than 90 percent. Our results show that (i) Random Forests are the best models for predicting software refactoring, (ii) process and ownership metrics seem to play a crucial role in the creation of better models, and (iii) models generalize well in different contexts.

Index Terms—Software engineering, software refactoring, machine learning for software engineering

1 INTRODUCTION

REFACTORING, as defined by Fowler [1] is “*the process of changing a software system in such a way that does not alter the external behavior of the code yet improves its internal structure*”. Over the years, empirical studies have established a positive correlation between refactoring operations and code quality metrics (e.g., [2], [3], [4], [5], [6]). All these evidence indicates that refactoring should be regarded as a first-class concern of software developers.

However, *deciding when and what (as well as understanding why) to refactor* have long posed a challenge to developers. Software development teams should not simply refactor their software systems at will, or decide not to refactor a piece of code that causes technical debt, as any refactoring activity comes with costs [7], [8].

To that aim, software developers have been relying more and more on different static analysis tools and linters as a way to collect feedback about their source code [9]. Developers not only use these tools to find bug-related issues in their systems (e.g., [10], [11]), but also for code quality-related advice [12], [13]. Popular tools such as PMD, ESLint, and Sonarqube offer detection strategies for common code smells, such as *God*

Classes or Long Methods. These tools have been now integrated into different stages of the developers' workflow, e.g., inside IDEs (e.g., PMD's plugin for IntelliJ or Eclipse), during code review (by means of bots), or as a overall quality report (for example, Sonarqube's Technical Debt report).

Identifying refactoring opportunities is an important stage that precedes the refactoring process. However, despite their importance to the software development world, the state-of-the-art tools that developers have been using to get refactoring recommendations often present a high number of false positives [14], making developers to lose their confidence on them. The tools' detection strategies are often either based on hard thresholds of single metrics (e.g., PMD considers all methods with more than 100 lines of code, “problematic”), or on Lanza's and Marinescu's seminal work on code smells detection strategies [15] which rely on a combination of code metrics and thresholds.

While tools provide some degree of customization, e.g., PMD lets developers choose their own thresholds, and Decor [16] enables developers to devise their own code smells detection strategies, such hand-made detection strategies may be too simplistic to capture the full complexity of software systems. This is where we conjecture a ML-based solution would help. *We argue that the task of identifying relevant refactoring opportunities, which currently heavily relies on developers' expertise and intuition, should be supported by sophisticated recommendation algorithms.*

Researchers have been indeed experimenting with different AI-based techniques to recommend refactoring, such as the use of search algorithms [17], [18], and pattern mining [19]. In this paper, we explore how machine learning (ML) can be harnessed to predict refactoring operations. ML algorithms have been showing promising results when

- Maurício Aniche is with the Delft University of Technology, 2628, CD, Delft, The Netherlands. E-mail: m.f.aniche@tudelft.nl.
- Erick Maziero and Rafael Durelli are with the Federal University of Lavras, Lavras, MG 37200-900, Brazil. E-mail: {erick.maziero, rafael.durelli}@ufla.br.
- Vinicius H. S. Durelli is with the Federal University of São João del Rei, São João del Rei, MG 36307-352, Brazil. E-mail: durelli@ufrj.edu.br.

Manuscript received 10 Jan. 2020; revised 6 July 2020; accepted 1 Sept. 2020. Date of publication 4 Sept. 2020; date of current version 18 Apr. 2022. (Corresponding author: Maurício Aniche.) Recommended for acceptance by M. Kim. Digital Object Identifier no. 10.1109/TSE.2020.3021736

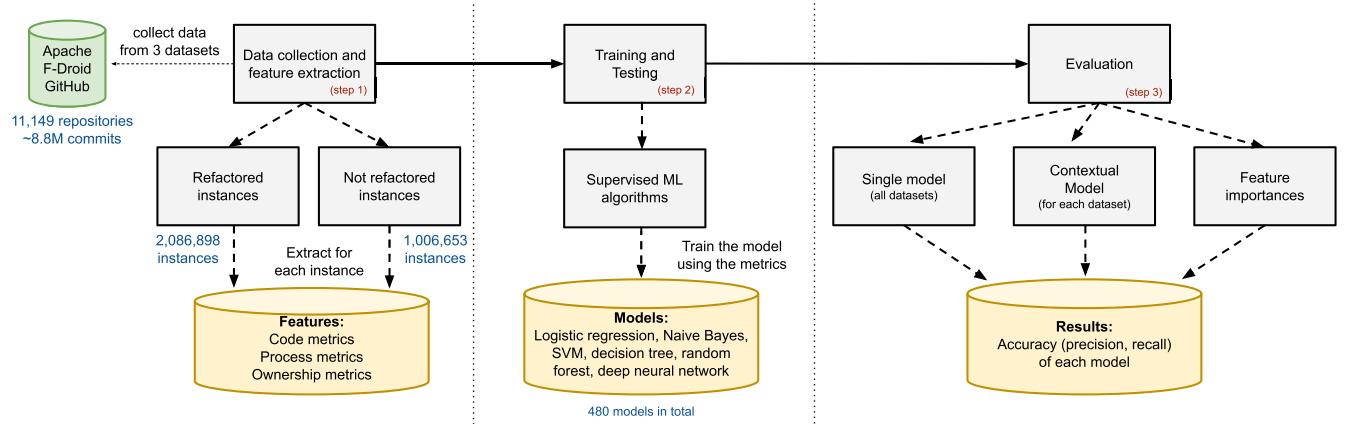


Fig. 1. Overview of the research methodology.

applied to different areas of software engineering, such as defect prediction [20], code comprehension [21], and code smells [22]. By learning from classes and methods that underwent refactoring operations in practice, we surmised that the resulting models would be able to provide more reliable refactoring recommendations to developers.

We formulate the prediction of refactoring opportunities as a binary classification problem. We build models that recommend several different refactoring operations (the full list of refactoring operations is shown in Table 2). Each model predicts whether a given piece of code should undergo a specific refactoring operation. For instance, given a method, the *Extract Method* model predicts whether that method should undergo a extract method refactoring operation. More formally, given a set R of possible refactorings for a source code element, we learn a set of models $M_r(e)$, $r \in R$, that predict whether a source code element e should be refactored by means of refactoring operation r .

To probe into the effectiveness of supervised ML algorithms in predicting refactoring opportunities, we apply six ML algorithms (i.e., Logistic Regression, Naive Bayes, Support Vector Machines, Decision Trees, Random Forest, and Neural Network) to a dataset containing more than two million labelled refactoring operations that happened in 11,149 open-source projects from the Apache, F-Droid, and GitHub ecosystems. The resulting models are able to predict 20 different refactoring operations at class, method, and variable-levels [1], with an average accuracy often higher than 90 percent.

Understanding the effectiveness of the different models is the *first and necessary step in building tools that will help developers in drawing data-informed refactoring decisions*. This paper provides the *first solid large-scale evidence that ML algorithms can model the refactoring recommendation problem accurately*.

In summary, this paper makes the following contributions:

- i) A large-scale in-depth study of the effectiveness of different supervised ML algorithms to predict software refactoring, showing that ML methods can accurately model the refactoring recommendation problem.
- ii) A dataset containing more than two million real-world refactorings extracted from more than 11 thousand real-world projects.

2 RESEARCH METHODOLOGY

The goal of this paper is to *evaluate the feasibility of using supervised ML algorithms to identify refactoring opportunities*. To this end, we framed our research around the following research questions (RQs):

RQ₁: How accurate are supervised ML algorithms in predicting software refactoring? In practice, some prediction algorithms perform better than others, depending on the task. In this RQ, we explore how accurate different supervised ML algorithms (i.e., Support Vector Machines, Naive Bayes, Decision Trees, Random Forest, and Neural networks) are in predicting refactoring opportunities at different levels (i.e., refactorings at class, method, and variable-levels), using Logistic Regression as a baseline for comparison.

RQ₂: What are the important features in the refactoring prediction models? Features (i.e., a numeric representation of a measurable property that is used to represent a ML problem to the model) play a pivotal role in the quality of the obtained models. In RQ₁, we build the models using all the features we had available (for a method-level refactoring, for example, we use 58 different features). In this RQ, we explore which features are considered the most relevant by the models. Such knowledge is essential because, in practice, models should be as simple as and require as little data as possible.

RQ₃: Can the predictive models be carried over to different contexts? Understanding whether refactoring prediction models should be trained specifically for a given context or whether it generalizes enough to different contexts can significantly reduce the cost of applying and re-training these models in practice. We set out to study whether prediction models, devised in one type of software systems (e.g., libraries and frameworks from the Apache ecosystem), are able to generalize to different types of software systems (e.g., mobile apps in the F-Droid ecosystem). We investigate the accuracy of predictive models against independent datasets (i.e., out-of-sample accuracy).

Fig. 1 shows an overview of the approach we used to answer the aforementioned RQs. Essentially, our approach is three-fold: (i) data collection and feature extraction, (ii) training and testing, and (iii) evaluation. These steps are outlined below and later better detailed in the following subsections.

TABLE 1
Overview of the Sample Used in Our Study

	Number of Projects	Total number of commits
Apache	844	1,471,203
F-Droid	1,233	814,418
GitHub	9,072	6,517,597
	11,149	8,803,218

The first step is centered around data preparation. This step involves mining software repositories for labelled instances of *refactored elements*, e.g., a method that was moved, or an inlined variable, and instances of *elements that were not refactored*. To both refactored and non-refactored instances, we extract code metrics (e.g., complexity and coupling), process metrics (e.g., number of commits in that class), and ownership metrics (e.g., number of authors). The code metrics are calculated at different levels, depending on the type of refactoring. For a class-level refactoring, we calculate class-level metrics; for a method-level refactoring, we calculate both class and method-level metrics; for a variable-level refactoring, we calculate class, method, and variable-level metrics.

In the second step, we use the examples of refactored and non-refactored elements we collected as training and testing data to different ML algorithms. We generate a model for each combination of datasets (all datasets together, Apache, F-Droid, and GitHub), refactoring operations (the 20 refactoring operations we show in Table 2), and ML algorithms (i.e., Logistic Regression, Naive Bayes, Support Vector Machines, Decision Tree, Random Forest, and Neural Network). Before training the final model, our pipeline balances the dataset, performs a random search for the best hyperparameters, and stores the best configuration and the ranking of importance of each feature.

In the third step, we evaluate the accuracy of each generated model. First, we test the model using single datasets. Next, we test the models that were trained using data from just one dataset and test it in all the other datasets (e.g., the model trained with the Apache dataset is tested on the GitHub and F-Droid datasets). In all the runs, we record the model's precision, recall, and accuracy.

2.1 Experimental Sample

We selected a very large and representative set of Java projects from three different sources:

- The Apache Software Foundation (ASF) is a non-profit organization that supports all Apache software projects. The ASF is responsible for projects such as Tomcat, Maven, and Ant. Our tools successfully processed 844 out of their 860 Java-based projects. We discuss why the processing of some projects have failed in Section 2.6.
- F-Droid is a software repository of Android mobile apps. The repository contains only free software apps. Our tools successfully processed 1,233 out of their 1,352 projects.
- GitHub provides free hosting for open source projects. GitHub has been extensively used by the open source community. As of May 2019, GitHub has 37 million users registered. We collected the first 10,000

most starred Java projects. Note that ASF and F-Droid projects might also exist in GitHub; we removed duplicates. In the end, our tools were able to process 9,072 projects.

The three different sources of projects provide the dataset with high variability in terms of size and complexity of projects, domains and technologies used, and community. The resulting sample can be seen in Table 1. It comprises the 11,149 projects (844 from Apache, 1,233 from F-Droid, and 9,072 from GitHub). These projects together a history of 8.8 million commits, measured at the moment of data collection, in March of 2019.

2.2 Extraction of Labelled Instances

In a nutshell, our data collection process happens in three phases. In the first phase, the tool clones the software repository, uses RefactoringMiner [23] to collect refactoring operations that happened throughout the history of the repository, and collects the code metrics of the refactored classes. In the second phase, where all the refactoring operations and their respective files are already known, the tool then collects the process and ownership metrics of the refactored classes. Finally, the tool collects instances of non-refactored classes (as well as their code, process, and ownership metrics).

For each project, we visit its entire master branch from the oldest to the most recent commit. For each commit, we invoke RefactoringMiner [23]. The tool can receive, as an input, a pair of commits. It then uses the *diff* between the two provided commits to identify refactoring operations that have happened.¹ We highlight that RefactoringMiner is the current state-of-the-art tool to identify refactoring operations, having the highest recall and precision rates (98 and 87 percent, respectively) among all currently available refactoring detection tools [23].

For each refactoring operation that is detected by RefactoringMiner, we extract code metrics of the refactored element in its version *before* the refactoring has been applied. The intuition behind using the version before the refactoring is that models should learn how to identify refactorings by looking at the elements as they were prior to being refactored. We collect the information at the precise level of the refactoring. For example, if the refactoring is at class-level, we collect all the class-level metrics related to the class under refactoring; if it is a method-level refactoring, we collect metric-level metrics related to the method under refactoring; the same applies for variable-level refactorings.

After all the refactorings were identified, our tool collects the process and ownership metrics of the refactored classes. These metrics are also collected at the version *before* the refactoring had been applied.

Finally, our tool collects instances of *non-refactored classes, methods, and variables*, i.e., code elements that did not undergo any refactoring operations, to serve as counterexamples to the model. This is a fundamental step as binary classification models should learn how to separate between the two classes; in this case, between methods that need to be refactored, and methods that do not need to be refactored.

¹ Given that we need a pair of commits in order to identify the refactoring operations, we skip the first commit of the repository, and start from commit no. 2.

TABLE 2
The 20 Refactoring Operations That are Studied in This Paper

Refactoring	Problem and Solution
Class-level refactorings	
Extract Class	A class performs the work of two or more classes. Create a class and move the fields and methods to it.
Extract Subclass	A class owns features that are used only in certain scenarios. Create a subclass.
Extract Super-class	Two classes own common fields and methods. Create a super class and move the fields and methods.
Extract Interface	A set of clients use the same part of a class interface. Move the shared part to its own interface.
Move Class	A class is in a package with non-related classes. Move the class to a more relevant package.
Rename Class	The class' name is not expressive enough. Rename the class.
Move and Rename Class	The two aforementioned refactorings together.
Method-level refactorings	
Extract Method	Related statements that can be grouped together. Extract them to a new method.
Inline Method	Statements unnecessarily inside a method. Replace any calls to the method with the method's content.
Move Method	A method does not belong to that class. Move the method to its rightful place.
Pull Up Method	Sub-classes have methods that perform similar work. Move them to the super class.
Push Down Method	The behavior of a super-class is used in few sub-classes. Move it to the sub-classes.
Rename Method	The name of a method does not explain the method's purpose. Rename the method.
Extract And Move Method	The two aforementioned refactorings together.
Variable-level refactorings	
Extract Variable	Hard-to-understand/long expression. Divide the expression into separate variables.
Inline Variable	Non-necessary variable holding an expression. Replace the variable references with the expression itself.
Parameterize Variable	Variable should be a parameter of the method. Transform variable into a method parameter.
Rename Parameter	The name of a method parameter does not explain its purpose. Rename the parameter.
Rename Variable	The name of a variable does not explain the variable's purpose. Rename the variable.
Replace Variable w/ Attribute	Variable is used in more than a single method. Transform the variable to a class attribute.

Definitions derived from Fowler [1].

Given that there is no clear way of extracting code elements that do not need to be refactored out of the source code history of software systems, we propose an heuristic: we consider a class to be a non-refactoring instance if it was modified (i.e., a change committed in the Git repository) precisely k times without a single refactoring operation being applied in between this time. The heuristic aims at identifying classes that can still be evolved by developers (as developers have been evolving them) without the need for a refactoring (as we see that they did not apply any refactorings). We conjecture that such classes can serve as good counterexamples for the model.

After experimentation, we set $k = 50$ (we discuss the influence of k in Section 4.1). The tool, therefore, collects all classes that were modified precisely k times and did not go through any refactoring operation. We then extract its source code, process, and ownership metrics. Note that we extract the metrics at time 0, and not at time k , as we want the models to learn from the code element that, back then, did not require any refactoring from developers. The same element can appear more than once in this dataset (although always with different metric values), as whenever we collect an instance of non-refactoring, we restart its counter and continue to visit the repository.

Note that our approach ignores test code (e.g., JUnit files) and only captures refactoring operations in production files. Test code quality has been the target of many studies (e.g., [24], [25], [26]). In this work, we assume that refactorings that happen in test code are naturally different from the ones that happen in production code; our future agenda includes the development of refactoring models for test code.

In Table 3, we show the number of refactored and non-refactored instances we collected per dataset. We highlight the fact that the number of instances varies per refactoring, which reflects how much developers apply each of these

refactorings. For example, the dataset contains around 327 thousand instances of *Extract Method*, but only 654 instances of *Move and Rename Class*. We see this as a positive point to our exploration, as the model will have to deal with refactorings where the number of instances is not high.

2.3 Feature Selection

We extract source code, process, and ownership metrics of all refactored and non-refactored instances. These three types of metrics have been proven useful in other prediction models in software engineering (e.g., [20], [27], [28]). Moreover, earlier studies based on the correlation between refactoring and code quality metrics postulated that an increase in the former leads to improvements in the latter (e.g., [3], [4], [6]).² Table 4 lists all the metrics we chose to train predictive models. In our online appendix [30], we show the distribution (i.e., descriptive statistics) of the values of each feature. The following subsections detail the source code, process, and code ownership metrics we collect.

Source Code Metrics. Features in this category are derived from source code attributes. We collect CK metrics [31] as they express the complexity of the element. More specifically, CBO, WMC, RFC, and LCOM. We also collect several different attributes of the element, e.g., number of fields, number of loops, number of return statements. These metrics are collected at class (37 metrics), method (20 metrics), and variable-levels (1 metric).

Process Metrics. Process metrics have been proven useful in defect prediction algorithms [32], [33]. We collect five

² It is worth noting that studying the effect of refactoring on software quality is a topic that remains relatively underdeveloped (despite being a highly active topic). Therefore, while this research topic is evolving, the evidence is likely to be far from clear-cut and, in some cases, it might even be contradictory (e.g., [5], [29]).

TABLE 3
Overview of the Number of Instances of Refactoring and Non-Refactoring Classes

	All	Apache	GitHub	F-Droid
Class-level refactorings				
Extract Class	41,191	6,658	31,729	2,804
Extract Interface	10,495	2,363	7,775	357
Extract Subclass	6,436	1,302	4,929	205
Extract Superclass	26,814	5,228	20,027	1,559
Move And Rename Class	654	87	545	22
Move Class	49,815	16,413	32,259	1,143
Rename Class	3,991	557	3,287	147
Method-level refactorings				
Extract And Move Method	9,723	1,816	7,273	634
Extract Method	327,493	61,280	243,011	23,202
Inline Method	53,827	10,027	40,087	3,713
Move Method	163,078	26,592	124,411	12,075
Pull Up Method	155,076	32,646	116,953	5,477
Push Down Method	62,630	12,933	47,767	1,930
Rename Method	427,935	65,667	340,304	21,964
Variable-level refactorings				
Extract Variable	6,709	1,587	4,744	378
Inline Variable	30,894	5,616	23,126	2,152
Parameterize Variable	22,537	4,640	16,542	1,355
Rename Parameter	33,6751	61,246	261,186	14,319
Rename Variable	324,955	57,086	250,076	17,793
Replace Variable w/ Attr.	25,894	3,674	18,224	3,996
Non-refactoring instances				
Class-level	10,692	1,189	8,043	1,460
Method-level	293,467	38,708	236,060	18,699
Variable-level	702,494	136,010	47,811	518,673

different process metrics: quantity of commits, the sum of lines added and removed, number of bug fixes, and number of previous refactoring operations. The number of bug fixes is calculated by means of an heuristic: Whenever any of the keywords {bug, error, mistake, fault, wrong, fail, fix} appear in the commit message, we count one more bug fix to that class. The number of previous refactoring operations is based on the refactorings we collect from RefactoringMiner.

Code Ownership Metrics. We adopt the suite of ownership metrics proposed by Bird *et al.* [34]. The *quantity of authors* is the total number of developers that have contributed to the given software artifact. The *minor authors* represent the number of contributors that authored less than 5 percent (in terms of the number of commits) of an artifact. The *major authors* represent the number of developers that contributed at least 5 percent to an artifact. Finally, *author ownership* is the proportion of commits achieved by the most active developer.

The cardinality of the set of features we use to train each model varies. The feature set for training models whose desired output is to predict class-level refactoring comprises 46 features: 37 source code metrics, 5 process metrics, and 4 ownership metrics. As for the training of method-level models, we use a set of features that comprises all the 37 class-level source code metrics plus 20 method-level source code metric features, totaling 57 features. The same holds for variable-level models, all class, method, and variable-level source code metrics features are used to fit these models.

Process and ownership metrics are only used in class-level refactoring models. Our tool relies on Git data to measure ownership and process metrics. However, Git provides

TABLE 4
List of Features Collected at Class, Method, and Variable Levels

Class-level (total of 46 metrics)

Source Code (37 metrics): CBO, WMC, RFC, LCOM, number of methods, number of static methods, number of public methods, number of private method, number of protected method, number of abstract methods, number of final methods, number of synchronized methods, number of fields, number of static fields, number of public fields, number of private fields, number of protected fields, number of default fields, number of final fields, number of synchronized fields, number of static invocations, lines of code, number of ‘return’ statements, number of loops, number of comparison expressions, number of try catches, number of expressions with parenthesis, number of string literals, number of ‘number constants’, number of assignments, number of mathematical operators, number of declared variables, max number of nested blocks, number of anonymous classes, number of sub classes, number of lambda expressions, number of unique words.

Process (5 metrics): Quantity of commits, sum of lines added, sum of lines deleted, number of bug fixes, number of previous refactoring operations.

Ownership (4 metrics): Quantify of authors, quantity of minor authors, quantity of major authors, author ownership.

Method-level (total of 20 metrics + 37 code metrics at class-level)

Source Code (20 metrics): CBO, WMC, RFC, lines of code, number of ‘return’ statements, number of variables, number of parameters, number of loops, number of comparison operators, number of try / catches, number of expressions with parenthesis, number of string literals, number of ‘number constants’, number of assignment, number of mathematical operators, max number of nested blocks, number of anonymous classes, number of sub-classes, number of lambda expressions, number of unique words.

Variable-level (total of 1 metric + 57 method+class level)

Source Code (1 metric): Number of times the variable is used.

information solely at file and line levels. While process and ownership metric values for a file are good approximations of process and ownership metric values for classes, the same does not hold for methods and variables. Technically speaking, extracting such metrics in a fine-grained manner (i.e., which methods or variables were modified, precisely) would cost extra computational analysis, which we decided to avoid. We discuss the importance of such metrics later in Section 4.3.³

2.4 Model Training

In this step, we train different ML algorithms to predict refactoring opportunities. We use the collected refactoring instances (and their non-refactoring counterexamples) as training data.

We make use of six different (binary classification) supervised ML algorithms, all available in the scikit-learn [36] and keras:

3. Recent work by Higo *et al.* [35] proposes a “finer Git”, which tracks changes in individual methods. Such tool was not available at the time of this research.

By using class-level features in the training of method-level refactoring prediction models (or similarly, class-level and method-level features in variable-level refactoring models), we give models a “sense of context”. The intuition is that developers might not decide to refactor a method by only looking at it; rather, they might look at the overall context (i.e., class) that the method belongs to.

Subsequently, the input to a trained model is a feature vector containing the source code, process, and ownership metrics of the class, method, or variable one wants to predict.

- i) Logistic Regression [37]: Logistic Regression is, similarly to linear regression, centered on combining input values using coefficient values (i.e., weights) to predict an outcome value. However, differently from linear regression, the outcome value being modeled ranges from 0 to 1.
- ii) (Gaussian) Naive Bayes [38]: Naive Bayes algorithms describe a set of steps to apply Bayes' theorem to classification problems. These algorithms use training data to compute the probability of each outcome based on the information extracted from feature values.
- iii) Support Vector Machines [39]: Support Vector Machines computes a hyper-plane in a high-dimensional space to classify data into predefined classes. The algorithm searches for the best hyper-plane to separate the training instances into their respective classes.
- iv) Decision Trees [40]: Decision Tree algorithms yield hierarchical models composed of decision nodes and leaves. Essentially, the resulting models represent a partition of the feature space.
- v) Random Forest [41]: Random Forest is an ensemble of decision tree predictors. That is, such algorithm uses a number of decision trees with random subsets of the training data.
- vi) Neural Networks [42]: Neural Networks are a family of algorithms designed to loosely resemble how the human brain processes information. The elements that comprise the architecture of such algorithms are similar to neurons, and Neural Networks are made up of one or more layers of these neurons. Essentially, these layers of neurons act as a function, mapping inputs into their respective classes.

We decided to choose a mixture of simple/less sophisticated learners (e.g., Logistic Regression and Naive Bayes) and smarter learners (e.g., Decision Trees and Random Forests). Simple learners serve as a baseline to understand whether more complex learners are needed.

Our training pipeline works as follows:

- i) We collect the refactoring and the non-refactoring instances for a given dataset d and a refactoring R . We merge them in a single dataset, where refactoring instances are marked with a *true* value and non-refactoring instances are marked with a *false* value. These instances will later serve as training and test data.
- ii) The number of refactoring instances vary per refactoring; thus, the number of refactoring instances might be greater than or smaller than the (fixed) number of non-refactoring instances. Thus, we balance the dataset as to avoid the model to favour the majority class. To that aim, we use scikit-learn's random under sampling algorithm, which randomly selects instances of the over-sampled class.⁴
- iii) We scale all the features to a $[0,1]$ range to speed up the learning process of the algorithms [43]. We use the Min-Max scaler provided by the scikit-learn framework.
- v) We tune the hyper parameters of each model by means of a random search. We use the randomized

4. We discuss the impact of balancing the classes in Section 6.2.

search algorithm provided by the scikit-learn. We set the number of iterations to 100 and the number of cross-fold validations to 10. Thus, we create 1,000 different models before deciding the model's best parameters. For the Support Vector Machines (SVM) in particular, we use number of iterations as 10 and number of cross-fold validations to 5, given its slow training time (which we discuss more below). For each algorithm, we search the best configuration among the following parameters:

- *Logistic Regression: C*: This parameter specifies, inversely, the strength of the regularization. Regularization is a technique that diminishes the chance of overfitting the model.
- *Naive Bayes: Smoothing*: It specifies the variance of the features to be used during training.
- *SVM: C*: This parameter informs the SVM optimization algorithm how much it is desired to avoid misclassifying training instances. Like the C parameter in the Logistic Regression, it helps in avoiding overfitting. Moreover, given that our goal is to also understand which features are important to the model (RQ₂), we opt only for the linear kernel of the SVM. Future research should explore how non-linear kernels perform.
- *Decision tree: Max depth*: It specifies the maximum depth of the generated tree. The deeper the tree, more complex the model becomes; *Max features*: It defines the maximum number of features to be inspected during the search for the best split, generating inner nodes; *Min sample split*: It indicates the minimum number of instances needed to split an internal node, supporting the creation of a new rule; *Splitter*: It defines the strategy in choosing the split at each node, varying from "best to random" strategies; *Criterion*: It defines the function to measure the quality of a split.
- *Random Forest*: The *max depth*, *max features*, *min samples split*, and *criterion* parameters have similar goals as to the ones in the Decision Tree algorithm; *Bootstrap*: It specifies whether all training instances or bootstrap samples are used to build each tree; *Number of estimators*: It indicates the number of trees in the forest.
- *Neural Network*: As we intend to explore sophisticated and more appropriated Deep Learning architectures in the future work (Section 4), here we compose a sequential network of three dense layers with 128, 64, and 1 units, respectively. Also, to avoid overfitting, we added dropout layers between sequential dense layers, keeping the *learning* in 80 percent of the units in dense layers. The number of epochs was set to 1,000. This architecture is similar to a Multilayer Perceptron, in the sense that it is a feedforward deep network.
- v) Finally, we perform a stratified 10-fold cross-validation (i.e., 9 folds for training and 1 fold for testing) using the hyper parameters established by the search. We return the precision, recall, and accuracy of all the models.

Once a binary classification model for a given refactoring R is trained, given a code element e (i.e., a class, method, or a variable), the model would predict true in case e should undergo through a refactoring R , or false in case e should not undergo through a refactoring R .⁵

2.5 Evaluation

To answer RQ₁, we report and compare the mean precision, recall, and accuracy among the different models after the 10 stratified cross-fold executions.⁶ We apply stratified sampling in all the cross-fold executions to make sure both training and test datasets contain the same amount of positive and negative instances. For SVM and the Neural Network, we set the number of cross-folds to 5. The SVM and the Neural Network models training and validation processes took 237 and 232 hours, respectively. The precision, recall, and accuracy across the five folds of both models were highly similar, indicating that the models are stable (numbers can be found in our appendix [30]), and thus, we have no reason to believe that the smaller number of cross-fold validations for the SVM and Neural Network affected their results.

For clarity, we revisit what a correct prediction means in this context. We recall that the feature vectors of the positive labels (i.e., elements that underwent some refactoring operations) are represented by the code metrics collected at the commit right before developers refactored them. In other words, the feature vector represents the code element at the moment that the developer decided that it needed to undergo a refactoring. On the other hand, the feature vectors of the negative labels are represented by the code metrics of classes that did not undergo the refactoring operation for k commits in a row. In other words, code that can be maintained for at least k commits without undergoing a refactoring. Thus, a correct prediction means that the model was able to predict that a code element with that characteristic underwent a refactoring operation.

For example, let us suppose a method `m1()` underwent a *Extract Method* in commit 10. This means a developer, when working with `m1()`'s implementation at version 9 decided the method needed a *Extract Method*. When testing the model, we give a feature vector representing `m1()` in commit 9, and we expect the model to return “true” (i.e., this method needs a *Extract Method*). If the model returned “false”, that would result in a false negative. Moreover, suppose another method `m2()` that was changed 50 times between commits [10, 200]. In none of these changes developers refactored this method. When testing the model, we give a feature vector representing method `m2()` in commit 10, and we expect the model to return “false” (i.e., the method does not need a *Extract Method*). If the model returned “true”, that would be an example of a “false negative”.

To answer RQ₂, we report how often each feature (from Table 4) appears among the top-1, top-5, and top-10 most important features of all the generated models. We use scikit-learn’s ability to extract the feature importance of the Logistic Regression, SVM, Decision Trees, and Random Forest

5. For completeness, in such models, a false positive would mean that the model predicted true for an element e that, in fact, did not undergo a refactoring R ; a false negative would mean that a model predicted false for an element e that, in fact, did undergo a refactoring R .

6. We kept the 50-50 distribution in all the 10 folds.

models. The framework does not currently have a native way to extract feature importance of Gaussian Naive Bayes and Neural Networks. We intend to extract the feature importance of both algorithms via “permutation importance” in future work. Given the high number of different models we build (we extracted the feature importance of 320 out of the 480 models we created), we have no reason to believe the lack of these two models would affect the overall findings of this RQ. Given that the number of features vary per refactoring level, we generate different rankings for the different levels (i.e., different ranks for class, method, and variable-level refactorings). Some models (e.g., SVM) might return the importance of a feature as a negative number, indicating that the feature is important for the prediction of the negative class. We consider such a feature also important to the overall model, and thus, we build the ranking using the absolute value of feature importance returned by the models.

Finally, to answer RQ₃, we test each of our dataset-specific models on the other datasets. For example, we test the accuracy of all Apache’s models in the GitHub and F-Droid datasets. More formally, for each combination of datasets d_1 and d_2 , where $d_1 \neq d_2$, and refactoring r we: 1) load the previously trained r model of the d_1 dataset, 2) open the data we collected for r of the d_2 dataset, 3) apply the same pre-processing steps (i.e., sampling and scaling), 4) use d_1 ’s model to predict all data points of d_2 ’s dataset, 5) and report the precision, recall, and accuracy of the model.

2.6 Implementation and Execution

The data collection tool is implemented in Java and stores all its data in a MySQL database. The tool integrates natively with RefactoringMiner [23] (also written in Java) as well as with the source code metrics tool.

The tool gives RefactoringMiner a timeout of 20 seconds per commit to identify a refactoring. We define the timeout as RefactoringMiner performs several operations to identify refactorings, and these operations grow exponentially, according to the size of the commit. Throughout the development of this study, we observed some commits taking hours to be processed. The 20 seconds was an arbitrary number decided after experimentation. In practice, most commits are resolved by the tool in less than a second. Given that its performance is related to the size of the commit and not to the size of the class under refactoring or the number of refactorings in a commit, we do not believe that ignoring commits where RefactoringMiner takes a long time influences our sample in any way.

Given that our tool integrates different tools, there are many opportunities for failures. We have observed (i) the code metrics tool failing when the class has an invalid structure (and thus, ASTs can not be built), (ii) our tool failing to populate process and ownership metrics of refactored classes (often due to files being moved and renamed multiple times throughout history, which our tool could not track in 100 percent of the cases), (iii) RefactoringMiner requiring more memory than what is available in the machine. To avoid possible invalid data points, we discard all data points that were involved in any failure (a total of 10 percent of the commits we analyzed).

We had 30 Ubuntu 18.04 LTS (64bits) VMs, each with 1 GB of Ram, 1 CPU core, and 20 GB of disk available for data

TABLE 5
The Precision (Pr), Recall (Re), and Accuracy (Acc) of the Different ML Models, When Trained and Tested in the Entire Dataset (Apache + F-Droid + GitHub)

	Logistic Regression			SVM (linear)			Naive Bayes (gaussian)			Decision tree			Random Forest			Neural Network		
	Pr	Re	Acc	Pr	Re	Acc	Pr	Re	Acc	Pr	Re	Acc	Pr	Re	Acc	Pr	Re	Acc
Class-level refactorings																		
Extract Class	0.78	0.91	0.82	0.77	0.95	0.83	0.55	0.93	0.59	0.82	0.89	0.85	0.85	0.93	0.89	0.80	0.94	0.85
Extract Interface	0.83	0.93	0.87	0.82	0.94	0.87	0.58	0.94	0.63	0.90	0.88	0.89	0.93	0.92	0.92	0.88	0.90	0.89
Extract Subclass	0.85	0.94	0.89	0.84	0.95	0.88	0.59	0.95	0.64	0.88	0.92	0.90	0.92	0.94	0.93	0.84	0.97	0.89
Extract Superclass	0.84	0.94	0.88	0.83	0.95	0.88	0.60	0.96	0.66	0.89	0.92	0.90	0.91	0.93	0.92	0.86	0.94	0.89
Move And Rename Class	0.89	0.93	0.91	0.88	0.95	0.91	0.69	0.94	0.76	0.92	0.95	0.94	0.95	0.95	0.95	0.88	0.94	0.91
Move Class	0.92	0.96	0.94	0.90	0.97	0.93	0.67	0.96	0.74	0.98	0.96	0.97	0.98	0.97	0.98	0.92	0.97	0.94
Rename Class	0.87	0.94	0.90	0.86	0.96	0.90	0.63	0.96	0.69	0.94	0.91	0.93	0.95	0.94	0.94	0.88	0.94	0.91
Method-level refactorings																		
Extract And Move Method	0.72	0.86	0.77	0.71	0.89	0.76	0.63	0.94	0.69	0.85	0.75	0.81	0.90	0.81	0.86	0.79	0.85	0.81
Extract Method	0.80	0.87	0.82	0.77	0.88	0.80	0.65	0.95	0.70	0.81	0.86	0.82	0.80	0.92	0.84	0.84	0.84	0.80
Inline Method	0.72	0.88	0.77	0.71	0.89	0.77	0.61	0.94	0.67	0.94	0.87	0.90	0.97	0.97	0.97	0.77	0.85	0.80
Move Method	0.72	0.87	0.76	0.71	0.89	0.76	0.63	0.93	0.70	0.98	0.87	0.93	0.99	0.98	0.99	0.76	0.84	0.78
Pull Up Method	0.78	0.90	0.82	0.77	0.91	0.82	0.68	0.95	0.75	0.96	0.88	0.92	0.99	0.94	0.96	0.82	0.87	0.84
Push Down Method	0.75	0.89	0.80	0.75	0.90	0.80	0.66	0.94	0.73	0.97	0.76	0.87	0.97	0.83	0.90	0.81	0.92	0.85
Rename Method	0.77	0.89	0.80	0.76	0.90	0.80	0.65	0.95	0.71	0.78	0.84	0.80	0.79	0.85	0.81	0.81	0.82	0.81
Variable-level refactorings																		
Extract Variable	0.80	0.83	0.82	0.80	0.83	0.82	0.62	0.94	0.68	0.82	0.83	0.82	0.90	0.83	0.87	0.84	0.89	0.86
Inline Variable	0.76	0.86	0.79	0.75	0.87	0.79	0.60	0.94	0.66	0.91	0.85	0.88	0.94	0.96	0.95	0.81	0.82	0.82
Parameterize Variable	0.75	0.85	0.79	0.74	0.86	0.78	0.59	0.94	0.65	0.88	0.81	0.85	0.93	0.92	0.92	0.80	0.83	0.81
Rename Parameter	0.79	0.88	0.83	0.80	0.88	0.83	0.65	0.95	0.71	0.99	0.92	0.95	0.99	0.99	0.99	0.82	0.87	0.84
Rename Variable	0.77	0.85	0.80	0.76	0.86	0.79	0.58	0.92	0.63	0.99	0.93	0.96	1.00	0.99	0.99	0.81	0.84	0.82
Replace Variable With Attribute	0.79	0.88	0.82	0.78	0.89	0.82	0.64	0.95	0.71	0.90	0.84	0.88	0.94	0.92	0.93	0.79	0.92	0.84

Values range between [0,1]. Numbers in bold represent the highest accuracy for each refactoring operation.

collection. These machines, altogether, spent a total of 933 hours to collect the data. We observe that the majority of projects (around 99 percent of them) took less than one hour to be processed. 159 of them took more than one hour, and 70 of them more than two hours. A single project took 23 hours.

The ML pipeline was developed in Python. Most of the code relies on the *scikit-learn* framework [36] and *keras* for the Neural Networks training. To the ML training, we had under our disposition two machines: one Ubuntu 18.04.2 LTS VM, 396 GB of RAM, 40 CPU cores, and one Ubuntu 18.04.2 LTS VMS with 14 CPUs and 50 GB of RAM. Given the hyperparameter search and cross-validations, our ML pipeline experimented with a total of 404,080 models. The overall computation (training and testing) time was approximately 500 hours.

2.7 Reproducibility

Our online appendix [30] contains: (*i*) the list of the 11,149 projects analyzed, (*ii*) a spreadsheet with the full results, (*iii*) the source code of the data collection and the ML tools, and (*iv*) a two million refactorings dataset.

3 RESULTS

In the following subsections, we answer each of the RQs.

3.1 RQ1: How Accurate are Supervised ML Algorithms in Predicting Software Refactoring?

In Table 5, we show the precision, recall, and accuracy of each ML algorithm in each one of the 20 refactoring operations, when training and testing in the entire dataset. Due to space constraints, we show the results of training and testing in individual datasets, as well as the confusion matrix, in our appendix [30].

Observation 1: Random Forest models are the most accurate in predicting software refactoring. Random Forest has the highest overall accuracy among all types of refactorings. Its average accuracy for class, method, and variable-level refactorings, when trained and tested in the entire dataset, are 0.93, 0.90, and 0.94, respectively. The only three refactorings that are below the 90 percent threshold are *Extract Class*, *Extract and Move Method*, and *Extract Method*. Its average accuracy among all refactorings in all the datasets together, as well as Apache, GitHub, and F-Droid datasets only, are 0.93, 0.94, 0.92, and 0.90, respectively. As a matter of comparison, the second best model is Decision Trees, which achieves an average accuracy of 0.89, 0.91, 0.88, and 0.86 in the same datasets.

Observation 2: Random Forest was outperformed only a few times by Neural Networks. In the F-Droid dataset, Neural Networks outperformed Random Forest 4 times (in terms of accuracy). Neural Networks also outperformed Random Forest in two opportunities in both the Apache and GitHub datasets. However, we note that the difference was always marginal (around 1 percent).

Observation 3: Naive Bayes models present high recall, but low precision. The Naive Bayes models presented recalls of 0.94, 0.93, 0.94, and 0.84 in the entire dataset, Apache, GitHub, and F-Droid datasets, respectively. These numbers are often slightly higher than the ones from Random Forest models, which were the best models (on average, 0.01 higher). Nevertheless, Naive Bayes models presented the worst precision values: 0.62, 0.66, 0.62, and 0.67 in the same datasets. Interestingly, no other models presented such low precision.

Observation 4: Logistic Regression, as a baseline, shows good accuracy. Logistic Regression being, perhaps, the most straightforward model in our study, presents a

TABLE 6
Most Important Features for the Models
at Different Refactoring Levels

Class-level refactorings	
Top-1: quantity of commits (68), author ownership (32), lines added (6)	
Top-5: quantity of commits (108), lines added (63), previous refactorings (63), author ownership (56), unique words in the class (47)	
Top-10: quantity of commits (111), lines added (90), previous refactorings (90), unique words in the class (78), class LOC (70)	
Method-level refactorings	
Top-1: class LOC (39), number of unique words in a class (15), number of methods in a class (13), class LCOM (9), number of fields in a class (6)	
Top-5: class LOC (74), number of methods in a class (55), number of unique words in a class (52), class LCOM (37), number of final fields in a class (25)	
Top-10: number of methods in a class (90), class LOC (88), class LCOM (71), number of unique words in a class (54), class CBO (54)	
Variable-level refactorings	
Top-1: class LOC (27), class LCOM (10), number of unique words in a class (9), method LOC (7), number of public fields in a class (7)	
Top-5: class LOC (61), number of unique words in a class (48), number of string literals in a class (38), number of variables in the method (30), number of public fields in a class (24)	
Top-10: number of string literals in a class (72), class LOC (71), number of unique words in a class (66), number of variables in a class (55), number of variables in a method (49)	

Top-1, Top-5, and Top-10 indicate the number of times (in parenthesis) a specific feature appeared in the top-N ranking. For class and method level refactorings, a feature can at most appear 112 times; 96 times for a variable level refactoring. We show only the first five features per ranking; full list in the online appendix [30].

somewhat high overall accuracy, always outperforming Naive Bayes models. The average accuracy of the model in all the refactorings in the entire dataset is 0.83. Its best accuracy was in the *Move Class* refactoring: 0.94 (which also presented high values in the individual datasets: in F-Droid, 0.94, in GitHub, 0.93, and in Apache, 0.95), and its worst accuracy, 0.77, was in the *Extract and Move Method* and *Inline Method* refactorings. The overall averages are similar in the other datasets: 0.85 in Apache, 0.83 in GitHub, and 0.78 in F-Droid.

3.2 RQ2: What are the Important Features in the Refactoring Prediction Models?

In Table 6, we show the most important features per refactoring level. The complete ranking of features importance can be found in the online appendix [30].

Observation 5: Process metrics are highly important in class-level refactorings. Metrics such as *quantity of commits*, *lines added in a commit*, and *number of previous refactorings* appear in the top-1 ranking very frequently. In the top-5 ranking, seven out of the first ten features are process metrics; six out of the first ten are process metrics in the top-10 ranking. Ownership metrics are also considered important by the models. The *author ownership* metric appears 32 times in the top-1 ranking; the *number of major authors* and *number of authors* metrics also appear often in the top-5 and top-10 rankings.

Observation 6: Class-level features play an important role in method-level and variable-level refactorings.

TABLE 7
The Average Precision (Pr) and Recall (Re) of the 20 Refactoring Prediction Random Forest Models, When Trained in One Dataset and Tested in Another Dataset

	Apache		GitHub		F-Droid	
	Pr	Re	Pr	Re	Pr	Re
Apache	-	-	0.84	0.79	0.77	0.70
GitHub	0.87	0.84	-	-	0.84	0.80
F-Droid	0.77	0.73	0.81	0.76	-	-

Rows represent datasets used for training, and columns represent datasets used for testing.

Method-level refactoring models often consider class-level features (e.g., lines of code in a class, number of methods in a class) to be more important than method-level features. In the top-1 ranking for the method-level refactoring models, 13 out of the 17 features are class-level features. In variable-level refactoring models, the same happens in 11 out of 17 features. Interestingly, the most fine-grained feature we have, the *number of times a variable is used* appears six times in the top-1 ranking for the variable-level refactoring models.

Observation 7: Some features never appear in any of the rankings. For class-level refactoring models, the *number of default fields*, and the *number of synchronized fields*⁷ do not appear even in the top-10 ranking. Nine other features never appear in the top-10 feature importance ranking of method-level refactoring models (e.g., *number of comparisons*, *math operations*, and *parenthesized expressions*), and ten features never make it in the variable-level refactoring models (e.g., *number of loops*, and *parenthesized expressions*).

3.3 RQ3: Can the Predictive Models be Carried Over to Different Contexts?

We show the precision and recall of each model and refactoring, in all the pairwise combinations of datasets in our appendix [30]. In Table 7, we show the overall average precision and recall of the Random Forest models (the best model, according to RQ₁ results) when trained in one dataset and tested in another dataset.

Observation 8: Random Forest still presents excellent precision and recall when generalized, but smaller when compared to previous results. Random Forest models achieve precision and recall of 0.87 and 0.84, when trained using the GitHub repository, the largest repository in terms of data points, and tested in Apache. When trained in the smallest dataset, F-Droid, Random Forest still performs reasonably well: precision and recall of 0.77 and 0.73 when tested in Apache, and 0.81 and 0.76 when tested in GitHub. Nevertheless, we remind the reader that in terms of accuracy, Random Forest achieved average scores of around 90 percent. In other words, models seem to perform best when trained with data collected from different datasets.

⁷ By looking at the features distribution in our appendix [30], we observe that most classes do not have synchronized fields; we discuss how feature selection might help in simplifying the final models in Section 4.

Observation 9: Method and variable-level refactoring models perform worse than class-level refactoring. In general, class-level refactoring models present higher precision and recall than the method- and variable-level refactoring models. Using a model trained with the GitHub data set and tested in the F-Droid data set, the average precision and recall for Random Forest models at class-level are 0.92 and 0.92. On the other hand, the average precision and recall for Random Forest models at method-level are 0.77 and 0.72, respectively; at variable-level, we observe precision and recall of 0.81 and 0.75.

Observation 10: SVM outperforms Decision Trees when generalized. We observed Decision Trees being the second best model in RQ₁. When carrying models to different contexts, however, we observe that SVM is now the second best model, and only slightly worse than the Random Forest. For example, in the appendix, we see that for a model trained in GitHub and tested in Apache, the average precision and recall of SVM models is 0.84 and 0.83 (in contrast, Random Forest models have 0.87 and 0.84). The difference between both models is, on average, 0.02.

Observation 11: Logistic Regression is still a somewhat good baseline. Logistic Regression baseline models, when carried to different contexts, still present somewhat good numbers. As an example, the models trained with GitHub data and tested in the Apache dataset show an average precision and recall of 0.84 and 0.83. The worst averages happen in the models trained with the Apache dataset and tested in the F-Droid dataset (precision of 0.75 and recall of 0.72).

Observation 12: Heterogeneous datasets might generalize better. More homogeneous datasets (i.e., the Apache and F-Droid datasets), when carried to other contexts, present lower precision and recall. This phenomenon can be seen whenever Apache and F-Droid models are cross tested; their precision and recall never went beyond 0.78. This phenomenon does not happen when GitHub, a more heterogeneous dataset in terms of different domains and architectural decisions, is tested on the other two datasets.

4 DISCUSSION

In the following, we extensively discuss some important ramifications of our research. More specifically, we discuss:

- 1) the challenges in defining k as a constant to collect non-refactored instances,
- 2) the features used for model building as well as their interpretability,
- 3) the importance of process and ownership metrics (and the need for fine-grained metrics),
- 4) the need for larger and more heterogeneous datasets to achieve higher generalizability,
- 5) how to prioritise the refactoring recommendation suggestions given by the models,
- 6) the need for more fine-grained refactoring recommendations,
- 7) the recommendation of high-level refactorings,
- 8) taking the developers' motivations into account,

- 9) the use of Deep Learning (and Natural Language Processing algorithms) for software refactoring, and
- 10) the challenges of deploying ML-based refactoring recommendation models in the wild.

4.1 Collecting Non-Refactored Instances via an Heuristic

The identification of negative instances, i.e., code elements that did not undergo a refactoring operation, is an important theoretical problem that our research community should overcome.

We propose the use of code elements that did not undergo refactoring operations for k commits in a row. In this particular paper, we chose $k = 50$ (i.e., 50 commits in a row without being refactored) as a constant to determine whether a class, its methods, and its variables should be considered an instance of a non-refactoring. The number 50 was chosen after manual exploration in the dataset.

To measure the influence of k in our study, we re-executed our data collection procedure in the entire dataset (11,149 projects) with two different values for k :

- $k = 25$. The half of the value used in the main experiment. A threshold of 25 means that we are less conservative when considering instances for the non-refactoring dataset. In this dataset, we have a total of 7,210,452 instances (at class, method, and variable levels). This represents an increase of 7.1 times when compared to the dataset in the main experiment.
- $k = 100$. The double of the value used in the main experiment. A threshold of 100 means that we are more conservative when it comes to considering a class as an instance of a non-refactoring. In this dataset, we have a total of only 120,775 instances. This represents around 12 percent of the dataset in the main experiment.

We note that the distribution of the features values of the non-refactored instances in $k = 25$ and $k = 100$ datasets are somewhat different from each other. As examples, the quantiles of the *CBO at class-level* in $k = 25$ dataset are [1Q=17, median=35, mean=57, 3Q=69], whereas the quantiles in $k = 100$ dataset are [1Q=6, median=28, mean=54, 3Q=75]; for the *WMC at class-level*, we observe, [1Q=59, median=145, mean=273, 3Q=343] for $k = 25$, and [1Q=72, median=266, mean=425, 3Q=616] for $k = 100$; for the *LOC at class-level*, we observe [1Q=320, median=734, mean=1287, 3Q=1626] for $k = 25$, and [1Q=466, median=1283, mean=1568, 3Q=2189] for $k = 100$.

We trained Random Forest models (given that it was the algorithm with the best accuracy in RQ₁) in both $k = 25$ and $k = 100$ datasets. In $k = 25$, the average of the absolute difference in the precision and recall of the 20 refactoring models, when compared to $k = 50$, are 0.0725 and 0.099, respectively. In $k = 100$, the average of the absolute difference in precision and recall when compared to $k = 50$ are 0.0765 and 0.064, respectively. The precision and recall of each refactoring is in our online appendix [30].

In $k = 25$, however, in only four (*Move Class*, *Move and Rename Class*, *Extract Method*, and *Rename Method*; out of 20) models, the precision values were better than in the $k = 50$. Similarly, only a single model (*Rename Method*) had a better

recall when compared to $k = 50$. This might indicate that $k = 25$ is not a good threshold, as it might be too small. In $k = 100$, while it is hard to distinguish whether it has a better precision than $k = 50$ (11 models did better with $k = 50$ and 9 models did better with $k = 100$), models in $k = 100$ had almost always a better recall (16 models out of 20). This might indicate that more conservative thresholds might help in increasing recall at the expense of precision performance. This discussion shows the importance of finding the right threshold to determine classes, methods, and variables that can serve as non-refactoring instances.

It is worth emphasizing that our proposed heuristic to detect counterexamples of refactoring instances is an approximation. While we believe that our assumption that classes that can still be evolved by developers without the need for refactoring can serve as good counterexamples for the model, these data points are simply approximations. There might be other more effective counterexamples that would contribute to the creation of better models.

As an alternative, when designing the model, we considered the possibility of doing the extraction of non-refactored instances at commit-level. For example, whenever a refactoring R was detected in a class, method, or variable a , we extracted all the other elements that existed in the modified files of that commit as examples of non-refactored instances. We relied on the assumption that the elements that did not change in that commit could be used as counterexamples during model creation. We, however, discarded this idea after some exploration. When looking at individual commits *only* and not at larger time windows, one can not determine whether a code element is an example of an element that does not need to be refactored. The same code element might have changed in the subsequent commit, thus rendering the previously collected data invalid (i.e., mischaracterizing the counterexample). Furthermore, another factor we took into account was that, if we consider all elements that were not refactored in every single commit as a counterexample, the number of extracted data points would be orders of magnitude higher than the number of data points for refactoring instances. That would make the dataset highly imbalanced. We decided not to deal with a highly imbalanced dataset because that is a known challenge in ML [44].

Given that the current state-of-the-art enables us to precisely identify refactoring operations that have happened in software systems, but the identification of non-refactoring instances is challenging, we suggest the possibility of training models using solely a single class. In this case, one would train the model solely on the real-world refactoring instances, and use the outcome probability of the model to decide whether to recommend a refactoring operation. We expect models to return a very low probability in methods that do not need to undergo refactoring. Note that, in this way, there is no need for collecting non-refactoring data points, which would avoid the problem discussed in the previous subsection. We therefore suggest future work to explore the performance (as well as the drawbacks) of such models in recommending software refactoring.

Nevertheless, the fact that our community does not have an accurate dataset composed of examples of code elements that do not need refactoring is a threat to any study in software refactoring. Our community has been working on

several approaches to point developers to problematic pieces of code for a long time. However, less research has been dedicated to revealing exemplary pieces of code (exemplary in the sense that these pieces of code do not warrant refactoring operations). Given the data-driven era we find ourselves in, research investigating the identification and creation of a sample of such pieces of code might be highly relevant.

4.2 Features and Their Interpretability

Our models use a set of source code, process, and ownership metrics as features (see Table 4 for the complete list). The choice of features was mostly based on previous ML models for software engineering tasks (e.g., [20], [27], [28]).

Our conjecture when we settled on using structural metrics was that the structure of a class or method is an important factor that developers take into consideration when identifying pieces of code to refactor, e.g., a complex method is much more prone to being refactored than a structurally less complex method. Given the high accuracy, precision, and recall that we observed in our empirical study, our conjecture seems to hold. We understand that some of the metrics might seem counterintuitive. Some developer might be hard-pressed to explain why something as the *number of mathematical operations* in a given part of the code may indicate that refactoring is warranted.

Given the amount of features that are readily available and that have been used in the literature, we decided not to perform manual feature selection (i.e., manually selecting the most appropriate features given the data and the model). Rather, we decided to let the model decide which ones have more predictive power.

Interestingly, as the results we used to answer our RQ₂ seem to suggest, models tend to select features that also make more sense to humans. For example, number of methods in a class (which was chosen 13 times as the most important predictor in method-level refactorings) or number of lines of code in a class (which was selected 39 times as the features that most contributed to model building). On the other hand, the number of parenthesized expressions and number of lambdas do not seem to help models in learning how to recommend refactoring; such features were automatically discarded (i.e., never used) by these models.

We, nevertheless, understand that the interpretability of these models can play a decisive factor in whether developers will accept the recommendation. Developers might want to know why the model is suggesting a specific refactoring. While interpretability of models is a complex problem in the area of ML in general [45], [46], making use of metrics that developers can better relate to, as well as showing them what metrics most influenced the model to recommend a given refactoring might make the developer more confident in accepting the recommendations. Interpretability of refactoring recommendation models is therefore an important future work.

4.3 The Importance of Process and Ownership Metrics (and the Need for Fine-Grained Metrics)

We observed that process metrics are indeed considered important by the models (see RQ₂). For example, the *number of commits* metric figured as the most important feature in

the class-level refactoring models. Additionally, related research suggests that defect prediction models [20], [32], [33] also benefit from process metrics.

While source code metrics are able to capture the structure of a code element, process metrics are able to capture its evolution history (e.g., number of changes, code churn, number of bugs, or refactorings over time). Such characteristics seem to play an important factor when deciding whether to refactor the code element. We would argue that this is inline with general knowledge on software design. For example, changing a class several times eventually leads to brittle design (following Lehman's laws of software evolution [47], [48]), which drives developers to remedy the situation by refactoring the class; or a class that has presented a high number of bugs tend to require more "clean up" than classes that do not suffer from the same issue. Process metrics, thus, provide models with a perspective on the evolution of the class.

We currently use process and ownership metrics to support the prediction of class-level refactorings only. These metrics are naturally collected at file-level, and collecting them in a more fine-grained manner (i.e., method and variable ownership) would require complex tooling to be developed. We can only conjecture that process metrics would also help models in better predicting refactoring at method and variable levels. To that aim, it is our goal to (*i*) develop a tool that is able to collect process and ownership metrics at method and variable levels, (*ii*) feed our models with these new features, (*iii*) re-execute our ML pipeline and examine how accuracy is affected.

In addition, D'Ambros *et al.* [20] observed that the number of bugs, when extracted by means of string matching in the commit message (which is our case), might reduce the quality of the resulting predictor. Our models currently indicate that the "number of bugs" feature is relevant. This feature frequently appeared in the top five and top ten ranking of features that most contributed to model building. We surmise that a more precise approach to detecting and counting bugs, which might require better integration with issue tracking systems, will improve the quality of the recommendations.

4.4 Making a Case for Larger and More Heterogeneous Datasets

According to the results we used to answer RQ₃, larger and more heterogeneous datasets tend to generalize better. We would argue that large amounts of diverse refactoring operations contribute to the creation of stronger, more accurate models.

While our dataset might be already considered a large one, with around 3 million labelled instances, we believe that the collection of even larger datasets compressed of different types of systems and refactoring operations will result in more helpful models able to provide developers with more accurate refactoring recommendations. Moreover, it is a common observation in ML studies (not only in software engineering tasks) that simple models trained on large datasets often work better than complex models trained on small datasets [49], [50]. Simple models are cheaper to train and store.

4.5 Prioritizing Refactoring Recommendations

All our models currently perform binary classification. In other words, each model is only able to predict a single

refactoring operation. Our empirical study shows that, when tested in isolation, models have high accuracy.

We envision a recommendation tool making use of all the models together in order to recommend all the possible refactorings. Suppose we want to offer refactoring recommendations for a given method, we would need to pass the method through the seven different method-level refactoring models; each of these seven models would give its own prediction, and we would show the resulting list of recommendations to the developer.

We understand that in a scenario in which developers are faced with lots of refactoring recommendations it might be hard for them to work out which refactorings to prioritize. An avenue to explore in future work is to take advantage of the probability values that are internally produced by the models to prioritize which refactorings are more appropriate in a given context. A tool that presents these probability values to the end-users could allow them to decide which refactorings they should apply and in which order (we discuss more usability concerns of such a tool in Section 4.10).

4.6 The Need for More Fine-Grained Refactoring Recommendations

In this first step, we have showed that ML can model the refactoring recommendation problem. Although the current models provide recommendations at different levels of granularity (i.e., class, method, and variable levels), there is room for improvement by fine-tuning models to offer even more fine-grained refactoring recommendations. Take as an example the *Extract Method* refactoring. Our models can identify which method would benefit from an extraction; however, it currently does not point to which parts of that method should be extracted (i.e., initial token and end token). Another example are refactorings that involve more than one class, e.g., *Move Method* or *Pull Up Method*: to which class should the method be moved to?

We see a future where, for each of the refactorings we studied, a highly-specific model, able to provide fine-grained recommendations, is devised. We conjecture that models that learn precisely, e.g., what tokens to extract out of a method, would need to be deep. Therefore, we believe that deep learning will play an important role in the field of software refactoring in the near future. We discuss deep learning later in this section.

4.7 The Recommendation of High-Level Refactorings

In this study, we explore recommendations of low-level refactorings, i.e., small and localized changes that improve the overall quality of the code. We did not explore recommendations of high-level refactorings, i.e., larger changes that improve the overall quality of the design.

We see that the great challenge of recommending high-level refactorings is that the model requires even more context to learn from. Before applying a design pattern to the source code, developers often think about how to abstract the problem in such a way that the pattern would fit.

As an initial step, the book of Kerievsky [51] might serve as a guide. In his book, the author shows how to move code, that is often implemented in a procedural way, to a design pattern oriented solution, by means of low-level refactorings.

Our next step is to explore how we can “aggregate” several low-level refactoring recommendations in order to provide developers with high-level refactoring suggestions.

4.8 Taking the Developers’ Motivations into Account

Empirical research shows that developers refactor for several reasons, other than to “only improve the quality of the code” (e.g., [7], [52]). In our first foray into applying ML algorithms to predict refactoring operations, our models do not factor in “motivation”. Nevertheless, note that our large dataset of refactorings contains refactorings that have happened for varying reasons (given that we never filtered refactoring based on motivation from the projects). Interestingly, our models still show high accuracy. Exploring whether models built specifically for, e.g., “refactoring to add new functionality”, would provide even better results, not only in terms of accuracy, but also in terms of “developer satisfaction”. We defer this development to future work.

4.9 The Use of Deep Learning (and Natural Language Processing Algorithms) for Software Refactoring

Programming languages have phenomena like syntax and semantics [53], [54], [55]. Motivated by several recent works that use advanced ML algorithms on source code with the goal of (semi) automating several non-trivial software engineering tasks such as suggesting method and class names [56], code comments [57], generation of commit messages [58], and defect prediction [59], we intend to experiment NLP-specific deep learning architectures to deal with code refactorings. Using models like Seq2Seq [60] and Code2Vec [49], both refactorings predictions and refactored code can be outputs of the model, having the source code only as input. To facilitate the work of future researchers interested on the topic, our online appendix [30] contains a dataset with all the refactored classes studied here.

4.10 Refactoring Recommendation Models in the Wild

As mentioned, popular tools such as PMD and Sonarqube offer detection strategies for common code smells, e.g., *God Classes* and *Long Methods*. These tools have been integrated into different stages of the developers’ workflow, e.g., inside IDEs, during code review, or their results have been incorporated into quality reports. We envision a ML-powered refactoring recommendation tool finding its way into the daily life of a developer in the same way linters and code quality recommendation tools (e.g., PMD, Checkstyle, Sonarqube) currently belong to their daily routine. However, the deployment of ML-based refactoring recommendation models does not come without its challenges.

First, prediction models take up a lot of disk space (some of the models we built throughout this research take up around 700 MB to 1 GB of disk space), which makes them unwieldy to deploy inside IDEs (without mentioning that loading them into memory would require sizable memory resources). While the ML research field is still looking for efficient ways of compressing large models (see [61] for details), introducing the 20 ML models that we built into the developers’ machines/IDEs is certainly not a feasible solution. A possible workaround to this challenge would be to provide a centralized server that provides recommendations to clients (e.g., IDEs and code review tools).

Another way to reduce the size of our models would be to build leaner models. In RQ₂, we show that some features never make to the top-10 ranking features; others were never even used. Future work should investigate which features can be removed without significant loss of prediction power, thus on removing features that have no real prediction power, and on identifying the simplest model that works by, e.g., performing feature reduction. As a reference, we refer the reader to Kondo *et al.*’s work [62]. Authors explored the impact of eight different feature reduction techniques on defect prediction models; we suggest the same line of work for refactoring recommendation models.

Moreover, our empirical study shows that the training of these models take hours (some of our Random Forest models took approximately 2 hours running on a machine equipped with a 40-core processor). On the other, once these models are trained, prediction happens almost instantly. This is due to the fact that our models require a feature vector composed solely by code metrics that are easily extracted from source code. The long training time reinforces the need for generalizable models (which are possible to obtain according to our results), given that many companies are not able to afford the costly model training.

Second, program analysis tools solely require access to the source code of the program for the recommendation to happen. Our current ML models also require ownership and process metrics. While our results show that these metrics play an important role in the models, they are less trivial to be calculated, requiring access to the full history of the project as well as maintenance. Future work should evaluate what to do in cases where the developer does not have access to these metrics, i.e., when offering consultancy to a company that does not provide the consultant with the full repository, or when the project is in earlier stages and the repository still does not contain useful data.

Third, the usability aspects that such a tool would need in order for developers to trust it. In this paper, we do not explore such aspects. While this is not unique to ML-based recommendation models, we believe this is an important aspect to be explored, given that the interpretability of these (black box) models are harder than the detection strategies our community currently relies on. Guidelines on how to recommend software refactoring [19] as well as lessons learned on building large-scale recommendation tools [63], [64] are of great help. Given that ML models are drawing a lot of attention from the software engineering community, other researchers have already started to probe into the usability-related issues of ML-powered solutions to software engineering tasks (e.g., [65]).

Finally, understanding whether it is possible for a company to reuse existing models (a practice commonly used in other communities, such as the reuse of pre-trained models as Word2Vec [66] and BERT [67]) and how often the refactoring recommendation models should be re-trained are fundamental questions that still need to be answered.

5 RELATED WORK

After the publication of Fowler’s seminal book [1], refactoring went mainstream and many surveys and literature reviews on the subject were performed. One of the early

surveys that brought refactoring into the limelight of researchers was carried out by Mens and Tourwé [68]: their survey is centered around refactoring activities, supporting techniques, and tool support. Specifically, their discussion is organized around software artifacts and how refactoring applies to them, so the authors emphasize requirement refactoring, design refactoring, and code refactoring. Additionally, Mens and Tourwé briefly share their outlook on the impact of refactoring on software quality. Their survey, however, took only a few studies on identifying refactoring opportunities into account and did not follow a systematic approach. As mentioned, since then several systematic literature reviews have been conducted on refactoring.

The existing literature discusses different automatic refactoring approaches whose purpose is helping practitioners in detecting code smells, some of which are even able to suggest the refactoring activities that should be performed by the practitioners in order to remove the detected code smells. Most approaches are either based on rules, employ search-based algorithms, or ML approaches. A recent systematic literature review [69] shows that there has been an increase in the number of studies on automatic refactoring approaches. According to the results of such literature review, source code approaches have been receiving more attention from researchers than model based approaches. In addition, the results indicate that search-based approaches are gaining increasing popularity and researchers have recently begun exploring how ML can be used to help practitioners in identifying refactoring opportunities. The concepts and rule-based approaches proposed by early researchers that laid the theoretical foundation for more recent advances in the area are presented in Section 5.1. Related work on search-based approaches applied to refactoring is discussed in Section 5.2 and related work on ML is reviewed in Section 5.3.

5.1 Code Smell Detection

In hopes of providing a insightful understanding of code smells, the goals of studies on code smells, approaches used to probe into code smells, and evidence that bolsters the fundamental premise that code smells are symptoms of issues in the code, Zhang *et al.* [70] carried out a systematic literature review in which they synthesized the results of 39 studies on code smells. Since we consider the identification of code smells and the detection of refactoring opportunities two related problems, it is also worth mentioning the systematic literature review performed by Al Dallal [71]. Al Dallal discusses studies that consider both code smells and refactoring opportunities from a different perspective: the main focus of their literature review is providing an overview of code smell identification approaches. Based on an analysis of 47 studies, Al Dallal concluded that although there was a sharp increase in the number of studies on identifying refactoring opportunities, up to 2013 the results of these studies were derived mostly from relatively small datasets. Singh and Kaur [72] extended the systematic literature review carried out by Al Dallal focusing on code smells identification and anti-patterns. The two main contributions of their survey is highlighting the datasets and the tools employed in the selected studies and the identification of the code smells that were most used in these studies.

Recently, Santos *et al.* [73] performed a systematic literature review to summarize knowledge about how code smells impact software development practices, which the authors termed “smell effect”. Santos *et al.* selected and analyzed 64 studies that were published between 2000 and 2017. One of the main findings reported by the authors is that human-based evaluation of smells is not reliable: a trend in the selected studies seems to indicate that developers have a low level of consensus on smell detection. Furthermore, their analysis of the selected studies suggests that demographic data as developers’ experience can significantly impact code smell evaluation.

Fernandes *et al.* [74] carried out a systematic literature review on code smell detection tools. Their study is centered around the identification of code smell detection tools, their main features, and the types of code smells that these tools are able to identify. Fernandes *et al.* also performed a comparison of the four most widely used tools (i.e., most frequently mentioned in the selected studies). It is worth mentioning that considering the selected studies, which were published from 2000 to 2016, no tool implements a ML based approach: this indicates that only recently researchers have begun investigating ML models in this context. Rasool and Arshad [75] also performed a systematic literature review on tools and approaches to mining code smells from the source code. Essentially, Rasool and Arshad classified tools and approaches based on their detection methods. Rasool and Arshad emphasized mining approaches, thus they did not take ML-based approaches into account.

5.2 Search-Based Refactoring

Mariani and Vergilio [17] carried out a systematic literature review of how search-based approaches have been applied to refactoring. Mariani and Vergilio found that evolutionary algorithms and, in particular, genetic algorithms were the most commonly used algorithms in the analyzed studies. In addition, they found that the most widely used and investigated refactorings are the ones in Fowler’s catalog [1]. More recently, Mohan and Greer [76] also looked at search-based refactoring. However, differently from the literature survey by Mariani and Vergilio, Mohan and Greer give a more in-depth review of the selected studies in the sense that Mohan and Greer also cover other aspects of the literature. For instance, Mohan and Greer also discuss the tools used in the selected studies as well as provide an investigation of how some metrics have been tested and discussed in the selected literature. In addition, Mohan and Greer detail how the search-based approaches described in the selected studies have evolved over time. Similarly to the results presented by Mariani and Vergilio, Mohan and Greer also found that evolutionary algorithms are the most commonly used algorithms in the selected studies.

5.3 ML Algorithms

To our best knowledge, only one systematic literature review [77] has been conducted with the purpose of summarizing the research on ML algorithms for code smell prediction. Azeem *et al.* selected 15 studies that describe code smell prediction models. Azeem *et al.* analyzed the selected studies in terms of (*i*) code smells taken into account, (*ii*) setup of the ML based approaches, (*iii*) how these approaches were

evaluated, and (iv) a meta-analysis on the performance of the code smell prediction models described in the selected studies. According to the results, God Classes, Long Methods, Functional Decomposition, and Spaghetti Code are the most commonly considered code smells. Decision Trees and SVM are the most widely used ML algorithms for code smell detection. Additionally, JRip and Random Forest seem to be the most effective algorithms in terms of performance.

6 THREATS TO VALIDITY

This section outlines the threats to the validity of our study.

6.1 Construct Validity

Threats to construct validity concern the relation between the theory and the observation, and in this work are mainly due to the measurements we performed.

- Our strategy for gathering the large amount of data we investigated entailed mining a large number of software repositories for instances of class, method, and variable refactorings. Thus, the main internal validity threat is the data collection process. We cannot rule out the issues that arise when performing large scale data extraction (issues indeed happened, as discussed in Section 2.6). We provide a replication package containing all experimental scripts and datasets used in our study so that researchers and practitioners can fully replicate and confirm our results.
- As mentioned in Section 2.3, over the course of our data extraction process, we determined the number of bug fixes by employing a keyword matching approach. The approach is widely used by the mining software repositories community to detecting bug fix related information in software repositories. It is worth noting that the effectiveness of such approach depends on the keywords used during the data extraction process, so we acknowledge the possibility that we might have overlooked the inclusion of relevant keywords.
- Our data collection mechanism makes use of RefactoringMiner [23], a tool that is able to identify refactoring operations in the history of a repository. Therefore, the soundness of our approach hinges on the effectiveness of refactoring detection tool we used. RefactoringMiner presents a precision and recall of 98 and 87 percent, respectively, in detecting the refactoring operations we study. We did not re-evaluate the precision and recall of RefactoringMiner in the studied sample, as this was already established in their research. Given how RefactoringMiner works internally and that RefactoringMiner was evaluated on projects with similar characteristics (in fact, 65 percent of the projects in RefactoringMiner's evaluation dataset are in our dataset), we have no reason to believe that the accuracy reported in the literature would not apply to our study.
- An underlying assumption of our research is that refactorings that have happened in the past are good examples of refactorings that will happen in the future. Our models never learn from "refactorings

that developers find to be important, but never got around to carrying them out". Nevertheless, given the amount of data points we use for training, we have no reason to believe that "refactorings that developers consider relevant but ended up never being carried out" are so intrinsically different from the "refactorings that developers carried out". In ML terms, we do not believe their feature vectors would have such a different distribution that models would not be able to predict them with a reasonably good accuracy. This is, however, a conjecture. Case studies in industrial settings, in which developers annotate not only whether the recommendations of ML-based models were pertinent, but also refactorings they would like to perform in elements that our models do not identify refactoring opportunities, is a necessary step in order to test this conjecture.

- Finally, one of the metrics we also used to train our models was the "number of default methods" (at class-level). However, later in one of our inspections, we observed that, due to a bug in the metric collections tool, the number of default methods was always zero.⁸ All the learning algorithms ignored this metric, as it indeed added no value to the learning process; in fact, it appeared on the list of features that were never used by our model. Therefore, we affirm that this bug does not influence the overall results of our paper. Moreover, we have no reason to believe that the adding this feature would bring significant improvements. Nevertheless, we propose researchers to use this feature in future replications of this paper.

6.2 Internal Validity

Threats to internal validity concern external factors we did not consider that could affect the variables and the relations being investigated.

- We removed projects that failed during data collection. As we discussed in Section 2.1, our pipeline is composed of several tools, all of them being prone to failures, e.g., RefactoringMiner running out of memory. The percentage of failed projects is small (8 percent), and does not affect the representativeness of our final dataset.
- Owing to the fact that, in most cases, there are more instances of the non-refactoring class in our dataset than instances of the refactoring class (see Table 3), we had to cope with an imbalanced dataset. Given that there is no reliable estimate on the distribution of refactoring and non-refactoring instances "in the wild", we decided to perform under-sampling. That is, we chose to remove instances from the over-represented class by means of random under-sampling. This means that the dataset for each refactoring operation is bounded by the minimum between the number of positive and the number of negative instances (e.g., as we see in Table 3, although we have 41,191

⁸ <https://github.com/mauricioaniche/ck/commit/f0590677271fb413ecfb4c2c5d0ffbaf8444075>.

instances of *Extract Class* refactorings, we have only 10,692 instances of non-refactored classes, and thus, our model is trained on all the 10,692 instances of non-refactored classes + 10,692 randomly sampled instances of refactored classes).

To better measure the impact of this choice, we re-created the Random Forest models using a “Near Miss” under-sampling strategy [78]. While the average absolute difference in precision and recall are 0.116 and 0.052, respectively, it is hard to distinguish which strategy helps the model in improving accuracy. In nine out of 20, Near Miss improved the precision when compared to the random sampling strategy (and thus, random performed better in 11 models), whereas in 12 out of 20, Near Miss improved the recall. As we conjecture that, in the refactoring problem, classes will always be unbalanced by nature, future research is necessary to better understand how to under (or even over) sample. Nevertheless, we acknowledge that a balanced dataset may be different from the distribution that is expected in real life. Hence, a balanced dataset has the potential to lead to less accurate models in practice. Exploring the performance of models trained on datasets that reflect reality is important future work that should be tackled once, as a community, we understand what the real distribution is.

- Our ML pipeline performs scaling and undersampling. Improving the pipeline, e.g., by applying better feature reduction, different balancing strategies, and extensive hyperparameter search, will only make our results better. While developing production-ready models was not the main goal of this paper, we note that our open-source implementation available in our appendix [30] enables it effortlessly. In other words, any researcher or company can download our implementation and datasets, use their available infrastructure, and train (even more accurate) models.
- Code smells are symptoms that might indicate deeper problems in the source code [1]. While code smells have been shown to greatly indicate problematic pieces of code, in this work, we did not use them as features to our model. However, we note that code smells are detected by combinations of proxy metrics (i.e., detection strategies [79]). These proxy metrics are commonly related to the structure of the source code (e.g., complexity/WMC, coupling/CBO) and are highly similar to the structural metrics we use as features (see Table 4). In other words, we train our models with metrics that are similar to the metrics used by the code smells detection strategies. Therefore, we conjecture that using code smells as features would add only a small amount of information for the models to learn from. That being said, making sure that all catalogued code smells are covered by our features is interesting future work which might increase the accuracy of refactoring recommendation models. Our own previous research shows that code smells might be architecture-specific [80], [81], [82], [83], e.g., MVC systems might suffer from different and specific smells than Android systems.

- As we discuss in Section 4, we did not take into account the different reasons a developer might have when deciding to refactor, e.g., to add a new functionality, or to improve testability. These motivations might indeed change (or even help the developer to prioritize) which refactors to apply. Nevertheless, we affirm that the goal of this first study was to explore whether ML can model the refactoring recommendation problem. Given that we observed high accuracy, we can only conjecture that taking the motivation into consideration will only increase the accuracy (or again, help in prioritization) of the models. We leave it as future work.
- We consider our dataset as a set of unordered refactorings. As a contrast, studies in defect prediction consider datasets as a set of ordered events, e.g., they do not mix “past” and “future” when evaluating the accuracy of their models (e.g., [84]). We argue that there is no need for such design, given that we devise a single cross-project model, based on hundreds of thousands of data points from more than 11k projects altogether. In other words, we do not devise one model per project, as commonly done in defect prediction. Thus, we affirm that the model has little chance of memorizing specific classes. Our 10-fold random cross validation (and the individual precision and recall of each fold, that can be seen in our appendix [30]) also gives us certainty that this is not a threat. Nevertheless, to empirically show that our decision of not ordering the dataset does not influence our results, we trained Random Forest models using the first 90 percent of refactorings that have happened (ordered by time) and tested on the remaining 10 percent of refactorings that happened afterwards. We obtained an average accuracy of 87 percent among the 20 refactorings. The individual results per type of refactoring can be seen in our online appendix [30].
- In RQ₃, when studying the generalization of our models, we observed that class-level refactoring models outperformed method- and variable-level refactorings. We took a harder look at our data and noticed that this phenomenon tends to happen when models are built from smaller datasets, F-Droid and Apache. When training our models with GitHub data (the largest dataset), the phenomenon still occurs, although with a smaller difference. Nevertheless, we can not offer a clear explanation on why that happens, based on the data we collected. There might be an unseen factor which we did not collect data and analyze. Future work should understand the reasons for this phenomenon.

6.3 External Validity

Threats to external validity concern the generalization of results.

- Our results are based only on open source projects, which might affect their generalizability to industrial settings. It is worth mentioning, however, that our sample contains many industrial-scale projects that

- span different domains. To the best of our knowledge, this is the most extensive study of ML algorithms for the prediction of refactorings to date. Nevertheless, replicating this research in a large dataset of industry projects is necessary.
- One of our goals was to understand whether ML models trained on a set of systems are able to accurately recommend refactoring operations to improve completely different software systems. We experimented with different ecosystems as an approximation for “completely different software systems”. While we believe this is a reasonable approximation, we are not able to make strong assumptions about the accuracy of those models in large-scale enterprise industrial systems. We suggest that researchers perform case studies together with industrial partners in hopes of providing evidence to support such hypothesis.
 - Moreover, since we considered Java as the language of choice, we cannot be sure that our results carry over to other programming languages. Thus, replications of this study are needed for different programming languages. However, we cannot think of any reason why the results would be different for other imperative object-oriented languages.

7 CONCLUSION

Supervised ML algorithms are effective in predicting refactoring opportunities and might indeed support developers in making faster and more educated decisions concerning what to refactor.

Our main findings show that:

- 1) Random Forest models outperform other ML models in predicting software refactoring;
- 2) Process and ownership metrics seem to play a crucial role in the creation of better models; and
- 3) Models trained with data from heterogeneous projects generalize better and achieve good performance.

More importantly, this paper shows that *ML algorithms can accurately model the refactoring recommendation problem*. We hope that this paper will pave the way for more data-driven refactoring recommendation tools.

Given that we are more confident that ML models might provide accurate recommendations to developers, the next step of this research should work on devising and building the necessary tools to deploy and perform case studies on the efficiency of refactoring recommendation models in the wild.

ACKNOWLEDGMENT

We thank Prof. Dr. Alfredo Goldman (University of São Paulo), Diogo Pina (University of São Paulo), and Matheus Flauzino (Federal University of Lavras) for their feedback on the early steps of this work.

REFERENCES

- [1] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts, *Refactoring: Improving the Design of Existing Code*. Reading, MA, USA: Addison-Wesley, 1999.
- [2] E. A. AlOmar, M. W. Mkaouer, A. Ouni, and M. Kessentini, “Do design metrics capture developers perception of quality? An empirical study on self-affirmed refactoring activities,” in *Proc. 13th ACM/IEEE Int. Symp. Empir. Softw. Eng. Meas.*, 2019, pp. 300–311.
- [3] Y. Kataoka, T. Imai, H. Andou, and T. Fukaya, “A quantitative evaluation of maintainability enhancement by refactoring,” in *Proc. Int. Conf. Softw. Maintenance*, 2002, pp. 576–585.
- [4] R. Leitch and E. Stroulia, “Assessing the maintainability benefits of design restructuring using dependency analysis,” in *Proc. 5th Int. Workshop Enterprise Netw. Comput. Healthcare Ind.*, 2004, pp. 309–322.
- [5] M. Alshayeb, “Empirical investigation of refactoring effect on software quality,” *Inf. Softw. Technol.*, vol. 51, no. 9, pp. 1319–1326, 2009.
- [6] R. Shatnawi and W. Li, “An empirical assessment of refactoring impact on software quality using a hierarchical quality model,” *Int. J. Softw. Eng. Appl.*, vol. 5, no. 4, pp. 127–149, 2011.
- [7] M. Kim, T. Zimmermann, and N. Nagappan, “A field study of refactoring challenges and benefits,” in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, Art. no. 50.
- [8] P. Kruchten, R. L. Nord, and I. Ozkaya, “Technical debt: From metaphor to theory and practice,” *IEEE Softw.*, vol. 29, no. 6, pp. 18–21, Nov./Dec. 2012.
- [9] M. Beller, R. Bholanath, S. McIntosh, and A. Zaidman, “Analyzing the state of static analysis: A large-scale evaluation in open source software,” in *Proc. IEEE 23rd Int. Conf. Softw. Anal. Evol. Reeng.*, 2016, pp. 470–481.
- [10] N. Ayewah, W. Pugh, D. Hovemeyer, J. D. Morgenthaler, and J. Penix, “Using static analysis to find bugs,” *IEEE Softw.*, vol. 25, no. 5, pp. 22–29, Sep./Oct. 2008.
- [11] S. Habchi, X. Blanc, and R. Rouvoy, “On adopting linters to deal with performance concerns in android apps,” in *Proc. 33rd ACM/IEEE Int. Conf. Automated Softw. Eng.*, 2018, pp. 6–16.
- [12] K. F. Tómasdóttir, M. Aniche, and A. Van Deursen, “The adoption of JavaScript linters in practice: A case study on ESLint,” *IEEE Trans. Softw. Eng.*, vol. 46, no. 8, pp. 863–891, Aug. 2020.
- [13] K. F. Tómasdóttir, M. Aniche, and A. van Deursen, “Why and how JavaScript developers use linters,” in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 578–589.
- [14] B. Johnson, Y. Song, E. Murphy-Hill, and R. Bowdidge, “Why don’t software developers use static analysis tools to find bugs?” in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 672–681.
- [15] M. Lanza and R. Marinescu, *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Berlin, Germany: Springer, 2007.
- [16] N. Moha, Y.-G. Gueheneuc, L. Duchien, and A.-F. Le Meur, “DECOR: A method for the specification and detection of code and design smells,” *IEEE Trans. Softw. Eng.*, vol. 36, no. 1, pp. 20–36, Jan./Feb. 2010.
- [17] T. Mariani and S. R. Vergilio, “A systematic review on search-based refactoring,” *Inf. Softw. Technol.*, vol. 83, pp. 14–34, 2017.
- [18] M. O’Keeffe and M. O. Cinnéide, “Search-based refactoring for software maintenance,” *J. Syst. Softw.*, vol. 81, no. 4, pp. 502–516, 2008.
- [19] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, “Recommending refactoring operations in large software systems,” in *Recommendation Systems in Software Engineering*. Berlin, Germany: Springer, 2014, pp. 387–419.
- [20] M. D’Ambros, M. Lanza, and R. Robbes, “Evaluating defect prediction approaches: A benchmark and an extensive comparison,” *Empir. Softw. Eng.*, vol. 17, no. 4/5, pp. 531–577, 2012.
- [21] K. Liu *et al.*, “Learning to spot and refactor inconsistent method names,” in *Proc. 41st Int. Conf. Softw. Eng.*, 2019, pp. 1–12.
- [22] M. I. Azeem, F. Palomba, L. Shi, and Q. Wang, “Machine learning techniques for code smell detection: A systematic literature review and meta-analysis,” *Inf. Softw. Technol.*, vol. 108, pp. 115–138, 2019.
- [23] N. Tsantalis, M. Mansouri, L. M. Eshkevari, D. Mazinanian, and D. Dig, “Accurate and efficient refactoring detection in commit history,” in *Proc. 40th Int. Conf. Softw. Eng.*, 2018, pp. 483–494.
- [24] A. V. Deursen, L. Moonen, A. V. D. Bergh, and G. Kok, “Refactoring test code,” in *Proc. 2nd Int. Conf. Extreme Program. Flexible Processes Softw. Eng.*, 2001, pp. 92–95.
- [25] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, “An empirical analysis of the distribution of unit test smells and their impact on software maintenance,” in *Proc. 28th IEEE Int. Conf. Softw. Maintenance*, 2012, pp. 56–65.
- [26] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, “Are test smells really harmful? An empirical study,” *Empir. Softw. Eng.*, vol. 20, no. 4, pp. 1052–1094, 2015.

- [27] S. Scalabrino, G. Bavota, C. Vendome, M. Linares-Vásquez, D. Poshyvanyk, and R. Oliveto, "Automatically assessing code understandability: How far are we?" in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 417–427.
- [28] S. McIntosh, Y. Kamei, B. Adams, and A. E. Hassan, "The impact of code review coverage and code review participation on software quality: A case study of the qt, VTK, and ITK projects," in *Proc. 11th Work. Conf. Mining Softw. Repositories*, 2014, pp. 192–201.
- [29] S. Kaur and P. Singh, "How does object-oriented code refactoring influence software quality? Research landscape and challenges," *J. Syst. Softw.*, vol. 157, pp. 110–394, 2019.
- [30] M. Aniche, E. Maziero, R. Durelli, and V. Durelli, "Appendix: The effectiveness of supervised machine learning algorithms in predicting software refactoring," appendix: [Onlinel]. Available: <https://zenodo.org/record/3598352>, Dataset: [Online]. Available: <https://zenodo.org/record/3547639>, Final models: [Online]. Available: <https://zenodo.org/record/3598361>, Source code: [Online]. Available: <http://github.com/refactoring-ai/predicting-refactoring-ml> (commit e397ab8f)
- [31] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE Trans. Softw. Eng.*, vol. 20, no. 6, pp. 476–493, Jun. 1994.
- [32] F. Rahman and P. Devanbu, "How, and why, process metrics are better," in *Proc. 35th Int. Conf. Softw. Eng.*, 2013, pp. 432–441.
- [33] L. Madeyski and M. Jureczko, "Which process metrics can significantly improve defect prediction models? An empirical study," *Softw. Qual. J.*, vol. 23, no. 3, pp. 393–422, 2015.
- [34] C. Bird, N. Nagappan, B. Murphy, H. Gall, and P. Devanbu, "Don't touch my code!: Examining the effects of ownership on software quality," in *Proc. 19th ACM SIGSOFT Symp. 13th Eur. Conf. Found. Softw. Eng.*, 2011, pp. 4–14.
- [35] Y. Higo, S. Hayashi, and S. Kusumoto, "On tracking Java methods with git mechanisms," *J. Syst. Softw.*, vol. 165, 2020, Art. no. 110571.
- [36] F. Pedregosa *et al.*, "Scikit-learn: Machine learning in Python," *J. Mach. Learn. Res.*, vol. 12, pp. 2825–2830, 2011.
- [37] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Germany: Springer-Verlag, 2006.
- [38] H. Zhang, "The optimality of naïve bayes," in *Proc. 17th Int. Florida Artif. Intell. Res. Soc. Conf.*, 2004.
- [39] C. Cortes and V. Vapnik, "Support-vector networks," *Mach. Learn.*, vol. 20, pp. 273–297, 1995.
- [40] J. R. Quinlan, *C4.5: Programs for Machine Learning*. San Mateo, CA, USA: Morgan Kaufmann, 1993.
- [41] L. Breiman, "Random forests," *Mach. Learn.*, vol. 45, no. 1, pp. 5–32, 2001.
- [42] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. Cambridge, MA, USA: MIT Press, 2016. [Online]. Available: <http://www.deeplearningbook.org>
- [43] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *Proc. Mach. Learn. Res.*, vol. 37, 2015, pp. 448–456.
- [44] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Trans. Knowl. Data Eng.*, vol. 21, no. 9, pp. 1263–1284, Sep. 2009.
- [45] M. Rapp, E. L. Mencia, and J. Fürnkranz, "Simplifying random forests: On the trade-off between interpretability and accuracy," in *Proc. 1st Workshop Deep Continuous-Discrete Mach. Learn.*, 2019, pp. 1–3.
- [46] B. Kim *et al.*, "Interpretability beyond feature attribution: Quantitative testing with concept activation vectors (TCAV)," in *Proc. Mach. Learn. Res.*, vol. 80, 2018, pp. 2668–2677.
- [47] M. M. Lehman, J. F. Ramil, P. D. Wernick, D. E. Perry, and W. M. Turski, "Metrics and laws of software evolution - The nineties view," in *Proc. 4th Int. Symp. Softw. Metrics*, 1997, pp. 20–32.
- [48] M. M. Lehman, "Laws of software evolution revisited," in *Proc. Eur. Workshop Softw. Process Technol.*, 1996, pp. 108–124.
- [49] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019. [Online]. Available: <http://doi.acm.org/10.1145/3290353>
- [50] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," in *Proc. Int. Conf. Learn. Representations*, 2013, pp. 1–12.
- [51] J. Kerievsky, "Refactoring to Patterns," Addison-Wesley Professional, pp. 464, 2004.
- [52] D. Silva, N. Tsantalis, and M. T. Valente, "Why we refactor? Confessions of GitHub contributors," in *Proc. 24th ACM SIGSOFT Int. Symp. Found. Softw. Eng.*, 2016, pp. 858–870.
- [53] M. Allamanis, E. T. Barr, P. Devanbu, and C. Sutton, "A survey of machine learning for big code and naturalness," *ACM Comput. Surv.*, vol. 51, no. 4, 2018, Art. no. 81.
- [54] A. Hindle, E. T. Barr, M. Gabel, Z. Su, and P. Devanbu, "On the naturalness of software," *Commun. ACM*, vol. 59, no. 5, pp. 122–131, Apr. 2016. [Onlinel]. Available: <http://doi.acm.org/10.1145/2902362>
- [55] V. J. Hellendoorn and P. Devanbu, "Are deep neural networks the best choice for modeling source code?" in *Proc. 11th Joint Meeting Found. Softw. Eng.*, 2017, pp. 763–773. [Onlinel]. Available: <http://doi.acm.org/10.1145/3106237.3106290>
- [56] M. Allamanis, E. T. Barr, C. Bird, and C. Sutton, "Suggesting accurate method and class names," in *Proc. 10th Joint Meeting Found. Softw. Eng.*, 2015, pp. 38–49.
- [57] E. Wong, J. Yang, and L. Tan, "AutoComment: Mining question and answer sites for automatic comment generation," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2013, pp. 562–567.
- [58] S. Jiang, A. Armaly, and C. McMillan, "Automatically generating commit messages from diffs using neural machine translation," in *Proc. 32nd IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2017, pp. 135–146.
- [59] P. Deep Singh and A. Chug, "Software defect prediction analysis using machine learning algorithms," in *Proc. 7th Int. Conf. Cloud Comput. Data Sci. Eng.*, 2017, pp. 775–781.
- [60] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. 27th Int. Conf. Neural Inf. Process. Syst.*, 2014, pp. 3104–3112.
- [61] C. Buciluă, R. Caruana, and A. Niculescu-Mizil, "Model compression," in *Proc. 12th ACM SIGKDD Int. Conf. Knowl. Discov. Data Mining*, 2006, pp. 535–541.
- [62] M. Kondo, C.-P. Bezemer, Y. Kamei, A. E. Hassan, and O. Mizuno, "The impact of feature reduction techniques on defect prediction models," *Empir. Softw. Eng.*, vol. 24, pp. 1925–1963, 2019.
- [63] C. Sadowski, J. V. Gogh, C. Jaspan, E. Söderberg, and C. Winter, "Tricorder: Building a program analysis ecosystem," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 598–608.
- [64] C. Sadowski, E. Aftandilian, A. Eagle, L. Miller-Cushon, and C. Jaspan, "Lessons from building static analysis tools at Google," *Commun. ACM*, vol. 61, no. 4, pp. 58–66, 2018.
- [65] R. Kumar, C. Bansal, C. Maddila, N. Sharma, S. Martelock, and R. Bhargava, "Building Sankie: An AI platform for DevOps," in *Proc. IEEE/ACM 1st Int. Workshop Bots Softw. Eng.*, 2019, pp. 48–53.
- [66] Y. Goldberg and O. Levy, "word2vec explained: Deriving Mikolov *et al.*'s negative-sampling word-embedding method," 2014, *arXiv:1402.3722* [cs.CL].
- [67] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv: 1810.04805* [cs.CL].
- [68] T. Mens and T. Tourwe, "A survey of software refactoring," *IEEE Trans. Softw. Eng.*, vol. 30, no. 2, pp. 126–139, Feb. 2004.
- [69] A. A. B. Baqais and M. Alshayeb, "Automatic software refactoring: A systematic literature review," *Softw. Qual. J.*, vol. 28, pp. 459–502, 2020.
- [70] M. Zhang, T. Hall, and N. Baddoo, "Code bad smells: A review of current knowledge," *J. Softw. Maintenance Evol.: Res. Practice*, vol. 23, no. 3, pp. 179–202, 2011.
- [71] J. Al Dallal, "Identifying refactoring opportunities in object-oriented code: A systematic literature review," *Inf. Softw. Technol.*, vol. 58, pp. 231–249, 2015.
- [72] S. Singh and S. Kaur, "A systematic literature review: Refactoring for disclosing code smells in object oriented software," *Ain Shams Eng. J.*, vol. 9, no. 4, pp. 2129–2151, 2018.
- [73] J. A. M. Santos, J. B. Rocha-Junior, L. C. L. Prates, R. S. do Nascimento, M. F. Freitas, and M. G. de Mendonça, "A systematic review on the code smell effect," *J. Syst. Softw.*, vol. 144, pp. 450–477, 2018.
- [74] E. Fernandes, J. Oliveira, G. Vale, T. Paiva, and E. Figueiredo, "A review-based comparative study of bad smell detection tools," in *Proc. 20th Int. Conf. Eval. Assessment Softw. Eng.*, 2016, pp. 18:1–18:12.
- [75] G. Rasool and Z. Arshad, "A review of code smell mining techniques," *J. Softw.: Evol. Process*, vol. 27, no. 11, pp. 867–895, 2015.
- [76] M. Mohan and D. Greer, "A survey of search-based refactoring for software maintenance," *J. Softw. Eng. Res. Develop.*, vol. 6, no. 1, pp. 3–55, 2018.
- [77] M. I. Azeeem, F. Palomba, L. Shi, and Q. Wang, "Machine learning techniques for code smell detection: A systematic literature review and meta-analysis," *Inf. Softw. Technol.*, vol. 108, pp. 115–138, 2019.

- [78] I. Mani and I. Zhang, "KNN approach to unbalanced data distributions: A case study involving information extraction," in *Proc. Workshop Learn. Imbalanced Datasets*, vol. 126, 2003, pp. 1–7.
- [79] R. Marinescu, "Detection strategies: Metrics-based rules for detecting design flaws," in *Proc. 20th IEEE Int. Conf. Softw. Maintenance*, 2004, pp. 350–359.
- [80] S. Goularte Carvalho, M. Aniche, J. Veríssimo, R. Durelli, and M. Gerosa, "An empirical catalog of code smells for the presentation layer of android apps," *Empir. Softw. Eng.*, vol. 24, no. 6, pp. 3546–3586, 2019.
- [81] M. Aniche, G. Bavota, C. Treude, M. Gerosa, and A. van Deursen, "Code smells for model-view-controller architectures," *Empir. Softw. Eng.*, vol. 23, no. 4, pp. 2121–2157, 2018.
- [82] M. Aniche, G. Bavota, C. Treude, A. van Deursen, and M. Gerosa, "A validated set of smells in model-view-controller architectures," in *Proc. IEEE Int. Conf. Softw. Maintenance Evol.*, 2016, pp. 233–243.
- [83] M. Aniche, C. Treude, A. Zaidman, A. van Deursen, and M. A. Gerosa, "SATT: Tailoring code metric thresholds for different software architectures," in *Proc. IEEE 16th Int. Work. Conf. Source Code Anal. Manipulation*, 2016, pp. 41–50.
- [84] L. Pascarella, F. Palomba, and A. Bacchelli, "Fine-grained just-in-time defect prediction," *J. Syst. Softw.*, vol. 150, pp. 22–36, 2019.



Rafael Durelli received the PhD degree in computer science from the University of São Paulo (USP), Brazil. He is an assistant Professor with the Department of Computer Science, Federal University of Lavras, Brazil.



Vinicius H. S. Durelli received the PhD degree in computer science from the University of São Paulo (USP), Brazil, and undertook postdoctoral research at the University of Groningen, The Netherlands. He is an assistant professor with the Department of Computer Science, Federal University of São João del Rei, Brazil. His research interests include software engineering, with particular emphasis on software testing and refactoring.



Maurício Aniche received the PhD degree in computer science from the University of São Paulo (USP), Brazil. He is an assistant professor in software engineering at Delft University of Technology, The Netherlands. His research interests include software quality, maintenance, and evolution.



Erick Maziero received the PhD degree in computer science from the University of São Paulo (USP), Brazil. He is an adjunct professor with the Department of Computer Science, Federal University of Lavras, Brazil. His research interests include artificial intelligence and natural language processing.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csdl.

A Fast Clustering Algorithm for Modularization of Large-Scale Software Systems

Navid Teymourian[✉], Habib Izadkhah[✉], and Ayaz Isazadeh

Abstract—A software system evolves over time in order to meet the needs of users. Understanding a program is the most important step to apply new requirements. Clustering techniques through dividing a program into small and meaningful parts make it possible to understand the program. In general, clustering algorithms are classified into two categories: hierarchical and non-hierarchical algorithms (such as search-based approaches). While clustering problems generally tend to be NP-hard, search-based algorithms produce acceptable clustering and have time and space constraints and hence they are inefficient in large-scale software systems. Most algorithms which currently used in software clustering fields do not scale well when applied to large and very large applications. In this paper, we present a new and fast clustering algorithm, FCA, that can overcome space and time constraints of existing algorithms by performing operations on the dependency matrix and extracting other matrices based on a set of features. The experimental results on ten small-sized applications, ten folders with different functionalities from Mozilla Firefox, a large-sized application (namely ITK), and a very large-sized application (namely Chromium) demonstrate that the proposed algorithm achieves higher quality modularization compared with hierarchical algorithms. It can also compete with search-based algorithms and a clustering algorithm based on subsystem patterns. But the running time of the proposed algorithm is much shorter than that of the hierarchical and non-hierarchical algorithms. The source code of the proposed algorithm can be accessed at <https://github.com/SoftwareMaintenanceLab>.

Index Terms—Software clustering, software modularization, software maintenance, software comprehension, architecture recovery

1 INTRODUCTION

SOFTWARE plays a key role in government agencies and organizations and as an interface, it has an important role in communications. Over the time, the requirements of an organization will change according to the environmental conditions and software engineers need to make changes in the software system to meet the needs of the organization. To make changes to the software, developers require to understand the software structure (software architecture). During software maintenance, software engineers spend a considerable amount of time on program comprehension activities [1]. Because of the complex structure and relationships, understanding the structure of a large software system and applying new changes is not an easy task. Recovering software architecture to understand software systems is therefore particularly important because it facilitates the maintenance and evolution of software systems [2], [3].

The software architecture recovery aims to use techniques to partition a software system into meaningful subsystems (modules) [4], [5]. For this reason, numerous attempts have been done to develop techniques for extracting software architecture automatically. One of these techniques is clustering [5]. “The objective of software clustering is to reduce the

complexity of a large software system by replacing a set of artifacts with a cluster, a representative abstraction of all artifacts grouped within it. Thus, the obtained decomposition is easier to understand” [6]. The purpose of clustering is to divide a software system into clusters (modules) so that the relationships between artifacts in a cluster are maximized (i.e., cohesion) and the relationships between clusters (i.e., coupling) are minimized. Fig. 1 illustrates the clustering process of a software system. The input of a clustering algorithm is an Artifact Dependency Graph (ADG) which is constructed from source code [6]. The nodes of this graph indicate the artifacts (e.g., class, method, file, function, etc.), and the links indicate the relationships between artifacts (e.g., calling dependency, inheritance dependency, semantic dependency, etc.).

In general, clustering algorithms are classified into hierarchical and non-hierarchical categories (such as search-based and graph-based) [7]. In the hierarchical algorithms, artifacts are first considered as separate clusters and are then merged in a repeating process. These algorithms require a data table representing relationships between artifacts and a table that shows similarities between artifacts. These tables should be updated at each step of the clustering process. In very large software systems, these tables will be very big and it will also be time consuming to calculate the similarities between the artifacts. For example, the LIMBO algorithm [6], at the first step of the clustering process, requires about 15×10^{11} operations for a graph with 10,000 artifacts. Given all the clustering steps, the number of operations will be much higher than this number. This algorithm requires a large amount of time for large graphs, which reduces its efficiency.

In the literature, because of the NP-Hardness of clustering problem, search-based methods (such as genetic algorithm)

• The authors are with the Department of Computer Science, University of Tabriz, Tabriz 5166616471, Iran. E-mail: nvd.teymourian@gmail.com, {izadkhah, isazadeh}@tabrizu.ac.ir.

Manuscript received 22 Jan. 2020; revised 23 Aug. 2020; accepted 3 Sept. 2020. Date of publication 7 Sept. 2020; date of current version 18 Apr. 2022. (Corresponding author: Habib Izadkhah.)

Recommended for acceptance by L. Tan.

Digital Object Identifier no. 10.1109/TSE.2020.3022212

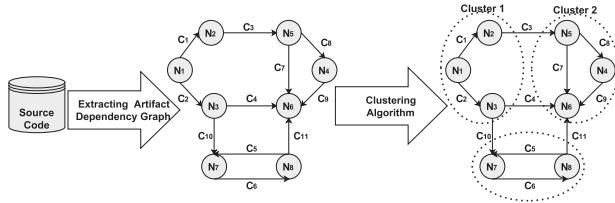


Fig. 1. Software clustering process.

have been widely used [8], [9]. Search-based algorithms are an effective way to solve the clustering problem [9]. The search-based algorithms may take a long time to execute, if computing the fitness function in each iteration takes a long time to perform. Therefore, the main drawback of these methods is that they work very slowly when faced with large-sized graphs. Due to the problems of the existing algorithms, in this paper, we designed a new clustering algorithm that operates on artifact dependency matrix constructed from source code.

The main problem addressed in this paper is scalability in terms of running time. We aim to provide a deterministic clustering algorithm that its running time grows slowly as the input size increases. We claim that by performing a series of simple operations on the dependency matrix and extracting other matrices based on a set of features, a developer can quickly cluster a software system while the clustering quality is acceptable. The proposed algorithm was tested on ten small-sized software systems, ten folders of Mozilla Firefox, and two large and very large software systems. The results showed that the algorithm achieves acceptable clustering quality according to evaluation criteria (internal and external criteria), in less run time, compared to the tested algorithms.

This paper is structured as follows. In Section 2, we discuss related work on software clustering. Section 3 introduces the proposed algorithm, and we present the experimental setup in Section 4. The result of research and threats to validity are discussed in Sections 5 and 6, respectively. Finally, Section 7 is the conclusions of this research and future work.

2 RELATED WORK

So far many algorithms have been developed for clustering, but given the NP-Hardness of clustering problem, designing a proper clustering algorithm is a difficult work. In this section, first, a description of the clustering algorithms classification is presented, and then some state-of-the-art clustering algorithms are described.

Most software clustering algorithms fall into two major categories: agglomerative hierarchical and search-based algorithms, and there also are a few algorithms that are graph-based [8] and pattern-based. Hierarchical algorithms are greedy and phased and at each stage, the most similar artifacts are merged. Single linkage, complete linkage, average linkage, weighted average link are some of the classical hierarchical algorithms [10]. Maqbool and Babri presented two hierarchical clustering algorithms for software architecture recovery, namely combined Algorithm (CA) and Weighted combined Algorithm (WCA) [11]. Weighted combined Algorithm is a popular hierarchical clustering technique. Considering the ways to compute the similarity

between artifacts, Unbiased Ellenberg (UE) and Unbiased Ellenberg-NM (UENM), WCA has two versions WCA-UE and WCA-UENM. Another popular hierarchical clustering algorithm is scaLable InforMation BOtleneck (LIMBO) [6]. This algorithm employs information theory and entropy concepts for software clustering. In [10], the cooperative clustering technique (CCT) as a consensus-based technique is utilized for the software clustering problem. In this technique more than one similarity measure cooperates during the hierarchical clustering process.

In search-based algorithms, the clustering process is considered as an optimization problem [5]. Then, heuristic or meta-heuristic search methods are used to find the near-optimal solution. The search process is guided and evaluated by a quality function (objective function) that shows how appropriate the solution is. The searching process in these algorithms is performed in two global and local ways. In the global search-based systems algorithm, the entire search space is considered the solution space. In these algorithms, an operator is intended to discover new areas in the search space. Local search algorithms start from a selective solution and then gradually move from the current solution to the neighboring solution using search changes. However, these algorithms inherently have the problem of being trapped in local optima solution [8]. It is also possible to combine global and local search algorithms. Graph-based clustering algorithms can be applied to various areas such as social networking, image segmentation, and software clustering. Mohammadi and Izadkhah in [7] use a neighboring tree generated from the ADG to cluster a software system. The clustering quality obtained by this algorithm is better than hierarchical methods and worse than evolutionary methods. Spectral methods [12] use algebraic properties of the graph, such as eigenvalues and eigenvectors in the corresponding Laplacian matrix to perform clustering. ACDC [13] is a pattern-based algorithm that was introduced by Tzerpos and Holt. It uses several patterns to cluster code artifacts. Several previous studies, such as [14], [15], [16], have shown that ACDC performed well on the tested applications.

Depending on the features used for clustering, clustering techniques can be categorized in terms of structural and non-structural (or semantic). Some algorithms only use structural properties and some others use non-structural properties such as comments and name of artifacts. Some algorithms combine both of them. Table 1 shows some clustering algorithms with different categories. Here, we briefly describe some clustering algorithms.

Bunch: the Bunch toolset was proposed by Mitchell for the software clustering problem [5], [17]. Two search-based algorithms with different searching strategy namely genetic algorithm (GA) and Hill-climbing were employed in this toolset for the software clustering problem. Next Ascent Hill climbing (NAHC) and Steepest Ascent Hill climbing (SAHC) are two versions of the Hill-climbing algorithm which are presented in the Bunch. The algorithms presented in the Bunch use a real-valued encoding to represent the solutions. The search space produced by this encoding equals n^n , where n is the number of artifacts. Bunch input is a call dependency graph (CDG) made from source code. The output of the Bunch is clustering with minimum coupling and maximum cohesion among clusters.

TABLE 1
Some Search-Based Software Clustering Algorithms

Name	Type of algorithm	Type of objective function	Structural/Semantic Features
Bunch-NAHC and Bunch-SAHC [17]	Local Search	Single Objective	Structural
E-CDGM [18]	Local Search	Single Objective	Structural
Large neighborhood search [19]	Local Search	Single Objective	Structural
HC-SMCP [20]	Local Search	Single Objective	Structural
SHC [21]	Local Search	Single Objective	Semantic
Bunch-GA [5]	Global Search	Single Objective	Structural
DAGC [22]	Global Search	Single Objective	Structural
A multi-agent evolutionary algorithm [23]	Global Search	Single Objective	Structural
Harmony search [24]	Global Search	Multi-Objective	Structural
GA-SMCP [20]	Global Search	Single Objective	Structural
Hyper-heuristic approach [25]	Global Search	Multi-Objective	Structural
ECA and MCA [9]	Global Search	Multi-Objective	Structural
Estimation of distribution approach [26]	Global Search	Single Objective	Structural
EoD, CGH, CGoH [8]	Global Search	Multi-Objective	Structural and Semantic
Search based multiobjective software remodularization [27]	Global Search	Multi-Objective	Structural
Multiple relationship factors [28]	Global Search	Multi-Objective	Structural
Interactive evolutionary optimization [29]	Global Search	Multi-Objective	Structural
GAKH [30]	Global Search	Single Objective	Structural
MaABC [31]	Global Search	Multi-objective	Structural
ILOF [32]	Global Search	Support multi-objective	Structural

DAGC: similar to Bunch, this algorithm [22] employs genetic algorithm to perform clustering. But, each chromosome is encoded by a permutation of graph's nodes. The search space in this algorithm equals $n!$. It can be said that it has less search space than Bunch, in contrast to it uses a sophisticated encoding method.

Maximizing Cluster Approach (MCA) and Equal-size Cluster Approach (ECA) [9]: both algorithms are multi-objective and use two-archive Pareto optimal genetic algorithm to optimize the objectives. The objectives used for clustering in MCA are “maximize the number of edges inside the clusters”, “minimize the number of edges between all the clusters”, “maximize the number of clusters”, “maximize the TurboMQ”, and “minimize the number of single-member clusters”; and the objectives used in the ECA are similar to the MCA with one difference. ECA replaced the fifth objective of the MCA with “minimizing the difference between the maximum and minimum number of modules in a cluster.”

In semantic-based algorithms, how words are selected and which semantic analysis method employed is the main reason for the differences between these algorithms. Garcia *et al.* [33] proposed a hierarchical clustering algorithm-named Architecture Recovery using Concerns (ARC)- that uses concerns to perform an architecture recovery. ARC considers textual information (identifiers and comments) extracted from source code and extracts concerns based on Latent Dirichlet Allocation (LDA) model. Some studies, e.g., [16], have shown that this algorithm has acceptable accuracy.

Corazza *et al.* [34] proposed a natural language processing based clustering approach that partitions textual information (identifiers and comments) into different zones. The zones are weighted based on the Expectation-Maximization algorithm and then clustered by hierarchical agglomerative technique. Using the Hill-climbing algorithm, in [21], a semantic-based clustering algorithm, namely SHC, was presented which uses artifact names and comments for semantic analysis. Taking into account syntactic features such as call dependency and inheritance dependency and semantic features such as code comment and identifier name, Misra *et al.* [35] proposed an algorithm for clustering.

Most software clustering algorithms use static dependencies between artifacts. Xiao and Tzerpos [36] presented an approach for investigating the dynamic dependencies. The results of their experiments on some applications showed that dynamic clusters have significant competencies.

To sum up, the main limitations of the existing algorithms are:

- Search-based algorithms, such as genetic algorithms, are usually used to cluster software systems [8], [9]. Because of their exploration and exploitation ability, they can produce good quality answers. But on large applications, they are very time-consuming. Normally it takes more than a month for graphs with more than 10,000 nodes without parallelization to find a proper clustering. It should be noted, however, that parallelization cannot greatly reduce time. This is because the software systems selected for clustering in these methods are small- or medium-sized, e.g., see [8], [9], [17], [20], [24], [37]. Also, because evolutionary algorithms are stochastic, they need to be executed many times, which for large graphs is practically impossible because they are time-consuming.
- Hierarchical algorithms require a data table that represents relationships between artifacts and a table that shows similarities between artifacts. In very large software systems, these tables will be very big and it will also be very time consuming to calculate similarities between artifacts. For example, the LIMBO algorithm for a software system with 10,000 artifacts requires about 15×10^{11} computations, at the first step of the clustering process.

3 PROPOSED CLUSTERING ALGORITHM

The dependency graph is a mathematical way to model the relationships between artifacts. Let x and y denote two artifacts (two nodes in the graph) so that an edge between the two artifacts x and y indicate the existing dependency between them. In the dependency matrix corresponding to

the dependency graph, the intersection of two nodes is placed one if there exists an edge between them, and zero if the two nodes are not connected.

In this paper, we cluster software systems by defining a series of operations on the dependence matrix and extracting several features from it. We derive additional matrices from the dependence matrix, based on a set of features, as well as applying mathematical operations on the matrices to perform the clustering. In this algorithm, the dependency graph is the input of the algorithm and a modularized ADG is the output of the algorithm. Because the input of the proposed algorithm is the dependency graph, so the algorithm is independent of the programming language used. Tools such as Understand (<https://scitools.com/>) or NDepend (<https://ndepend.com>) can be used to extract the dependency graph from the source code of most programming languages. The proposed algorithm aims to maximize intra-connectivity within the clusters (i.e., cohesion) and minimize inter-dependencies between the clusters (i.e., coupling).

3.1 Algorithm Steps (FCA)

- 1) Input: dependency matrix constructed from the artifact dependency graph,
- 2) The neighborhood degree matrix is created from the dependency matrix. This matrix shows degree information. Each node has several neighbors, and this matrix shows the degree of neighbor nodes for each node. The steps to build this matrix are as follows:
 - a) The number of rows and columns in this matrix is equal to the number of nodes in the artifact dependency graph.
 - b) Let x and y denote the row number (node number) and column number in the matrix, respectively. For each node x in the dependency graph connected to node y , the degree of node y is placed at the intersection from x to y in the neighborhood degree matrix.

We use this matrix to select nodes for clustering. High-degree nodes and nodes that are connected to high-degree nodes are not good options to start the clustering process. To perform clustering, the algorithm starts with nodes that have a small degree and are not connected to high-degree nodes. We call these nodes “border” nodes. Larger-degree nodes tend to absorb the rest of the nodes and create larger clusters. In steps 3 and 4, the nodes are ranked and used as primary nodes in the clustering process.

- 3) To rank the nodes, the numbers in each row of the neighborhood matrix are summed up and placed in an $n \times 1$ matrix named *Sum* matrix, where n is the number of nodes. The entries of this matrix are S_1, S_2, \dots, S_n .

The *Sum* matrix is not enough to prioritize nodes. Because, in large graphs, the number of nodes in which the sum of their rows is equal is large. Therefore, in order to prioritize the nodes, it is necessary to normalize this matrix. Step 4 is used for this purpose.

- 4) To normalize the *Sum* matrix, for each node x (row x), the following equation is calculated and the

results are saved in an $n \times 1$ matrix named *Effect* matrix. Let *NDM* denotes the neighborhood degree matrix, the entries of *Effect* matrix, E_1, E_2, \dots, E_n , are calculated as follows:

$$E_x = \frac{S_x}{\sum_{i=1}^n (k_i \times S_i)}, \quad (1)$$

where

$$k_i = \begin{cases} 0, & NDM_{x,i} = 0; \\ 1, & NDM_{x,i} \neq 0. \end{cases}$$

The numbers in the *Effect* matrix are sorted in descending order. Node x having the largest E_x is used as the first node for clustering. It is important to note that the largest E_x is for a border node and the clustering process starts with this node.

- 5) The clustering steps start from the first node in the *Effect* matrix as follows:
 - a) For node x , the algorithm finds node y which has the lowest numeric value in the row associated with x in the neighborhood matrix. The reverse of this rule must hold: for the node y found by the algorithm, the node x must also have the lowest numeric value in the row associated with y in the neighborhood matrix. In such a case, the node x is co-clustered with the node y . Otherwise, the node x is added to an array named *Incompatible*.
 - b) Repeat step 5(a) for all nodes in *Effect* matrix. If the addressed node has already been clustered, it will be ignored.
- 6) Clustering all nodes in array *Incompatible*. For node x , the algorithm finds node y that has the first or second smallest numerical value in the row associated with x in the neighborhood matrix. The reverse of this rule must hold: for the node y found by the algorithm, the node x must also have the first or second smallest numerical value in the row associated with y in the neighborhood matrix. In such a case, the node x is co-clustered with the node y . Otherwise, the node x is added to an array named *Closed*.
- 7) The intuition behind steps 5 and 6 is to cluster nodes that are related to each other and also have a lower degree. This will reduce coupling. These steps prevent the formation of very large clusters.
- 8) The clusters obtained in steps 5 and 6 are merged if they have at least one node in common. The number of clusters obtained in steps 5 and 6 is high. This step reduces the number of clusters by merging some of them and increases cohesion.
- 9) Clustering all nodes in array *Closed*. Up to this step, there may be nodes that have not been clustered. All these nodes are in the array *Closed*. Using one of the following steps, we determine the cluster of these nodes.
 - a) There are nodes in array *Closed* that are only connected to one cluster. These nodes merge with the related clusters. The next condition is considered for the remaining nodes.

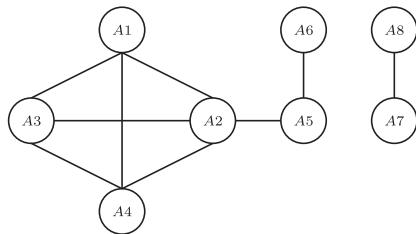


Fig. 2. A sample of an artifact dependency graph.

TABLE 2
Dependency Matrix Constructed From Fig. 2

	1	2	3	4	5	6	7	8
1	0	1	1	1	0	0	0	0
2	1	0	1	1	1	0	0	0
3	1	1	0	1	0	0	0	0
4	1	1	1	0	0	0	0	0
5	0	1	0	0	0	1	0	0
6	0	0	0	0	1	0	0	0
7	0	0	0	0	0	0	0	1
8	0	0	0	0	0	0	1	0

- b) For node x in array *Closed*, all nodes, y , which have an edge to node x are extracted in the dependency matrix. The node x goes to the cluster that has the least amount of *inter-connectivity*. Intuitively, *inter-connectivity* measures a cluster coupling. The lower the value of this relationship, the lower the coupling of a cluster, which is desirable.

Let X , Y , and Z denote the sum of the outer edges of the cluster containing the node y , the sum of E_x values of nodes in the cluster containing the node y , and the number of nodes in the cluster containing the node y , respectively. We have

$$\text{inter-connectivity} = \frac{X + Y}{Z}. \quad (2)$$

The value of $\frac{X}{Z}$ may be the same for clusters with different sizes and may not show the value of the coupling well. While one is superior to the other. That's why Y is used. If the *inter-connectivity* value for the clusters is equal, the next condition is investigated.

- c) The nodes go to a cluster that has the most relationship with the members of that cluster. If the number of relationships is the same, the node go to a cluster that has fewer members.

Using an example, we illustrate the steps of the proposed algorithm. Artifact dependency graph (ADG) of a sample software is shown in Fig. 2. Step 1- Table 2 shows the dependency matrix constructed from the ADG shown in Fig. 2.

Step 2- The neighborhood degree matrix is constructed for all nodes, as shown in Table 3. This matrix shows the degree of neighbor nodes for each node. For example, row 1 shows the neighbors of node 1, while row 3 shows the neighbors of node 3.

Step 3- The *Sum* matrix is shown in Table 4. This matrix represents the sum of the rows in Table 3.

TABLE 3
The Neighborhood Degree Matrix

	1	2	3	4	5	6	7	8
1	0	4	3	3	0	0	0	0
2	3	0	3	3	2	0	0	0
3	3	4	0	3	0	0	0	0
4	3	4	3	0	0	0	0	0
5	0	4	0	0	0	1	0	0
6	0	0	0	0	2	0	0	0
7	3	0	0	0	0	0	0	1
8	3	0	0	0	0	0	1	0

TABLE 4
The Sum Matrix

Sum
S ₁
S ₂
S ₃
S ₄
S ₅
S ₆
S ₇
S ₈

TABLE 5
The Effect Matrix

Effect
E ₁
E ₂
E ₃
E ₄
E ₅
E ₆
E ₇
E ₈

Step 4- According to Eq. (1), the *Effect* matrix is created, as shown in Table 5. For example:

$$E_1 = S_1 / (S_2 + S_3 + S_4) = 10 / (11 + 11 + 10) = 0.32.$$

The calculated *Effect* matrix is sorted in descending order as

$$\text{Effect name} = [E_7, E_8, E_6, E_5, E_1, E_3, E_4, E_2].$$

Step 5- The clustering process starts from the first node in the *Effect* matrix.

- 1) Node 7 can be co-clustered with a node that has the lowest numerical value in the row of Table 3, i.e., node 8. The important point is that node 8 must also have the lowest number in its corresponding row with node 7. This condition is true.
 - a) First cluster {7, 8}
 - b) Since node 8 is in the first cluster, then it is not checked in *Effect* matrix.
- 2) Node 6 can be co-clustered with a node that has the lowest numerical value in the row of Table 3, i.e.,

TABLE 6
The Description of Tested Software Systems

Name	Description	#Files	#Links
compiler	A small compiler developed at the University of Toronto	13	32
nos	A file system	16	52
boxer	Graph drawing tool	18	29
spdbs	A tool to analyze several proteins at the same time	21	33
ispell	Spelling and typographical error correction software	24	103
cald	Program dependency analysis tool	26	64
rccs	System used to manage multiple revisions of files	29	163
star	A program understanding tool	36	89
bison	Parser generator	37	179
cia	Program dependency graph generator for C programs	38	87

node 5. Node 5 also has the lowest number in its corresponding row with node 6. Thus

- a) Second cluster {5, 6}
- b) Since node 5 is in the first cluster, then it is not checked in *Effect* matrix.

After addressing the condition above for node 1, third cluster is {1, 3, 4}

- 3) Node 2 can be co-clustered with a node that has the lowest numerical value in its corresponding row of Table 3, which is node 5.
- a) since the lowest number in row 5 is equal to 1, which corresponds to node 6, so node 2 doesn't cluster with node 3.
- b) the condition is not fulfilled and node 2 is added to array *Incompatible*.

Step 6- Clustering array *Incompatible* starts from its first node i.e., node 2. In the neighborhood matrix, the first and second small numbers in row 2 are related with nodes 1, 3, 4 and 5. That is, node 2 can be clustered with these nodes if the inverse of these relationships are also true.

- 1) The first and second small numbers in row 1 (i.e., node 1) are related to nodes 2, 3 and 4; so the condition is true, and node 2 can be clustered with node 1.
- 2) This step is also applied on nodes 3, 4 and 5.
- 3) Fourth cluster {2, 1, 3, 4, 5}

Step 7- The second, third and fourth clusters are merged due to having common nodes, and the final clustering is as follows:

$$\text{First cluster} = \{1, 2, 3, 4, 5, 6\}, \quad \text{Second cluster} = \{7, 8\}.$$

Step 8- Due to the small size of the dependency graph used, array *Closed* is empty, and thus step 8 is not checked.

4 EXPERIMENTAL SETUP

To evaluate the proposed algorithm, it is necessary to mention the following.

4.1 Software System

Software systems play an important role in the evaluation and comparison of clustering algorithms. We selected ten real-world small-sized applications, as shown in Table 6. In this table, the number of links indicates the number of relationships between the artifacts within the dependency graph, as described in the Introduction. Mozilla Firefox¹ is an open-

source software system. Based on the Open hub site report, more than 13,000 developers work on this application. This application has 55 folders (clusters), so we selected ten of them with different functionalities and sizes. The names and specifications of these folders are presented in Table 7.

In addition to the above applications, two large and very large applications, namely ITK, Chromium, are selected. In place of those large-sized projects, we included ITK (including 7,310 files). We also included a very large project, Chromium (including 18,698 files). Detailed information about these projects can be found in Table 8.

4.2 Expert Decompositions

Expert decomposition (other names: ground-truth architecture or authoritative decomposition) is employed to evaluate the accuracy of the clustering algorithms [7], [10], [14], [15], [38]. An algorithm is reliable if its modularization result is close to decomposition provided by an expert [10]. In large projects, the directory structure of the project originally usually reflects the architecture of the project [39].

In this paper, the developer preview version of the Mozilla Firefox has been selected, because there is a credible (human) expert decomposition (the directory structure) of that. It is important to note that this version is a stable version. For example, folder DB has 97 files organized by the developers of this software in four sub-folders. Our method considers all of 97 files as flat and the aim is to determine how much the algorithm can achieve a clustering similar to the decomposition of Mozilla Firefox's developers.

We also used the ground-truth architectures created by Lutellier *et al.* [15] for ITK and Chromium applications to evaluate the accuracy of the clustering algorithms.²

4.3 Assessment of Results

In the literature, there are many software clustering algorithms and thus to determine the appropriate algorithm, some metrics were presented for determining the quality of clustering obtained by these algorithms. There exist generally two types of metrics for determining the quality of clustering: internal metrics, and external metrics. Internal metrics are independent of any ground-truth architecture, which calculate the quality of the recovered architectures.

TurboMQ presented in [5] is one of the internal metrics which is used in many research papers to evaluate the quality of the recovered architectures e.g., [9], [15], [40]. This metric is defined as follows:

$$\text{TurboMQ} = \sum_{i=1}^k \frac{2A_i}{2A_i + \sum_{j=1}^k (E_{i,j} + E_{j,i})}, \quad (3)$$

where A_i represents the internal communication into cluster i , E_{ij} represents the external communication between two clusters i and j . The higher TurboMQ value indicates better clustering.

The MoJoFM metric is also used to evaluate clustering techniques [41]. This metric is a well-known and widely used external assessment (e.g., see [7], [8], [10], [15], [37]). In

1. <https://ftp.mozilla.org/pub/firefox/releases/devpreview/1.9.3a4/source/>

2. The ground-truth architecture for ITK and Chromium are available at <http://asset.uwaterloo.ca/ArchRecovery>

TABLE 7
Properties of the Selected Folders

Folder Name	#Files	#Links	#Modules	Some Folder Functionalities
Accessible	179	293	8	enabling as many people as possible to use Web sites, even when those people's abilities are limited in some way. Files for accessibility (i.e., MSAA (Microsoft Active Accessibility), ATK (Accessibility Toolkit, used by GTK+ 2) support files).
Browser	45	45	4	Contains the front end code (in XUL, Javascript, XBL, and C++) for the Firefox browser Contains the front end code for the DevTools (Scratchpad, Style Editor, etc.) . Contains images and CSS files to skin the browser for each OS (Linux, Mac and Windows)
Build	21	4	2	Miscellaneous files used by the build process.
Content	881	2948	13	The data structures that represent the structure of Web pages (HTML, SVG, XML documents, elements, text nodes, etc.) and also implement some behaviors associated with those objects, such as link handling, form control behavior, and form submission. This directory also contains the code for XUL, XBL, XTF as well as the code implementing XSLT and event handling.
Db	97	494	4	Container for database-accessing modules.
Dom	163	324	5	IDL definitions of the interfaces defined by the DOM specifications The parts of the connection between JavaScript and the implementations of DOM objects Implementations of a few of the core "DOM Level 0" objects, such as window, window.navigator, window.location, etc.
Extensions	179	206	13	Contains several extensions to mozilla, which can be enabled at compile-time Implementation of the negotiate auth method for HTTP and other protocols. Has code for SSPI, GSSAPI, etc. Permissions backend for cookies, images, etc., as well as the user interface to these permissions and other cookie features. Support for the datetime protocol; Support for the finger protocol. A two-way bridge between the CLR/.NET/Mono/C# etc. world and XPCOM Implementation of W3C's Platform for Privacy Preferences standard. Support for implementing XPCOM components in python. Support for accessing SQL databases from XUL applications; Support for Webservices.
Gfx	342	644	7	Contains interfaces that abstract the capabilities of platform specific graphics toolkits along with implementations on various platforms These interfaces provide methods for things like drawing images, text, and basic shapes It also contains basic data structures such as points and rectangles used here and in other parts of Mozilla.
Intl	573	957	7	Internationalization and localization support; Code for "sniffing" the character encoding of Web pages Code for dealing with Complex Text Layout, related to shaping of south Asian languages Code related to determination of locale information from the operating environment Code that converts (both ways: encoders and decoders) between UTF-16 and many other character encodings Code related to implementation of various algorithms for Unicode text, such as case conversion.
Ipc	391	59	4	Container for implementations of IPC (Inter-Process Communication).

external assessment, the automatically prepared clustering, A, is compared with the decompositions prepared by human experts, B [10]. The value of MoJoFM is between 0 and 100, and the higher value means the more proximity between clustering generated by an algorithm and decomposition created by an expert and hence better results [10]. The MoJoFM is calculated by Eq. (4):

$$\text{MoJoFM}(A, B)(\%) = 1 - \frac{\text{mno}(A, B)}{\max(\text{mno}(\forall A, B))}, \quad (4)$$

where $\text{mno}(A, B)$ denotes the minimum operations required for converting clustering A to clustering B.

Cluster-to-cluster coverage (C2C) [14], [15], [16] is an external metric to assess component-level accuracy that measures the degree of overlap between the two architecture's clusters. Before calculating C2C between two recovered architecture, it is necessary to calculate the following equation:

$$c2c(c_i, c_j) = \frac{|\text{entities}(c_i) \cap \text{entities}(c_j)|}{\max(|\text{entities}(c_i)|, |\text{entities}(c_j)|)},$$

where c_i is an automatically prepared clustering; c_j is a ground-truth cluster; and $\text{entities}(c)$ is the set of entities in cluster c. C2C is computed as following:

$$C2C(A_1, A_2)(\%) = \frac{|\text{simC}(A_1, A_2)|}{|A_1.C|} \quad (5)$$

$$\text{simC}(A_1, A_2) = \{c_i | (c_i \in A_1, \exists c_j \in A_2) \wedge c2c(c_i, c_j) > th_{cvg}\}.$$

A_1 is the recovered architecture; A_2 is a ground-truth architecture; and $A_1.C$ are the clusters of A_1 . $\text{simC}(A_1, A_2)$

TABLE 8
Data Sets Specifications for Two Large and Very Large Software Systems

Project	Version	Description	SLOC	#File
ITK	4.5.2	Image Segmentation Toolkit	1M	7,310
Chromium	svn-171054	Web Browser	10M	18,698

returns A'_1 's clusters for which the $c2c$ value is above a threshold th_{cvg} .

To compare the overall results of FCA against other tested algorithms in terms of TurboMQ, MoJoFM, C2C, and running time, we utilized a non-parametric effect size statistic namely Cliff's δ which is used to quantify the amount of difference between two algorithms.

4.4 Algorithmic Parameters

The setting of parameters is necessary for search-based algorithms. For genetic-based algorithms, for our comparisons, we followed the algorithmic parameters setting used in [9], [20]. Algorithmic parameters are dependent on the number of artifacts (N). For the rest of the algorithms (i.e., Hill-climbing algorithm and Estimation of Distribution algorithm), we used the same parameters as those used by the authors of these algorithms. We obtained the implementation of the ACDC from its official web site.

As references [8], [9], [37], to reduce randomness the results of the search-based algorithms used in comparisons, we collect the best of 30 independent runs. For MoJoFM, TurboMQ, and C2C, we report the best values rounded to the closest integer. Let N denote the number of artifacts, the parameter setting for experiments is shown in Table 9.

4.5 Research Questions

The following questions are answered to evaluate the effectiveness of the proposed algorithm.

TABLE 9
The Parameter Setting for Experiments

Parameter	Value
Population size	10N
Maximum number of generations	200N
Crossover Rate	0.8
Mutation Rate	0.004log ₂ (N)
Termination condition	There has been no improvement in the population for 50 iterations

TABLE 10
Features of Selected Search-Based Algorithms for Comparison With the Proposed Algorithm

Algorithm	Algorithm type	#objective used	Search type	Structural based/Semantic-based	Encoding type
Bunch-GA [5]	Genetic algorithm	Single-objective	Global	Structural	real-valued
DAGC [22]	Genetic algorithm	Single-objective	Global	Structural	permutation-based
ECA [9]	two-Archive genetic algorithm	Multi-objective	Global	Structural	real-valued
MCA [9]	two-Archive genetic algorithm	Multi-objective	Global	Structural	real-valued
Bunch-SAHC [17]	Hill-climbing algorithm	Single-objective	Local	Structural	real-valued
SHC [21]	Hill-climbing algorithm	Single-objective	Local	Semantic	real-valued
GA-SMCP [20]	Genetic algorithm	Single-objective	Global	Structural	real-valued
EoD [8]	Estimation of Distribution algorithm	Multi-objective	Global	Semantic & Structural	real-valued

RQ1. Does the proposed algorithm perform better than the hierarchical and non-hierarchical clustering algorithms in terms of TurboMQ, MoJoFM, and C2C?

RQ2. Is the proposed algorithm scalable?

RQ3. Is there a statistically significant improvement between the FCA and the algorithms compared?

We ran the algorithms on a Laptop with Intel core i7 processor 2.60 GHz and 16 GB of memory.

5 EXPERIMENTAL RESULTS

This section presents the results of the empirical study. The aim is to compare the proposed algorithm, FCA, against some hierarchical and non-hierarchical algorithms in terms of TurboMQ, MoJoFM, C2C, and running time. To this end, eight search-based algorithms with different characteristics are chosen. The algorithms selected vary from each other in some different ways including single-objective, multi-objective, global search, local search, structured-based methods, and semantic-based methods. The software clustering approaches to which we compared FCA are Bunch-GA, DAGC, ECA, MCA, Bunch-NAHC, SHC, GA-SMCP, and EoD. The characteristics of these algorithms are described in Table 10. K-means, a basic machine learning algorithm, and ACDC, a clustering algorithm based on subsystem patterns are also used for comparison. Several previous studies [14], [15], [16] have shown that ACDC performed well on the tested applications. Besides, we selected agglomerative clustering algorithms such as Complete, Single, Average (Weighted), WCA-UE, WCA-UENM, and LIMBO for comparison.

To compare and evaluate the proposed algorithm, several software systems with different domains and sizes have been selected. Tables 6, 7, and 8 show the specifications of these software systems. Note that the dependency used in the software systems shown in Table 6 are call and include dependencies. The dependency used in the software systems shown in Table 7 are call and include dependencies, that we are obtained from their source code using Understand toolset (<https://scitools.com/>).

Lutellier *et al.* [15] have extracted various dependencies for ITK and Chromium applications such as include, symbol, Function call, etc. These dependencies alone do not cover the entire program. For example, in Chromium, the function call dependency only covers 12,627 artifacts of 18,698 artifacts. So, to cover the whole program, we merged these dependencies and removed duplicate dependencies.

The size of projects used for the experiments are 13 to 18,698 source files. Table 11 shows the size of projects used for experiments in some existing clustering algorithms.

To answer the research question RQ1, we compared the proposed algorithm against some hierarchical algorithms, k-means and ACDC in terms of TurboMQ on ten small-sized applications shown in Table 6. Table 12 shows the comparison results. The results demonstrate that in all cases the proposed algorithm has been able to obtain higher quality clustering than the algorithms tested.

We have also selected the Mozilla Firefox application. The reason for this choice is that there is an expert decomposition for it. Ten folders with different functionalities have been selected from this application. We have clustered these folders with eight evolutionary algorithms with different features, k-means and ACDC. Because the clustering problem is an NP-hard problem, evolutionary algorithms usually produce plausible solutions [8], [9]. In terms of MoJoFM, Table 13 shows that the proposed algorithm performs better in six out of ten cases. In the *Build* folder, there is a significant difference between the FCA and the other algorithms in the value of MoJoFM, and the FCA did not work well. The reason for this improper performance is that the dependency graph of this folder is disconnected and also has several isolated vertices.

TABLE 11
Projects Size Used for Experiments in Some Clustering Algorithms

Reference	Projects size (#modules or #files)
[9], [20]	20 to 198
[5], [17]	13 to 413
[31]	13 to 124
[8]	21 to 97
[32]	63 to 401
[24]	4 to 93
[21]	21 to 881
[20]	20 to 198
[40]	41 to 97
in this paper	13 to 18698

TABLE 12
Comparison of the Proposed Algorithm With Some Hierarchical Algorithms, k-Means and ACDC in Terms of TurboMQ

Software systems	WCA-UE	Average Linkage	Complete Linkage	Single Linkage	ACDC	k-means	FCA
Compiler	0.836	0.527	0.527	0.933	1	0.85	1.22
Boxer	1.343	0.964	0.983	0.964	2.82	0.79	3.020
Ispel	1.489	1.739	1.639	0.995	1.75	1.2	1.97
Bison	0.994	0.994	0.994	0.994	1	1	2.25
Cia	0.997	0.997	0.997	0.997	1.86	1.78	2.049
Ciald	0.984	0.487	1.093	0.984	1.70	0.78	1.72
Nos	0.969	0.990	0.990	0.990	1	0.98	1.08
Rcs	0.977	0.990	1.018	1.018	1	1.26	1.81
Spdb	0.933	0.933	0.933	0.933	5	1.15	5
Star	1.388	0.989	0.805	0.989	2.09	0.81	3.048

TABLE 13
Comparison of the Proposed Algorithm With Some State-of-the-Art Search-Based Algorithms in Terms of MoJoFM (M)(%) and C2C (C)(%)

Folder name	Bunch-GA		DAGC		ECA		MCA		Bunch-NAHC		SHC		GA-SMCP		EoD		ACDC		k-means		FCA	
	M	C	M	C	M	C	M	C	M	C	M	C	M	C	M	C	M	C	M	C	M	C
Accessible	42	40	27	0	37	45	39	49	28	20	27	0	38	28	42	45	40	31	35	10	42	40
Browser	70	55	45	12	60	49	72	55	50	38	52	19	70	58	48	58	65	33	57	60	87	60
Build	84	69	47	29	78	61	78	61	84	61	84	61	94	77	66	33	83	50	50	67	55	
Content	31	12	10	2	27	15	35	18	21	21	22	12	18	9	31	25	57	43	48	10	60	30
Db	94	60	50	43	96	72	96	72	94	71	94	65	91	69	91	81	96	72	95	50	95	50
Dom	58	33	25	0	52	21	53	23	40	19	38	20	58	31	58	42	82	67	47	33	83	67
Extensions	49	30	22	2	58	35	53	31	24	12	28	19	48	25	51	40	76	71	45	26	79	71
Gfx	54	38	29	11	60	49	67	53	42	28	42	22	54	21	60	31	71	36	65	20	64	26
Intl	79	69	40	15	74	65	83	68	75	68	75	61	71	65	75	62	91	79	79	59	81	73
Ipc	81	65	39	15	80	65	80	65	81	65	81	68	80	63	80	58	60	30	68	50	82	70

TABLE 14
Comparison of the Proposed Algorithm With Some State-of-the-Art Search-Based Algorithms, k-Means and ACDC in Terms of TurboMQ

Folder name	Bunch-GA	DAGC	ECA	MCA	Bunch-NAHC	SHC	GA-SMCP	EoD	ACDC	k-means	FCA
Accessible	6.26	0.93	22.17	28.98	4.80	10.01	14	29.21	12.26	0.82	17.92
Browser	3.72	0.92	4	28.5	5.85	9	6.5	9	11.45	0.98	7.57
Build	3	0.5	3	3	1	0.92	1.23	2.9	3	0.85	3
Content	6.76	0.19	46.09	39.40	5.41	16.97	12.01	10.11	28.05	2.69	36.66
Db	2.34	0.86	5.5	6.9	2.60	2.34	1.90	2.51	3.12	0.70	2.7
Dom	6.16	0.92	23.51	79.38	4.30	12.87	5.11	8	6.93	0.67	7.44
Extensions	11.80	0.91	24.92	32.85	6.66	8.90	10.99	13	7	1.8	26.62
Gfx	6.50	0.82	29.14	70.43	4.32	3.01	6.86	12.22	10.58	1.63	19.83
Intl	5.46	0.90	60.04	116.63	2.54	7.90	4.11	12.90	12.16	0.86	31.19
Ipc	5.64	0.84	17.7	18.29	3.92	4.50	5.64	10.90	19.68	1.63	11.41

TABLE 15
Comparison of the Proposed Algorithm With Some State-of-the-Art Search-Based Algorithms in Terms of Time (Second)

Folder name	Bunch-GA	DAGC	ECA	MCA	Bunch-NAHC	SHC	GA-SMCP	EoD	ACDC	k-means	FCA
Accessible	4535	7471	4521	4437	101.5	869	5921	3990	0.86	4.03	0.30
Browser	708	609.5	733.5	796.5	4.95	12.25	901	541	0.86	0.72	0.18
Build	512	383.805	421.5	425	3	3.2	540	431	0.26	0.05	0.03
Content	950337.5	4794247.5	943040	943021	698441.5	6531479	5224315	890021	3.31	2007.9	3.30
Db	494.5	1834.495	1363.5	1470.5	47.4	1350.5	2301	481	1.25	0.32	0.26
Dom	3110	6139	3082.5	3153	110.5	101.4	6341	2208	0.79	0.95	0.25
Extensions	6264	7653	6461	6730	266.5	4032	6421	6259	0.36	3.24	0.28
Gfx	15563	28173	14781	14966	1131	2036	21540	13238	0.70	19.05	0.51
Intl	222888	1333765.5	223645	222884	238100.5	419513.5	1034921	22198	1.24	40	0.82
Ipc	62770.5	424153.5	62642.5	62683.5	1196	899.5	99101	61211	0.15	0.48	0.11

Note that, the FCA works better in large folders than other algorithms. In terms of C2C ($th_{cvg} > 33\%$), the FCA can compete with other algorithms. In terms of TurboQ, Table 14 shows that the FCA has comparable results.

The important point is that the running time of the algorithm is much shorter than the algorithms compared. Table 15 shows the running times of the algorithms. In the folders where there are many artifacts, the difference in running time is considerable. In terms of time, the FCA has significant superiority over evolutionary algorithms. For example, the well-known MCA algorithm took about 260 hours to cluster the *content* folder, while the proposed algorithm took about 3 seconds. With the increase in the size of software systems, the performance of evolutionary algorithms slows down due to the size of their solutions, the time-consuming operators, and in some cases, memory problems. But the proposed solution only works on the matrix, which makes it less time consuming than the others.

To answer the research question RQ2, to further investigate the performance of the proposed algorithm, we have selected two large (ITK including 7,310 files) and very large software systems (Chromium including 18,698 files).

For future comparison, we selected four state-of-the-art algorithms, which all are published in IEEE Transactions on Software Engineering Journal. The algorithms selected are Bunch-SAHC, WCA-UE, WCA-UENM, and LIMBO. The Bunch-SAHC is a search-based algorithm and others are hierarchical algorithms. We also selected two famous algorithms k-means and ACDC. Due to time and memory problems, we were unable to select other algorithms from search-based methods. Table 16 shows the results in terms of TurboMQ, MoJoFM, C2C, and run time. The results show that the proposed algorithm performs better than hierarchical algorithms and can also compete with the Bunch-SAHC algorithm and ACDC. But the running time of the proposed algorithm is much shorter, which is discussed below.

For ITK and Chromium, the techniques compared take several hours to days to run. Considering the existing tested algorithms, running all experiments for Chromium would take more than a week of CPU time on a single machine or time out (TO), and the ACDC takes 10 hours for clustering. Bunch-SAHC, and LIMBO timed out after 24 and 8 days, respectively. For Bunch-SAHC, we report here the intermediate architecture recovered at that time. K-means can take

TABLE 16

Comparison of the Proposed Algorithm With Other Algorithms on ITK and Chromium in Terms of TurboMQ (T)(%), MoJoFM (M)(%), C2C (C)(%), Time (d: Day, h: Hour, s: Second)

Algorithm	ITK			Chromium			Time	
	T	M	C	T	M	C		
Bunch-SAHC	14	42	1	24 [†] d	14	53	10	24 [†] d
WCA-UE	1	33	0	16 h	1	21	0	31 h
WCA-UENM	2	31	0	18 h	1	23	0	38 h
LIMBO	10	28	0	8 d	TO	TO	TO	TO
ACDC	15	55	0	565 s	18	58	41	10 h
k-means	14	26	0	24 [†] h	6	33	7	36 [†] h
FCA	28	44	6	272 s	16	36	45	8 h

[†]Scores denote results for intermediate architectures recovered at that time.

varying numbers of clusters as input. For k-means, the algorithm has been executed with different values of k in steps of 5 increment, up to a specified runtime. It is important to note that it is not possible to terminate hierarchical methods in the clustering process and these algorithms must be executed until the last step. But it is possible to terminate search-based algorithms in the clustering process and report the intermediate results. The results obtained from this research question demonstrate that the proposed algorithm is scalable, and can cluster large applications in less time.

To answer the research question RQ3, Cliff's δ effect size metric is utilized. This test is a non-parametric effect size metric that quantifies the difference among two groups of observations (here FCA against other tested algorithms). The result of this metric is in range -1 to 1 and higher value shows that results of the first group (here, FCA) generally are better than the second group (other algorithms). To interpret, as [15], the following magnitudes are used: negligible ($|\delta| < 0.147$), small ($|\delta| < 0.33$), medium ($|\delta| < 0.474$), and large ($0.474 \geq |\delta|$). The results (Table 17) indicate that, in terms of time, the FCA is better than the other algorithms. Also, the values of MoJoFM, TurboMQ, and C2C in FCA are better than the other algorithms in general.

From the short review above, the main achievements, including contributions to the field can be summarized as follows:

- 1) Compared with hierarchical algorithms, the FCA results in a modularization of higher quality and is also comparable with search-based algorithms and ACDC, a state-of-the-art algorithm, in terms of the internal and external metrics.
- 2) Because the FCA has fewer and simpler operations than the other algorithms, it can cluster large graphs in less time and therefore it is scalable. Compared to the state-of-the-art algorithms, the proposed algorithm is the fastest software clustering algorithm.

6 THREATS TO VALIDITY

In this section, we discuss the threats that could affect the validity of the results obtained from the evaluation. Despite our efforts to avoid/reduce as many threats to validity as possible, some are inevitable. In the following, we address the threats to validity from two aspects of external and internal validity.

Threats to External Validity. Several factors may restrict the generality and limit the interpretation of our results.

TABLE 17
Cliff's δ Effect Size Test

Algorithm	TurboMQ	MoJoFM	C2C	Time
Bunch-GA	.59 (large)	.31 (small)	.24 (small)	1 (large)
ECA	-.19 (small)	.5 (large)	.23 (small)	1 (large)
MCA	-.48 (large)	.28 (small)	.15 (small)	1 (large)
Bunch-NAHC	.87 (large)	.44 (med.)	.39 (med.)	1 (large)
Bunch-SAHC	1 (large)	-.5 (large)	.5 (large)	1 (large)
SHC	.68 (large)	.44 (med.)	.48 (large)	1 (large)
GA-SMCP	.7 (large)	.34 (med.)	.31 (small)	1 (large)
EoD	.24 (small)	.36 (med.)	.05 (neg.)	1 (large)
ACDC	.19 (small)	.05 (neg.)	.1 (neg.)	.37 (med.)
k-means	.80 (large)	.26 (small)	.45 (large)	.48 (large)
WCA-UE	.91 (large)	1 (large)	1 (large)	1 (large)
WCA-UENM	1 (large)	1 (large)	1 (large)	1 (large)
LIMBO	1 (large)	1 (large)	1 (large)	1 (large)

A positive value indicates that the effect size favor of the FCA. The interpretation of the effect size is indicated in parenthesis. neg. stands for negligible and med. for medium.

The main external threat arises from the possibility that the selected application is not representative of software systems in general, with the result that the findings of the experiments do not apply to 'typical' software systems. To address these concerns, a variety of applications with different functionalities and sizes are considered. There is, therefore, reasonable cause for confidence in the results obtained and the conclusions drawn from them.

Threats to Internal Validity. The external metrics used to the evaluation can affect the validity of the results. As [10], [15], we utilized two well-known and widely used metrics, namely MoJoFM, C2C, for the evaluation. Different metrics, such as architecture-to-architecture [15], and EdgeSim [4], may produce different results for the same software system.

Another important factor that affects the experiment results is the accuracy of the authoritative decomposition achieved from a software system. We used the package structure (directory structure) of the Mozilla Firefox as an authoritative decomposition. The expert decompositions that we selected have been used earlier in software modularization experiments in [7], [8]. We know that there is a big threat as the directory structure of a project is often different from the actual "ground-truth decomposition." In well-structured projects, the directory structure of the project originally usually reflects the architecture of the project [39]. To handle this threat, we selected the developer preview version of the Mozilla Firefox, because there is a credible (human) expert decomposition (the directory structure) of that. It is worth mentioning that the selected version is a stable version, as small changes have been made to it and its directory structure has not changed.

Isolated vertices (single vertices). These nodes have no connection to other nodes. Thus, it is not possible to assign them into specific clusters. These single vertices are one of the reasons for the discrepancy between the results of the algorithms and the expert clustering.

7 CONCLUSION

Given the importance of clustering in understanding and maintaining software as well as its importance for extracting software architecture, in this paper, we proposed a new style of software system clustering based on the artifact dependency graph. To this end, we proposed a clustering

algorithm that works on the dependency matrix. Comparative results indicated that it performs better than hierarchical algorithms and competes with search-based algorithms in terms of TurboMQ (an internal metric) and two external metrics namely MoJoFM and C2C. The main feature of the FCA is its scalability. It can cluster very large software systems within a reasonable amount of time.

Future research should be devoted to the development of:

- 1) *Using the non-structural features.* The algorithm will be developed to consider some nonstructural features such as artifacts name and comments, along with structural features, in the process of software clustering.
- 2) *Preprocessing for determining libraries and utilities.* Some algorithms try to delete libraries and utilities before the clustering process.

REFERENCES

- [1] X. Xia, L. Bao, D. Lo, Z. Xing, A. E. Hassan, and S. Li, "Measuring program comprehension: A large-scale field study with professionals," *IEEE Trans. Softw. Eng.*, vol. 44, no. 10, pp. 951–976, Oct. 2018.
- [2] S. Ducasse and D. Pollet, "Software architecture reconstruction: A process-oriented taxonomy," *IEEE Trans. Softw. Eng.*, vol. 35, no. 4, pp. 573–591, Jul./Aug. 2009.
- [3] M. Shtern and V. Tzerpos, "Clustering methodologies for software engineering," *Advances Softw. Eng.*, vol. 2012, 2012, Art. no. 1.
- [4] A. Isazadeh, H. Izadkhah, and I. Elgedawy, *Source Code Modularization: Theory and Techniques*. Berlin, Germany: Springer, 2017.
- [5] B. S. Mitchell and S. Mancoridis, *A Heuristic Search Approach to Solving the Software Clustering Problem*. Philadelphia, PA, USA: Drexel Univ., 2002.
- [6] P. Andritsos and V. Tzerpos, "Information-theoretic software clustering," *IEEE Trans. Softw. Eng.*, vol. 31, no. 2, pp. 150–165, Feb. 2005.
- [7] S. Mohammadi and H. Izadkhah, "A new algorithm for software clustering considering the knowledge of dependency between artifacts in the source code," *Inf. Softw. Technol.*, vol. 105, pp. 252–256, 2019.
- [8] N. S. Jalali, H. Izadkhah, and S. Lotfi, "Multi-objective search-based software modularization: Structural and non-structural features," *Soft Comput.*, vol. 23, pp. 11141–11165, 2019.
- [9] K. Praditwong, M. Harman, and X. Yao, "Software module clustering as a multi-objective search problem," *IEEE Trans. Softw. Eng.*, vol. 37, no. 2, pp. 264–282, Mar./Apr. 2011.
- [10] R. Naseem, O. Maqbool, and S. Muhammad, "Cooperative clustering for software modularization," *J. Syst. Softw.*, vol. 86, no. 8, pp. 2045–2062, 2013.
- [11] O. Maqbool and H. Babri, "Hierarchical clustering for software architecture recovery," *IEEE Trans. Softw. Eng.*, vol. 33, no. 11, pp. 759–780, Nov. 2007.
- [12] A. Shokoufandeh, S. Mancoridis, and M. Maycock, "Applying spectral methods to software clustering," in *Proc. 9th Work. Conf. Reverse Eng.*, 2002, pp. 3–10.
- [13] V. Tzerpos and R. C. Holt, "ACDC: An algorithm for comprehension-driven clustering," in *Proc. 7th Work. Conf. Reverse Eng.*, 2000, pp. 258–267.
- [14] T. Lutellier et al., "Comparing software architecture recovery techniques using accurate dependencies," in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, 2015, pp. 69–78.
- [15] T. Lutellier et al., "Measuring the impact of code dependencies on software architecture recovery techniques," *IEEE Trans. Softw. Eng.*, vol. 44, no. 2, pp. 159–181, Feb. 2018.
- [16] J. Garcia, I. Ivkovic, and N. Medvidovic, "A comparative analysis of software architecture recovery techniques," in *Proc. 28th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2013, pp. 486–496.
- [17] B. S. Mitchell and S. Mancoridis, "On the automatic modularization of software systems using the bunch tool," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 193–208, Mar. 2006.
- [18] H. Izadkhah, I. Elgedawy, and A. Isazadeh, "E-CDGM: An evolutionary call-dependency graph modularization approach for software systems," *Cybern. Inf. Technol.*, vol. 16, no. 3, pp. 70–90, 2016.
- [19] M. C. Moncõres, A. C. Alvim, and M. O. Barros, "Large neighborhood search applied to the software module clustering problem," *Comput. Operations Res.*, vol. 91, pp. 92–111, 2018.
- [20] J. Huang and J. Liu, "A similarity-based modularization quality measure for software module clustering problems," *Inf. Sci.*, vol. 342, pp. 96–110, 2016.
- [21] M. Kargar, A. Isazadeh, and H. Izadkhah, "Semantic-based software clustering using hill climbing," in *Proc. Int. Symp. Comput. Sci. Softw. Eng. Conf.*, 2017, pp. 55–60.
- [22] S. Parsa and O. Bushehrian, "A new encoding scheme and a framework to investigate genetic clustering algorithms," *J. Res. Practice Inf. Technol.*, vol. 37, no. 1, 2005, Art. no. 127.
- [23] J. Huang, J. Liu, and X. Yao, "A multi-agent evolutionary algorithm for software module clustering problems," *Soft Comput.*, vol. 21, no. 12, pp. 3415–3428, 2017.
- [24] J. K. Chhabra et al., "Harmony search based remodularization for object-oriented software systems," *Comput. Lang. Syst. Struct.*, vol. 47, pp. 153–169, 2017.
- [25] A. C. Kumari and K. Srinivas, "Hyper-heuristic approach for multi-objective software module clustering," *J. Syst. Softw.*, vol. 117, pp. 384–401, 2016.
- [26] M. Tajgardan, H. Izadkhah, and S. Lotfi, "Software systems clustering using estimation of distribution approach," *J. Appl. Comput. Sci. Methods*, vol. 8, no. 2, pp. 99–113, 2016.
- [27] A. Prajapati and J. K. Chhabra, "An efficient scheme for candidate solutions of search-based multi-objective software remodularization," in *Proc. Int. Conf. Hum. Interface Manage. Inf.*, 2016, pp. 296–307.
- [28] J. Hwa, S. Yoo, Y.-S. Seo, and D.-H. Bae, "Search-based approaches for software module clustering based on multiple relationship factors," *Int. J. Softw. Eng. Knowl. Eng.*, vol. 27, no. 07, pp. 1033–1062, 2017.
- [29] A. Ramirez, J. R. Romero, and S. Ventura, "Interactive multi-objective evolutionary optimization of software architectures," *Inf. Sci.*, vol. 463, pp. 92–109, 2018.
- [30] M. Akbari and H. Izadkhah, "Hybrid of genetic algorithm and krill herd for software clustering problem," in *Proc. 5th Conf. Knowl. Based Eng. Innov.*, 2019, pp. 565–570.
- [31] H. Izadkhah and M. Tajgardan, "Information theoretic objective function for genetic software clustering," in *Proc. 5th Int. Electron. Conf. Entropy Appl.*, 2019, pp. 1–9.
- [32] J. K. Chhabra et al., "Many-objective artificial bee colony algorithm for large-scale software module clustering problem," *Soft Comput.*, vol. 22, no. 19, pp. 6341–6361, 2018.
- [33] J. Garcia, D. Popescu, C. Mattmann, N. Medvidovic, and Y. Cai, "Enhancing architectural recovery using concerns," in *Proc. 26th IEEE/ACM Int. Conf. Automated Softw. Eng.*, 2011, pp. 552–555.
- [34] A. Corazza, S. Di Martino, V. Maggio, and G. Scanniello, "Investigating the use of lexical information for software system clustering," in *Proc. 15th Eur. Conf. Softw. Maintenance Reeng.*, 2011, pp. 35–44.
- [35] J. Misra, K. M. Annervaz, V. Kaulgud, S. Sengupta, and G. Titus, "Software clustering: Unifying syntactic and semantic features," in *Proc. 19th Work. Conf. Reverse Eng.*, 2012, pp. 113–122.
- [36] C. Xiao and V. Tzerpos, "Software clustering based on dynamic dependencies," in *Proc. 9th Eur. Conf. Softw. Maintenance Reeng.*, 2005, pp. 124–133.
- [37] M. Kargar, A. Isazadeh, and H. Izadkhah, "Multi-programming language software systems modularization," *Comput. Elect. Eng.*, vol. 80, 2019, Art. no. 106500.
- [38] J. Garcia, I. Krka, C. Mattmann, and N. Medvidovic, "Obtaining ground-truth software architectures," in *Proc. Int. Conf. Softw. Eng.*, 2013, pp. 901–910.
- [39] J. Wu, A. E. Hassan, and R. C. Holt, "Comparison of clustering algorithms in the context of software evolution," in *Proc. 21st IEEE Int. Conf. Softw. Maintenance*, 2005, pp. 525–535.
- [40] I. Hussain et al., "A novel approach for software architecture recovery using particle swarm optimization," *Int. Arab J. Inf. Technol.*, vol. 12, no. 1, pp. 32–41, 2015.
- [41] Z. Wen and V. Tzerpos, "An effectiveness measure for software clustering algorithms," in *Proc. 12th IEEE Int. Workshop Program Comprehension*, 2004, pp. 194–203.



Navid Teymourian received the bachelor's degree in computer engineering-software from Hamedan Payame Noor University, Iran and the master's degree in computer science-systems theory from University of Tabriz, Iran, in 2013 and 2020, respectively. He also received the master's degree in computer science. His research interests include clustering algorithms, software engineering, cryptography, parallel computing, and algorithms. Recently he is working on encryption algorithms.



Habib Izadkhah is an assistant professor at the Department of Computer Science, University of Tabriz, Iran. More recently he has been working on the application of reverse engineering and modularization to the extract of software structure for large-scale and multi-programming languages source code. He contributed to various research projects, coauthored a number of research papers in international conferences, workshops and journals. He has authored a book entitled "*Source Code Modularization: Theory and Techniques*," published by Springer. His research interests include graph theory, optimization algorithms, software engineering, software architecture, reverse engineering, and organic software.



Ayaz Isazadeh received the master's degree in electrical engineering and computer science from Princeton University, in 1978, and Ph.D. degree in computing and information science from Queen's University, Canada, in 1996. He is a professor and the founding member of the Department of Computer Science, University of Tabriz, Iran. He has served as the chair and honorary chair of numerous national and international conferences. His research interests include software engineering, software architecture, and recently, software struc-

tural evolution, with more than 100 papers published in international journals and conference proceedings. Before returning to graduate school in 1992, he worked with AT&T Bell Laboratories for over eight years on various software systems development.

▷ For more information on this or any other computing topic, please visit our Digital Library at www.computer.org/csl.