STAMP

Deliverable D3.4

Consolidated services for online-test amplification

Project Number	:	731529
Project Title	:	STAMP
Deliverable Type	:	Report

Deliverable Number	:	D3.4
Title of Deliverable	:	Consolidated services for online-test amplification
Dissemination Level	:	Public
Version	:	1.01
Latest version	:	https://github.com/STAMP-project/docs-forum/blob/master/docs/d34_consolidated_services_for_online-test_amplification.pdf
Contractual Delivery Date	:	M34 - September 30, 2019
Contributing WPs	:	WP 7
Editor(s)	:	Xavier Devroey, TU Delft
Author(s)	:	Pouria Derakhshanfar, TU Delft Xavier Devroey, TU Delft Arie van Deursen, TU Delft Andy Zaidman, TU Delft
Reviewer(s)	:	Benoit Baudry, KTH Caroline Landry, INRIA Yosu Gorroñogoitia, Atos

Abstract

Search-based approaches have been applied to a variety of white-box testing activities, among which are test case and data generation and crash reproduction. This deliverable builds upon our previous work by leveraging contextual information using model-seeding *to generate input data during the search-process; and extending the code instrumentation used by Botsing to define a new test generation strategy for search-based test case generation and enhance the guidance for search-based crash reproduction. First, we present our new seeding strategy, called* behavioral model seeding*, which abstracts behavior observed in the source code and test cases using transition systems. The transition systems represent the (observed) usages of the classes and are used during the search to generate objects and sequences of method calls on those objects. Our evaluation shows that behavioral model-seeding outperforms both test seeding and no-seeding by a minimum of 6% without any notable negative impact on efficiency. Furthermore, we present how model-seeding can be used for test case generation. We outline the protocol for an empirical evaluation to compare model seeding with test seeding and no-seeding unit test generation. Second, we present our extension of the current code instrumentation mechanism used by Botsing to take several classes and the way they interact into account. This extension allows to generate class integration tests, able to kill on average 10% of mutants that are not killed by unit-level generated tests, using search-based software testing and to refine the guidance of the fitness function for search-based crash reproduction. Finally, this deliverable provides an update on the development status of Botsing and its user documentation.*

Revision History

Version	Type of Change	Author(s)
1.00	initial version	Xavier Devroey, TU Delft Pouria Derakhshanfar, TU Delft Arie van Deursen, TU Delft Andy Zaidman, TU Delft

Contents

Glossary

This glossary presents the terminology used across the different deliverables of work package 3.

Botsing: Meaning *crash* in Dutch, Botsing is a complete re-implementation and extension of the crash replication tool EvoCrash. Whereas EvoCrash was a full clone of EvoSuite (making it hard to update EvoCrash as EvoSuite evolves), Botsing relies on EvoSuite as a (maven) dependency only and provides a framework for various tasks for crash reproduction and, more generally, test case generation. Furthermore, it comes with an extensive test suite, making it easier to extend. The license adopted is Apache, in order to facilitate adoption in industry and academia. Botsing is the name of the framework for online test amplification developed in WP3.

Code instrumentation: Code instrumentation in Botsing consists in the injection of probes into the bytecode of a Java application to monitor and log the runtime behavior of specific classes.

JCrashPack: JCrashPack is a benchmark containing 200 crashes to assess crash replication tools capabilities.

Model seeding: in search-based software testing, seeding consist in providing external information to the search algorithm to help the exploration of the search space. Model seeding, developed within STAMP, is the seeding of transition systems (a formalism similar to state machines describing a dynamic behavior) to a search based test case generation algorithm. The models represent the common usages of classes and allow the generate objects with sequences of method calls, representing a common behavior in the application. In Botsing, models are learned from the source code (static analysis) and from the logs produced by the execution of the system (dynamic analysis using instrumentation) using n-gram inference.

Stack trace: a (crash) stack trace is a piece of log data usually denoting a crash failure. A stack trace provides information on the exception thrown and on the propagation of that exception trough the stack of method calls.

Stack trace preprocessing: stack traces can contain redundant and useless information preventing to automatically reproduce the associated crash. The preprocessing allows to filter stack traces to keep only relevant information.

Introduction

Search-based approaches have been applied to a variety of white-box testing activities [47], among which are test case and data generation [69] and crash reproduction [95]. In D3.2 we identified several future research directions to improve search-based crash reproduction that can also be applied to classical search-based unit test generation [39]. Among which (1) the usage of contextual data and seeding to address the input data generation problem, and (2) the improvement of the guidance of the fitness function. We initially addressed (1) in D3.3 by presenting a novel seeding approach: model seeding. This deliverable extends D3.3 by (i) evaluating model seeding for crash reproduction and (ii) applying it to unit test generation. This deliverable also presents our work to address (2) by extending the current code instrumentation mechanism (*i.e.,* the dynamic injection of log probes in the Java bytecode to log and monitor the runtime behavior of specific classes) used by Botsing, allowing to (iii) define and evaluate a new test case generation strategy and (iv) enhance the guidance for search-based crash reproduction strategy.

First, we present our new seeding strategy, called *behavioral model seeding*, which abstracts behavior observed in the source code and test cases using transition systems. The transition systems represent the (observed) usages of the classes and are used during the search to generate objects and sequences of method calls on those objects. Furthermore, we present how model-seeding can be used for test case generation. We outline the protocol for an empirical evaluation to compare model seeding with test seeding and no-seeding unit test generation. Results show that model seeding outperforms existing seeding for search-based crash reproduction by a minimum of 6% without any notable negative impact on efficiency. More generally, model seeding improves Botsing by using objects in the same way as in the existing tests and source code.

Second, we present our extension of the current code instrumentation mechanism used by Botsing to take several classes and the way they interact into account. This extension allows to generate class integration tests using search-based software testing. Our evaluation shows that those tests are able to kill on average 10% of mutants that are not killed by unit-level generated tests. Regarding search-based crash reproduction, the extension of the instrumentation mechanism allows to refine the guidance of the fitness function for search-based crash reproduction, leading to more crash reproduction and from a higher frame level.

Finally, this deliverable provides an update on the development status of Botsing and its user documentation.

The remainder of this document is structured as follows:

Chapter 1 - Test amplification for common behaviors presents the update on model seeding and its application for unit test generation.

Chapter 2 - Code instrumentation extension presents the extension of code instrumentation for class integration testing and search-based crash reproduction.

Chapter 3 - The Botsing framework presents the development status, and the user and developer documentation of Botsing.

Summary of Artifacts

1. **Botsing framework** (see Chapter 3)
   - Link: https://github.com/STAMP-project/botsing/releases/tag/1.0.7
   - Contact: Pouria Derakhshanfar (TUD), Xavier Devroey (TUD), Luca Andreatta (Eng), Pasquale Vitale (Eng)
2. **Botsing parallel execution** (see Chapter 3)
   - Link: https://github.com/STAMP-project/botsing-parallel
   - Contact: Pasquale Vitale (Eng)
3. **EvoSuite specific implementation** (see Section 3.2.3)
   - Link: https://github.com/STAMP-project/evosuite
   - Contact: Pouria Derakhshanfar (TUD)
4. **Botsing crash reproduction tutorial** (see WP4 T4.4)
   - Link: https://github.com/STAMP-project/botsing-demo
   - Contact: Pouria Derakhshanfar (TUD), Xavier Devroey (TUD)
5. **EvoSuite model seeding tutorial** (see WP4 T4.4)
   - Link: https://github.com/STAMP-project/evosuite-model-seeding-tutorial
   - Contact: Pouria Derakhshanfar (TUD), Xavier Devroey (TUD)

Chapter 1

Test amplification for common behaviors

This chapter presents how common behaviors observed during the execution of the program are abstracted using learning algorithms into a (state-machine like) behavioral model, latter used as a seed for (i) search-based crash reproduction (in Section 1.1), and (ii) search-based unit test generation (in Section 1.2). Section 1.1 extends the content of the previous deliverable D3.3 by evaluating model seeding on a larger number of crashes.

1.1 Extraction and seeding of common software behaviors for crash reproduction

This deliverable extends the work done in D3.3: we performed an evaluation on 124 crashes from 5 open-source applications to answer the following research questions:

RQ1 What is the influence of *test seeding used during initialization* on search-based crash reproduction?

RQ2 What is the influence of *behavioral model seeding used during initialization* on search-based crash reproduction?

We consider both research questions from the perspective of *effectiveness* (of initializing the population and reproducing crashes) and *efficiency*. We also investigate the factors (*e.g.*, the cost of analyzing existing tests) that influence the test and model seeding approaches and gain a better insight in how the search-based crash reproduction process works and how it can be improved. Generally, our results indicate that behavioral model seeding enables the search process to start for three additional crashes. In addition, this seeding strategy increases the number of reproduced crashes by 4 (out of 122). Model-seeding achieves all of these improvements without any impact on efficiency. In contrast, using test seeding in crash-reproduction leads to a lower crash-reproduction rate and search initialization.

1.1.1 Background and related work

Application crashes that happen during operations of the system are usually reported to developer teams through an issue tracking system for debugging [114]. Depending on the amount of information reported from the operations environment, this debugging process may take more or less time. Typically, the first step for the developer is to try to reproduce the crash in his development environment [116]. Various approaches [19, 28, 74, 98, 115] automate this process and generate a *crash-reproducing test case* without requiring human intervention during the generation process. Previous studies [28, 100] show that such kind of test cases are helpful for the developers to debug the application.

Listing 1.1: Stack trace of the XWIKI-13372 crash

```
java.lang.NullPointerException: null
  at com[...]BaseProperty.equals([...]:96)
  at com[...]BaseStringProperty.equals([...]:57)
  at com[...]BaseCollection.equals([...]:614)
  at com[...]BaseObject.equals([...]:235)
  at com[...]XWikiDocument.equalsData([...]:4195)
  [...]
```

For Java programs, the information reported from the operations environment ideally includes a *stack trace*. For instance, Listing 1.1 presents a stack trace coming from the crash XWIKI-13372.[1] The stack trace indicates the *exception* thrown (`NullPointerException` here) and the *frames*, *i.e.,* the stack of method calls at the time of the crash, indexed from 1 (at line 1) to 26 (not shown here).

Various approaches use a stack trace as input to automatically generate a test case reproducing the crash. CONCRASH [19] focuses on reproducing *concurrency* failures that violate thread-safety of a class by iteratively generating test code and looking for a thread interleaving that triggers a concurrency crash. JCHARMING [73, 74] applies model checking and program slicing to generate crash reproducing tests. MUCRASH [115] exploits existing test cases written by developers. MUCRASH selects test cases covering classes involved in the stack trace and mutates them to reproduce the crash. STAR [28] applies optimized backward symbolic execution to identify preconditions of a target crash and uses this information to generate a crash reproducing test that satisfies the computed preconditions. Finally, RECORE [88] applies a search-based approach to reproduce a crash using both a stack trace and a core dump, produced by the system when the crash happened, to guide the search.

1.1.2 Search-based crash reproduction

Search-based approaches have been widely used to solve complex, non-linear software engineering problems, that may have multiple optimization objectives which may be in conflict or competing [47]. Recently, Soltani *et al.* [98] proposed a search-based approach for crash reproduction called EvoCrash. EvoCrash is based on the EVOSUITE approach [38, 39] and applies a new *guided genetic algorithm* to generate a test case that reproduces a given crash using distance, similar to the one described by Rossler *et al.* [88], to guide the search. For a given stack trace, the user specifies a *target frame* relevant to his debugging activities: *i.e.,* the line with a class belonging to his system, from which the stack trace will be reproduced. For instance, applying EvoCrash to the stack trace from Listing 1.1 with a target frame 2 will produce a crash-reproducing test case for the class `BaseStringProperty` that produces a stack trace with the same two first frames.

An overview of the approach is shown at the right part of Figure 2.1 (box 5). The first step of this algorithm, called *guided initialization*, is to generate a random population. This random population is a set of random unit tests where a *target method* call (*i.e.,* the method in the target frame) is injected in each test. During the search, classical guided crossover and guided mutation are applied to the tests in such a way that they ensure that only the tests with a call to the target method are kept in the evolutionary loop. The overall process is guided by a *weighted sum fitness function* [96], applied to each test t:

$$fitness(t) = 3 \times d_l(t) + 2 \times d_e(t) + d_s(t) \quad (1.1)$$

The terms correspond to the following conditions when executing the test: (i) whether the execution distance from the target line (d_l) is equal to 0.0, in which case, (ii) if the target exception type is thrown (d_e), in which case, (iii) if all frames, from the beginning up until the selected frame, are

[1] Described in issue https://jira.xwiki.org/browse/XWIKI-13372.

included in the generated trace (d_s). The overall fitness value for a given test case ranges from 0.0 (crash is fully reproduced) to 6.0 (no test was generated), depending on the conditions it satisfies.

Seeding strategies for search-based testing

Seeding strategies use related knowledge to help the generation process and optimize the fitness of the population [37, 29, 63]. We focus here on the usage of the source code and the available tests as primary sources of information for search-based testing. Other approaches, for instance, search for string inputs on the internet [70], or use the existing test corpus [104] to mine relevant formatted string values (*e.g.*, XML or SQL statements).

Seeding from the source code Three main seeding strategies are using the source code for search-based testing [37, 87, 10]: (i) *constant seeding* uses static analysis to collect and reuse constant values appearing in the source code (*e.g.*, constant values appearing in boundary conditions); (ii) *dynamic seeding* complements constant seeding by using dynamic analysis to collect numerical and string values, observed only during the execution of the software, and reuse them for seeding; and (iii) *type seeding* is used to determine the object type that should be used as an input argument, based on a static analysis of the source code (*e.g.*, by looking at `instanceof` conditions or generic types for instance).

Seeding from the existing tests Rojas *et al.* [87] suggest two test seeding strategies, using *dynamic analysis* on existing test cases: *cloning* and *carving*. Dynamic analysis uses code instrumentation to trace the different method called during an execution, which, compared to static analysis, makes it easier to identify inter-procedural sequences of method calls (for instance, in the context of a class hierarchy). Cloning and carving have been implemented in EvoSuite and can be used for unit test generation.

For cloning, the execution of an existing test case is copied and used as a member of the initial population of a search process. More specifically, after its instrumentation and execution, the test case is reconstructed internally (without the assertions), based on the execution trace of the instrumented test. This internal representation is then used as-is in the initial population. Internal representation of the cloned test cases are stored in a *test pool*.

For carving, an object is reused during the initialization of the population and mutation of the individuals. In this case, only a subset of an execution trace, containing the creation of a new object and a sequence of methods called on that object, is used to internally build an object on which the methods are called. This object and the subsequent method calls are then inserted as part of a newly created test case (initialization) or in an existing test when a new object is required (mutation). Internal representation of the carved objects[2] are stored in an *object pool*.

The integration of seeding strategies into crash reproduction is illustrated in Figure 2.1, box 5. As shown, the test cases (resp. objects) to be used by the algorithm are stored in a test case (resp. object) pool, from which they can be used according to user-defined probabilities. For instance, if a test case only contains the creation of a new `LinkedList` (using `new`) that is filled using two `add` method calls, the sequence, corresponding to the execution trace <`new`, `add`, `add`>, may be used as-is in the initial population (cloning) or inserted by a mutation into other test cases (carving).

Challenges in seeding strategies The existing seeding techniques use only one resource to collect information for seeding. However, it is possible that the preferred resource does not have enough information about the usages of the interesting classes. For instance, test seeding only uses the carved call sequences from the execution of the existing test cases. If the existing test cases do not cover the behavior of the crash in the interesting classes, this seeding strategy may even misguide the search process. As another challenge, If the number of observed call sequences is large, the seeding strategy

[2] In this deliverable, we use the term *object* to refer to a carved object, *i.e.,* an object plus the sequence of methods called on that object.

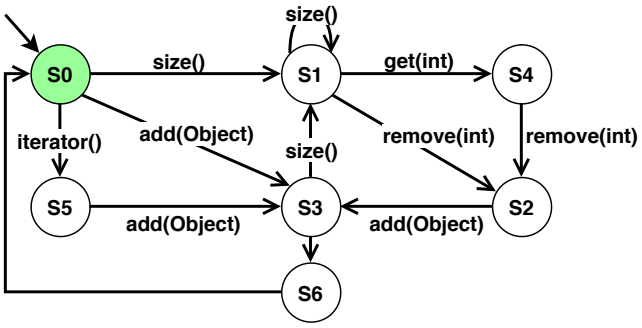


Figure 1.1: Transition system for method call sequences of the class `java.util.LinkedList` derived from Apache commons math source code and test cases.

needs a procedure to prioritize the call sequences for seeding. Seeding the random call sequences can misguide the search process. The existing seeding strategies do not have any procedure to fulfill this requirement.

Behavioral model-based testing

Model-based testing [108] relies on abstract specifications (models) of the system under test to support the generation of relevant test cases. *Transition systems* [16] have been used as a fundamental formalism to reason about test case generation and support the definition of formal test selection criteria [107]. Each abstract test case corresponds to a sequence of method calls on one object: *i.e.,* a path in the transition system starting from the initial state and ending in the initial state, a commonly used convention to deal with finite behaviours [33]. Once selected from the model, abstract test cases are concretized (by mapping the transition system's paths to concrete sequences of method calls) into *executable test cases* to be run on the system. In this deliverable, we derive abstract test cases (called *abstract object behavior* hereafter) and concretize them, producing pieces of code creating objects and calling methods on them. Those pieces of code are used as seeds for search-based crash reproduction.

Figure 1.1 shows an example of a transition system representing the possible *sequences* of method calls on `java.util.List` objects. Figure 1.1 illustrates usages of methods in `java.util.List` objects, learned from the code and tests, in terms of a transition system, from which *sequences* of methods calls can be derived. Derived sequences may correspond to the original sequences, but also combinations of those sequences. Hence the behavior described by the model is richer and subsumes the behavior of the sequences used to learn it. This property is very useful for seeding in search-based software testing as it allows better diversity of the objects, which helps during the search process.

A transition system is composed of a set of states with an initial state (s_0 in Figure 1.1) and transitions. Each transition may be labeled with an action, representing here a method call.

Abstract object behavior selection The abstract object behaviors are selected from the transition system following criteria defined by the tester. In the remainder of this deliverable, we use *dissimilarity* as selection criteria [26, 50]. Dissimilarity selection, which aims at maximizing the fault detection rate by increasing diversity among test cases, has been shown to be an interesting and scalable alternative to other classical selection criteria [50, 72]. This diversity is measured using a dissimilarity distance (here, 1 - the Jaccard index [53]) between the actions of two abstract object behaviors.

Model definition The model may be manually specified (and in this case will generally focus on specific aspects of the system) [108], or automatically learned from observations of the system [51, 62, 102, 101, 106, 111]. In the latter case, the model will be incomplete and only contain the *observed behavior* of the system [105]. For instance, the sequence `<new, addAll >` is valid for

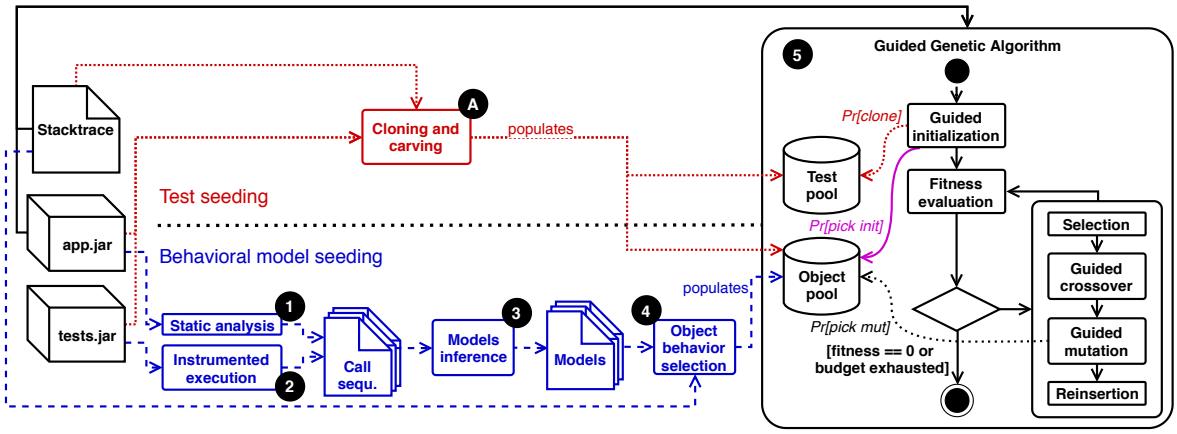


Figure 1.2: General overview of model seeding and test seeding for search-based crash reproduction

a `java.util.List` object but cannot be derived from the transition system in Figure 1.1 as the `addAll` method call has never been observed. The observed behavior can be obtained via static analysis [41] or dynamically [61]. Model inference may be used for visualization [62, 111], system properties verification [64, 43], or generation [51, 102, 101, 85, 117, 41] and prioritization [33, 35] of test cases.

We use n-gram inference to build the transition systems used for model seeding. N-gram inference takes a set of sequences of actions as input to produce a transition system where the n^{th} action depends on the $n-1$ previously executed actions. As a small value of n enables better diversity in the behavior allowed by the model (ending up in more diverse abstract object behaviors), requires less observations to reach stability of the model, simplifies the inference, and results in a more compact model [102, 101], we use 2-gram inference to build our models. For instance, the transition system model in Figure 1.1 has been partially generated from the following call sequences collected from the code and the test cases: `<size(), remove(), add(Object), size(), size(), size(), get(), remove(), add(Object)>`; `<iterator(),add(Object)>`; *etc.*. Each transition in Figure 1.1 corresponds to a method call in one of the sequences, and for each sequence, there exists a path in the model.

1.1.3 Behavioral Model and Test Seeding for Crash Reproduction

The goal of behavioral model seeding (denoted model seeding hereafter) is to abstract the behavior of the software under test using models and use that abstraction during the search. At the unit test level (which is the considered test generation level in this study), each model is a transition system, like in Figure 1.1, and represents possible usages of a class: *i.e.,* possible sequences of method calls observed for objects of that class.

The main steps of our model seeding approach, presented in Figure 1.2, are: the *inference* of the individual models ③ (described in Section 1.1.3) from the *call sequences* collected through *static analysis* ① performed on the application code (described in Section 1.1.3), and *dynamic analysis* ② of the test cases (described in Section 1.1.3); and for each model, the *selection of abstract object behaviors* ④, that are concretized into Java objects (described in Section 1.1.3), stored in an *object pool* from which the guided genetic algorithm ⑤ (described in Section 1.1.3) can randomly pick objects to build test cases during the search process.

Model inference

Call sequences are obtained by using static analysis on the bytecode of the application ① and by instrumenting and executing the existing test cases ②. For each class, the model ③ is obtained using

a 2-gram inference method using the call sequences of that class. For 2-gram inference, the next action to execute only depends on the current state in the transition system. Increasing the value of n for the n-gram inference would result in wider transition systems with more states and less incoming transitions, representing a more constrained behavior and producing less diverse test cases. For instance, in the transition system of Figure 1.1, the action `size()`, executed from state s_3 at step k only depends on the fact that the action `add(Object)` has been executed at step $k-1$, independently of the fact that there is a step $k-2$ during which the action `iterator()` has been executed.

Calls to constructors are considered as method calls during model inference. However, constructors may not appear in any transition of the model if no constructor call was observed during the collection of the call sequences. This is usually the case when the call sequences used to infer the model have been captured from objects that are parameters or attributes of a class. If an abstract object behavior does not start by a call to a constructor, a constructor is randomly chosen to initialize the object during the concretization.

For one version of the software under test, the model inference is a one time task. Models can then be directly reused for various crash reproductions.

Static analysis of the application The static analysis is performed on the bytecode of the application. We apply this analysis to all of the available classes in the software under test. In each method of these classes, we build the control flow graph, and for each object of that method, we collect the sequences of method calls on that object. For each object, each path in the control flow graph will correspond to one sequence of method calls. For instance, if the code contains a `if-then-else` statement, the `true` and `false` branches will produce two call sequences. In the case of a loop statement, the `true` branch is considered only once. The static analysis is *intraprocedural*, meaning that only the calls in the current method are considered. If an object is passed as a parameter of a call to a method that (internally) calls other methods on that object, those internal calls will not appear in the call sequences. This analysis ensures collecting all of the existing relevant call sequences for any internal or external class, which is used in the project.

Dynamic analysis for the test cases Since the existing manually developed test cases exemplify potential usage scenarios of the software under test, we apply dynamic analysis to collect all of the transpired sequences during the execution of these scenarios. In contrast to static analysis, which would require extensive effort and would produce imprecise call sequences, dynamic analysis is *interprocedural*. Meaning that the sequences include calls appearing in the test cases, but also internal calls triggered by the execution of the test case (*e.g.*, if the object is passed as a parameter to a method and methods are internally called on that object). Hence, through dynamic analysis, we gain a more accurate insight into the class usages in these scenarios.

Dynamic analysis of the existing tests is done in a similar way to the carving approach of Rojas *et al.* [87]: instrumentation adds log messages to indicate when a method is called, and the sequences of method calls are collected after execution. In similar fashion to static analysis, we collect call sequences of any observed object (even objects which are not defined in the software under test). The representativeness of the collected sequences depends on the coverage of the existing tests.

Abstract object behaviors selection

Abstract object behaviors are selected from the transition systems and concretized to populate the object pool used during the search. To limit the number of objects in the pool, we only select abstract object behaviors from two categories of models: models of internal classes (*i.e.,* classes belonging to packages of the software under test) and models of dependency classes (*i.e.,* classes belonging to packages of external dependencies) that are involved in the stack trace. Since we do not seek to validate the implementation of the application, the states are ignored during the selection process.

Listing 1.2: Concretized abstract object behavior for `LinkedList`

```
int[] t = new int[7];
t[3] = (-2147483647);
EuclideanIntegerPoint ep = new EuclideanIntegerPoint(t);
LinkedList<[...]> lst = new LinkedList<>();
lst.add(ep);
lst.add(ep);
```

Selection There exist various criteria to select abstract object behaviors from transition systems [108]. To successfully guide the search, we need to establish a good ratio between *exploration* (the ability to visit new regions of the search space) and *exploitation* (the ability to visit the neighborhood of previously visited regions) [31]. The guided genetic operators which are introduced in the EvoCrash approach [98] guarantee the exploitation by focusing the search based on the methods in the stack trace. However, depending on the stack trace, focusing on particular methods may reduce the exploration. Poor exploration decreases the diversity of the generated tests and may trap the search process in local optima.

To improve the exploration ability in the search process, we use *dissimilarity* as the criterion to select the abstract object behaviors. Compared to classical structural coverage criteria that seek to cover as many parts of the transition system as possible, dissimilarity tries to increase diversity among the test cases by maximizing a distance d (*i.e.,* the Jaccard index [53]):

$$d = 1 - \frac{t_1 \cap t_2}{t_1 \cup t_2}$$

Where $t_1 =< call_{11}, call_{12}, \ldots >$ and $t_2 =< call_{21}, call_{22}, \ldots >$ are two abstract object behaviors.

Concretization Each abstract object behavior has to be concretized to an object and method calls before being added to the objects pool. In other words, for each abstract object behavior, if the constructor invocation is not the first action, one constructor is randomly called; and the methods are called on this object in the order specified by the abstract object behavior with randomly generated parameter values. Due to the randomness, each concretization may be different from the previous one. For each abstract object behavior, n concretizations (default value is $n = 1$ to balance scalability and diversity of the objects in the object pool) are done for each abstract object behavior and saved in the object pool. For instance, Listing 1.2 shows the concretized abstract object behavior `<add(Object), add(Object)>` derived from the transition system model of Figure 1.1. The type of the parameters (`EuclideanIntegerPoint`) is randomly selected during the concretization and created with required parameter values (an integer array here).

Guided Initialization and Guided Mutation

Classes are instantiated to create objects during two main steps of the guided genetic algorithm: guided initialization, where objects are needed to create the initial set of test cases; and guided mutation, where objects may be required as parameters when adding a method call. When no seeding is used, those objects are randomly created (as in the concretization step described in Section 1.1.3) by calling the constructor and random methods.

Finally, to preserve exploration in model seeding, objects are picked from the object pool during guided initialization (resp. guided mutation) according to a user-defined probability $Pr[pick\ init]$ (resp. $Pr[pick\ mut]$), and randomly generated otherwise. In our evaluation, we considered four different values for $Pr[pick\ init] \in \{0.2, 0.5, 0.8, 1.0\}$, to study the effect of model seeding on the

Listing 1.3: Test generated for frame 2 of MATH-79b

```
public void testCluster() throws Exception{
 int[] t = new int[7];
 t[3] = (-2147483647);
 EuclideanIntegerPoint ep = new EuclideanIntegerPoint(t);
 LinkedList<[...]> lst = new LinkedList<>();
 lst.add(ep);
 lst.add(ep);
 KMeansPlusPlusClusterer<[...]> kmean = new KMeansPlusPlusClusterer<>(12);
 lst.offerFirst(ep);
 kmean.cluster(lst, 1, (-1357));
}
```

Listing 1.4: Stack trace excerpt for MATH-79b

```
1 java.lang.NullPointerException
   at ...KMeansPlusPlusClusterer.assignPointsToClusters()
3  at ...KMeansPlusPlusClusterer.cluster()
```

initialization of the search process. We fixed the value of $Pr[pick\ mut] = 0.3$, corresponding to the default value of $Pr[pick\ mut]$ for test seeding for classical unit test generation in EvoSuite.

As an example of object picking in action, test case generation with model seeding generated the test case in Listing 1.3 for the second frame of the stack trace from the crash MATH-79b from the Apache commons math project, reported in Listing 1.4. The target method is the last method called in the test (line 10) and throws a `NullPointerException`, reproducing the input stack trace. The first parameter of the method has to be a `Collection<T>` object. In this case, the guided genetic algorithm picked the list object from the object pool (from Listing 1.2) and inserted it in the test case (lines 2 to 7). The algorithm also modified that object (during guided mutation) by invoking an additional method on the object (line 9).

Test seeding

As described in Section 1.1.2, test seeding starts by executing the test cases (Figure 2.1 box Ⓐ) for carving, cloning and populating the test and object pools. Like for model seeding, only internal classes and external classes appearing in the stack trace are considered.

For crash reproduction, the test pool is used only during guided initialization to clone test cases that contain the target class, according to a user-defined $Pr[clone]$ probability. If the target method is not called in the cloned test case, the guided initialization also mutates the test case to add a call to the target method. The object pool is used during the guided initialization and guided mutation to pick objects. Differently from Rojas *et al.* [87], we make a distinction between picking objects during guided initialization and guided mutation. We use the the same user-defined probabilities as for model seeding, $Pr[pick\ init]$ and $Pr[pick\ mut]$, allowing a finer grained control over the search process.

1.1.4 Implementation

Based on the EvoCrash approach, we developed Botsing, an extendable framework for crash reproduction. Botsing relies on EvoSuite [36] for the code instrumentation during test generation and execution by using *evosuite-client* as a dependency. Our open-source implementation is available at https://github.com/STAMP-project/botsing. The current version of Botsing includes

both test seeding and model seeding as features. More information about the user documentation and development status are available in Chapter 3.

Test seeding

Test seeding relies on the implementation defined by Rojas *et al.* [87] and available in EvoSuite. This implementation requires the user to provide a list of test cases to consider for cloning and carving. In Botsing, we automated this process using the dynamic analysis of the test cases to automatically detect those accessing classes involved in a given stack trace. We also modified the standard guided initialization and guided mutation to preserve the call to the target method during cloning and carving.

Model seeding

The static analysis uses the reflection mechanisms of EvoSuite to inspect the compiled code of the classes involved in the stack traces and collect call sequences. The dynamic analysis relies on the test seeding mechanism used for cloning that allows inspecting an internal representation of the test cases obtained after their execution and collect call sequences. The resulting call sequences are then used to infer the transition system models of the classes using a 2-gram inference tool called YAMI [33]. A set of abstract object behaviors are selected from each of the models using a dissimilarity criterion (based on the Jaccard distance [53]), taken from VIBeS [34], a model-based testing tool, working with transition systems. If the size of this set is too small, the generated abstract object behaviors do not cover all of the transitions of the transition system. Also, using a small set of test cases can misguide the search process. On the contrary, if the size of the set is too large, the test concretization can become a time taking process. In this study, the size of this set is equivalent to the size of the individual population to make sure that we have enough test case to seed into the first population even if $Pr[pick\ init]$ is set to 1.0. The concretization of the test cases into objects for the objects pool is done using the EvoSuite API to create Java unit tests. The objects are then picked during the guided genetic algorithm execution in Botsing with user-defined probabilities $Pr[pick\ init]$ and $Pr[pick\ mut]$.

1.1.5 Empirical Evaluation

Behavioral model seeding exploits both test cases and source code, thereby *subsuming* test seeding regarding the observed behavior of the application that is reused during the search. Test seeding only uses dynamic analysis, which entails that it collects more accurate information from the potential usage scenarios of the software under test; it also means that this strategy collects more limited information for seeding. If these limited amounts of call sequences differ from the call sequences needed to reproduce the crash scenario, test seeding can misguide the crash reproduction search process. In order to assess the usage of test seeding applied to crash reproduction and our new model seeding approach during the guided initialization, we performed an empirical evaluation to answer the two research questions defined in Section .

RQ1 *What is the influence of test seeding used during initialization on search-based crash reproduction?* To answer this research question, we compare Botsing executions with *test seeding* enabled to executions where no additional seeding strategy is used (denoted *no seeding* hereafter), from their effectiveness to reproduce crashes and start the search process, the factors influencing this effectiveness, and the impact of test seeding on the efficiency. We divide **RQ1** into four sub-research questions:

RQ1.1 Can test seeding help to initialize the search process?
RQ1.2 Does test seeding help to reproduce more crashes?
RQ1.3 Does test seeding impact the efficiency of the search process?
RQ1.4 Which factors in test seeding impact the search process?

Table 1.1: Projects used for the evaluation with the number of crashes (**Cr.**), the average number of frames per stack trace ($\overline{\mathbf{frm}}$), the average cyclomatic complexity ($\overline{\mathbf{CCN}}$), the average number of statements ($\overline{\mathbf{NCSS}}$), the average line coverage of the existing test cases ($\overline{\mathbf{LC}}$), and the average branch coverage of the existing test cases ($\overline{\mathbf{BC}}$).

Application	Cr.	$\overline{\mathbf{frm}}$	$\overline{\mathbf{CCN}}$	$\overline{\mathbf{NCSS}}$	$\overline{\mathbf{LC}}$	$\overline{\mathbf{BC}}$
JFreeChart	2	6.00	2.75	63.01k	67%	59%
Commons-lang	22	2.04	3.28	13.38k	91%	87%
Commons-math	27	3.92	2.43	29.98k	90%	84%
Mockito	14	4.64	1.78	6.06k	97%	93%
Joda-Time	8	3.87	2.11	19.41k	89%	82%
XWiki	52	27.25	1.92	177.84k	73%	71%

RQ2 *What is the influence of behavioral model seeding used during initialization on search-based crash reproduction?* To answer this question, we compare Botsing executions with *model seeding* to executions with *test seeding* and *no seeding*. We also divide **RQ2** into four sub-research questions:
RQ2.1 Can behavioral model seeding help to initialize the search process compared to no seeding?
RQ2.2 Does behavioral model seeding help to reproduce more crashes compared to no seeding?
RQ2.3 Does behavioral model seeding impact the efficiency of the search process compared to no seeding?
RQ2.4 Which factors in behavioral model seeding impact the search process?

Setup

Crash selection In the earliest EvoCrash empirical study [98], Soltani *et al.* used a benchmark consisting of 50 crashes. They compared their search-based approach with the other crash reproduction approaches using this benchmark. In that study, they demonstrated that EvoCrash could replicate 82% of the crashes. Consequently, they introduced a new benchmark, which contains more complex and closer to the real world crashes, in their latest research [96]. The new benchmark includes 33 stack traces from 5 open source projects. They demonstrated that EvoCrash has a lower crash reproduction rate in the new benchmark. EvoCrash cannot replicate 12 crashes by setting any frame as the target frame.

For the empirical evaluation of model-seeding and test-seeding, we use JCrashPack (see D3.2) but exclude Elasticsearch due to incompatibilities with the Gradle build system used by that project. Table 1.1 provides more details about the extended benchmark.

Model inference Since the selected crashes for this evaluation is identified before the model inference process, we applied the dynamic analysis only on the test cases which use the involved classes in the crashes. During the static analysis, we spot all of the interesting test cases which call the methods of the classes which are appeared in the stack traces of the crashes. Next, we apply dynamic analysis only on the detected interesting test cases.

This filtering process helps us to shorten the model inference execution time without losing accuracy in the generated models.

Configuration parameters We used a budget of 62,328 fitness evaluations (corresponding on average to 15 minutes of executing Botsing with no seeding on our infrastructure) to avoid side effects on execution time when executing Botsing on different frames in parallel. We also fixed the population size to 100 individuals as it suggested by the latest study on search-based crash reproduction [96]. All other configuration parameters have their default value [87], and we used the default weighted sum scalarization fitness function (Equation 1.1) from Soltani *et al.* [96].

Test seeding During the guided initialization of the algorithm, test seeding can clone test cases according to a user-defined probability $Pr[clone]$. In addition to the default 0.2 value, we executed

model seeding with values 0.5, 0.8, and 1.0 for $Pr[clone]$. In the current EVOSUITE implementation, both $Pr[pick\ init]$ and $Pr[pick\ mut]$ are indicated as a single property called `p_object_pool`. This property indicates the probability of using the object pool during test generation (either during initialization or mutation). We left the default value of 0.3 for `p_object_pool`, and we only changed the value of $Pr[clone]$ in different configurations. Assessing the influence of test seeding on mutation is part of our future work.

Model seeding Our implementation of model seeding makes a distinction between using the object pool during guided initialization and guided mutation (as shown in Figure 2.1). This distinction enables to study the influence of seeding during the different steps of the algorithm independently. As discussed in Section 2.1.2, model seeding does not clone the test cases in the initial population. It only has the probability of using the collected objects in the object pool during the test generation. $Pr[pick\ init]$ indicates the probability of using object pool in the guided initialization, and $Pr[pick\ mut]$ means the probability of using object pool during the guided mutation. Since the focus of this study is using seeding to enhance the guidance of the search initialization, and $Pr[pick\ init]$ is the only property that we can use for modifying the probability of the object seeding in initialization, we use the following values for $Pr[pick\ init]$: 0.2, 0.5, 0.8, 1.0. The value of $Pr[pick\ mut]$ is fixed to 0.3.

For each frame (951 in total), we executed Botsing for *no seeding* (i.e., no additional seeding compared to the default parameters of Botsing) and each configuration of *model seeding*. Since *test seeding* needs existing test cases which are using the target class, we filtered out the frames that do not have any test for execution of this seeding strategy. Therefore, we executed each configuration of *test seeding* on the subset of frames (171 in total). To address the random nature of the evaluated search approaches, we repeated each execution 30 times.

Infrastructure We used 2 clusters (with 20 CPU-cores, 384 GB memory, and 482 GB hard drive) for our evaluation. For each stack trace, we executed an instance of Botsing for each frame which points to a class of the application. We discarded other frames to avoid generating test cases for external dependencies. We ran Botsing on 951 frames from 124 stack traces 30 times for no-seeding and each model-seeding strategy configuration. Also, we ran Botsing with test-seeding on 171 frames from 59 crashes. For this study, we executed a total of 186,560 independent executions. These executions took about 18 days.

Data analysis procedure

To check if the search process can achieve a better state using seeding strategies, we analyze the status of the search process after executing each of the cases (each run in one frame of a stack trace). We define 5 states:

(i) **not started**, the initial population could not be initialized, and the search did not start;
(ii) **line not reached**, the target line could not be reached;
(iii) **line reached**, the target line has been reached, but the target exception could not be thrown;
(iv) **ex. thrown**, the target line has been reached, and an exception has been thrown but produced a different stack trace; and
(v) **reproduced** the stack trace could be reproduced.

Since we repeat each execution 30 times, we use the majority of outcomes for a frame reproduction result. For instance, if Botsing could reproduce a frame in the majority of the 30 runs, we count that frame as a *reproduced*.

To measure the impact of each strategy in initializing the first population (**RQ1.1** and **RQ1.2**), we use the Odds Ratio (OR) because the distribution of the related data in this aspect is binary distributed (i.e., whether the search process can start the search or not). Also, we apply Fisher's exact test, with $\alpha = 0.05$ for the Type I error, to evaluate the significance of results.

Table 1.2: Evaluation results for comparing test-seeding and no-seeding in search initialization. $\overline{\text{rate}}$ and σ designate average successful search initialization rate and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Search started		Comparison to no s.		
	$\overline{\text{rate}}$	σ	better	no diff.	worse
no s.	29.5	3.94	-	-	-
test s. 0.2	27.4	8.49	0	0	4
test s. 0.5	26.9	9.22	0	0	5
test s. 0.8	27.9	7.67	1	0	4
test s. 1.0	26.9	9.22	0	0	5

We use the same procedure to assess the statistical difference of each strategy in crash reproduction rate (**RQ1.2** and **RQ2.2**). The distribution of related data in this aspect is binary too: a search process either reproduces a crash (the generated test replicates the stack trace from the highest frame which is reproduced by at least one of the other searches) or not.

Moreover, to answer **RQ1.3** and **RQ2.3**, which investigate the efficiency of the different strategies, we compare the number of fitness function evaluations needed by the search to reach crash reproduction. We use the Vargha-Delaney statistic [109] to appraise the effect size between strategies. In this statistic, a value lower than 0.5 for a pair of factors (A, B) gives that A reduces the number of needed fitness function evaluations, and a value higher than 0.5 shows the opposite. Also, we use the Vargha Delaney magnitude measure to partition the results into 3 categories having large, medium, and small impact. In addition, to examine the significance of the calculated effect sizes, we use the non-parametric Wilcoxon Rank Sum test, with $\alpha = 0.05$ for Type I error. For this case, since the number of reproductions could be different from one seeding configuration to another, executions that could not reproduce the frame simply reached the maximum allowed budget (62,328).

For all of the statistical tests in this study, we only use a level of significance $\alpha = 0.05$.

Also, it is notable that since the model inference (in model seeding) and test carving (in test seeding) can be applied as a one time process before running any search-based crash reproduction, we do not include them in the efficiency evaluation.

To answer **RQ1.4** and **RQ2.4**, we performed a manual analysis on the logs and crash reproducing test case (if any) produced by the executions of the crashes for which the search in one seeding configuration has a significant impact (according to the results of the previous sub-research questions) on (i) *initializing the initial population*, (ii) *crash reproduction*, (iii) or *search process efficiency* comparing to no-seeding. Based on the manual analysis, we used a card sorting strategy by assigning keywords to each frame result and grouping those keywords to identify influencing factors.

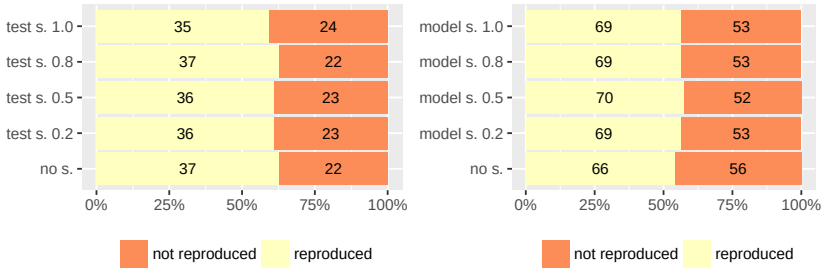
1.1.6 Evaluation Results

In this section, we present the results of the evaluation and answer the two research questions by comparing each seeding strategy with no-seeding.

Test seeding (RQ1)

Guided initialization effectiveness (RQ1.1) Table 1.2 indicates the number of crashes where test-seeding had a significant ($p - value < 0.05$) impact on the search initialization compared to no-seeding. As we can see in this table, any configuration of test-seeding has a negative impact on the search starting process for 4 or 5 crashes. Additionally, this strategy does not have any significant beneficial impact on this aspect except on one crash in $tests.0.8$.

Crash reproduction effectiveness (RQ1.2) In Figure 1.3a, the 59 crashes (100%) are classified for each seeding configuration of test-seeding and no-seeding according to the outcome observed in the majority of the 30 executions. Compared to no seeding, *test s. 0.8* reproduced the same amount of



(a) test-seeding vs. no-seeding (total) **(b)** model-seeding vs. no-seeding (total)

Figure 1.3: Outcomes observed in the majority of the executions for each crash in total.

Table 1.3: Evaluation results for comparing test-seeding and no-seeding in crash reproduction. $\overline{\text{rate}}$ and σ designate average crash reproduction rate and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Reproduction		Comparison to no s.		
	$\overline{\text{rate}}$	σ	better	no diff.	worse
no s.	25.4	9.65	-	-	-
test s. 0.2	23.5	10.93	1	0	4
test s. 0.5	23.8	10.76	1	0	4
test s. 0.8	23.4	10.74	2	0	5
test s. 1.0	23.7	11.01	2	0	5

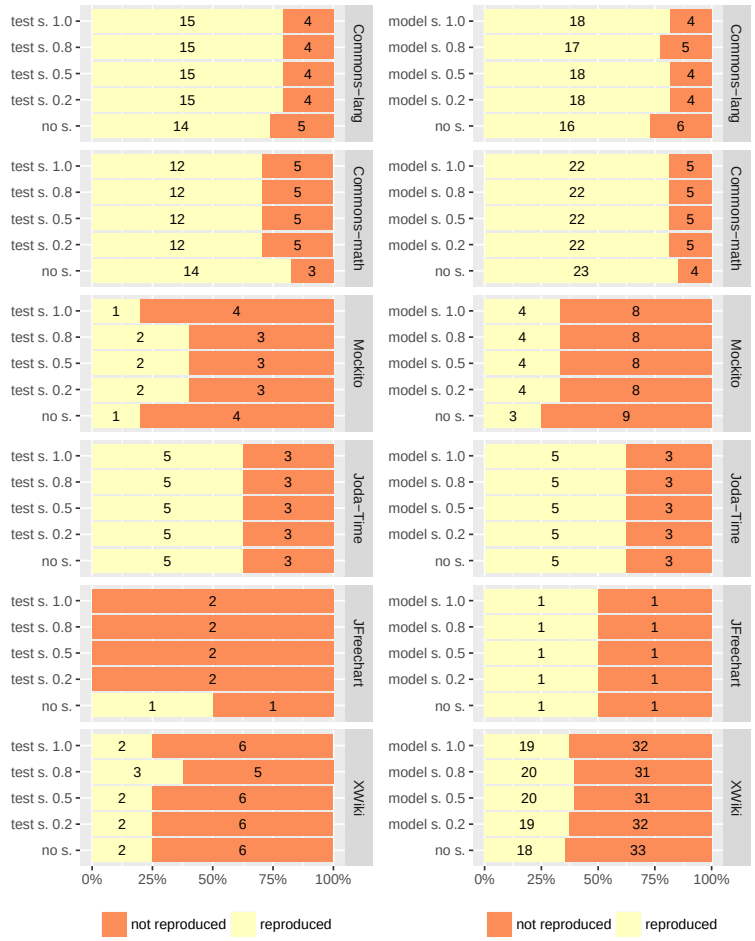
crashes. However, the other configurations of test-seeding reproduced fewer crashes in the majority of times.

Figure 1.4a indicates the number of reproduced crashes in the majority of runs for each application. As we can see, *test s. 0.8* reproduces one more crash in the XWiki project compared to no-seeding. However, in the other projects, test-seeding configurations perform either the same or worse when comparing to no-seeding.

Table 1.3 demonstrates the impact of test-seeding on the crash reproduction rate compared to no-seeding. It indicates that *test s. 0.2 & 0.5* have better crash reproduction rates for one of the crashes, while they perform significantly worse in 4 other crashes compared to no-seeding. The situation is almost the same for the other configurations of test seeding: *test s. 0.8 & 1.0* are significantly better in 2 crashes compared to no-seeding. However, they are significantly worse than no-seeding in 5 other crashes.

Crash reproduction efficiency (RQ1.3) Table 1.4 demonstrates the comparison of test-seeding and no-seeding in the number of needed fitness function evaluations for crash reproduction. The average number of fitness function evaluations increases when using test-seeding. Also, the values of the effect sizes indicate that the number of crashes which receive (large or medium) positive impacts from *test s. 0.2 & 0.5* for their reproduction speed is higher than the number of crashes that exhibit a negative (large or medium) influence. However, this is not the case for the other two configurations. In the worst case, *test s. 1.0* is considerably slower than no-seeding (with large effect size) in 13 crashes.

Influencing factors (RQ1.4) To answer this question, we manually analyzed the cases which cause significant differences, in various aspects, between no-seeding and test-seeding. From our manual analysis, we identified 3 factors of the test seeding process that influence the search: (i) **clone and mutate tests** (3 frames) used during guided initialization, (ii) **seed call sequences** (3 frames) that are used during the search process, and (iii) **expensive test execution for collecting tests**.



(a) test-seeding vs. no-seeding **(b)** model-seeding vs. no-seeding

Figure 1.4: Outcomes observed in the majority of the executions for each crash for each application.

Clone and mutate tests For the first factor, we observe that *cloning existing test cases* in the initial population leads to *reproduce new crashes* when the cloned tests include elements which are close to the crash reproducing test. For instance, all of the configurations of test seeding are capable of reproducing the crash LANG 6b, while no-seeding cannot reproduce it. For reproducing this crash, Botsing needs to generate a string of a specific format, and this format is available in the existing test cases which are seeded to the search process.

However, manually developed tests are not always helpful for crash reproduction. According to the results of Table 1.4, *test s. 1.0*, which always clones test cases, is considerably and largely slower than no-seeding in 13 crashes. In these cases, cloning all of the test cases to form the initial population can misguide the search process to reach the crash reproducing test. As an example, Botsing needs to generate a simple test case, which calls the target method with an empty string and null object, to reproduce crash LANG-12b. But, *test s. 1.0* clones tests which use the software under test in different ways. To summarize, the overall quality of results of our test seeding solution is highly dependent on the quality of the existing test cases in terms of factors like the distance of existing test cases to the area in which the crash occurs and the variety of input data.

Table 1.4: Evaluation results for comparing test-seeding and no-seeding in the number of fitness evaluations $\overline{\text{rate}}$ and σ designate average fitness function evaluations needed for crash reproduction and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Fitness		Comparison to no s.					
			large		medium		small	
	$\overline{\text{evaluations}}$	σ	< 0.5	> 0.5	< 0.5	> 0.5	< 0.5	> 0.5
no s.	10,467.8	22,368.13	-	-	-	-	-	-
test s. 0.2	14,089.7	25,464.80	4	3	1	1	2	-
test s. 0.5	13,366.2	25,043.79	5	3	1	-	2	1
test s. 0.8	14,254.1	25,496.19	3	4	1	5	1	3
test s. 1.0	13,856.0	25,097.07	3	13	4	3	1	3

Seed call sequences For the second factor, we observe that (despite the fixed value of $Pr[pick\ mut]$ for test seeding) the objects with call sequences carved from the existing tests and stored in the object pool help during the search. For instance, for crash MATH-4b, Botsing needs to initialize a `List` object with at least two elements before calling the target method in order to reproduce the crash. In test-seeding, such an object had been carved from the existing tests and allowed test seeding to reproduce the crash faster. Also, test-seeding can replicate this crash more frequently: the number of successfully replicated executions, in 30 runs, is higher with test-seeding.

In contrast, the carved objects can misguide the search process for some crashes which need another kind of call sequences. For instance, in crash MOCKITO-9b, Botsing cannot inject the target method into the generated test because the carved objects do not have the proper state to instantiate the input parameters of the target method.

In summary, if the involved classes in a given crash are well-tested, we have more chance to reproduce by utilizing test-seeding.

Expensive test execution for collecting tests The third factor points to the challenge of executing the existing test cases for seeding. The related tests for some crashes are either expensive (time/resource consuming) or challenging (due to the security issues) to execute. Hence, the EVOSUITE test executor, which is used by Botsing, cannot carve all of them. As an example, EvoSuite test executor spends more than 1 hour during the execution of the related test cases for replicating frame 2 of crash Math-1b. Eventually, it stopped without carving any object. Besides, EvoSuite test executor is not successful in running some of the existing tests. It throws an exception during this task. For instance, this executor throws `java.lang.SecurityException` during the execution of the existing test cases for CHART-4b, and it cannot carve any object for seeding.

In some cases, test-seeding faces the mentioned problems during the execution of all of the existing test cases for a crash. If test seeding cannot carve any object from existing tests, it will not have any beneficial call sequences in the object pool to seed during the search process. Hence, although the project contains some potentially valuable test scenarios for reproducing the given crash, there is no difference between no seeding and test seeding in these cases.

Summary (RQ1) Test seeding (for any configuration) loses against no-seeding in the search initialization because some of the related test cases of crashes are expensive or even impossible to execute. Also, we observe in the manual analysis that the lack of generality in the existing test cases prevents the crash reproduction search process initialization. In these cases, the carved objects from the existing tests mismatch the search process in the target method injection. Moreover, this seeding strategy can outperform no seeding in the crash reproduction and search efficiency for some cases (*e.g.*, LANG 6b), thanks to the call sequences carved from the existing tests. However, these carved call sequences can be detrimental to the search process in some cases, if the carved call sequences do not contain beneficial knowledge about crash reproduction, overusing them can misguide the search process.

Table 1.5: Evaluation results for comparing model-seeding and no-seeding in search initialization. $\overline{\text{rate}}$ and σ designate average successful search initialization rate and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Search started		Comparison to no s.		
	$\overline{\text{rate}}$	σ	better	no diff.	worse
no s.	29.2	4.72	-	-	-
model s. 0.2	29.5	3.87	2	0	1
model s. 0.5	29.7	2.75	2	0	0
model s. 0.8	30.0	0.00	3	0	0
model s. 1.0	30.0	0.28	3	0	0

Table 1.6: Evaluation results for comparing model-seeding and no-seeding in crash reproduction. $\overline{\text{rate}}$ and σ designate average crash reproduction rate and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Reproduction		Comparison to no s.		
	$\overline{\text{rate}}$	σ	better	no diff.	worse
no s.	21.3	12.32	-	-	-
model s. 0.2	21.6	12.00	3	0	1
model s. 0.5	21.8	11.86	4	0	0
model s. 0.8	21.9	11.92	4	0	1
model s. 1.0	22.0	11.58	4	0	0

Behavioral model seeding (RQ2)

Guided initialization effectiveness (RQ2.1) Table 1.5 provides a comparison between model-seeding and no-seeding in the search initialization rate. As shown in this Table, *model s. 0.2 & 0.5* significantly outperform no seeding in starting the search process for 2 crashes. This number increases to 3 for *model s. 0.8 & 1.0*. In contrast to test-seeding, most of the configurations of model-seeding do not have any significant negative impact on the search initialization (only *model s. 0.2* is significantly worse than no-seeding in one crash). It is notable that the average search initialization rate for *model s. 0.8 & 1.0* is 30/30 runs. Also, the standard deviations for these configurations are 0 or close to 0.

Crash reproduction effectiveness (RQ2.2) Figure 1.3b draws a comparison between model-seeding and no-seeding in the crash reproduction rate according to the results of the evaluation on all of the crashes (124 crashes). As depicted in this Figure, all of the configurations of model-seeding reproduce more crashes compared to no-seeding in the majority of runs. We observe that *model s. 0.2 & 0.5 & 1.0* reproduce 3 more crashes than no-seeding. In addition, in the best performance of model-seeding, *model s. 0.8* reproduces 70 out of 124 crashes. (6% more than no-seeding)

Figure 1.4b categorizes the content of Figure 1.3b per application. As we can see in this figure, model seeding replicates more crashes for XWiki, commons-lang, and Mockito. However, the number of reproduced crashes by no-seeding is one more than model-seeding for commons-math. For the other projects, the number of reproduced crashes does not change between no-seeding and different configurations of model-seeding. In general, various configurations of model-seeding reproduce 9 new crashes that no-seeding cannot reproduce.

Table 1.6 indicates the impact of model-seeding on the crash reproduction rate. As we can see in this table, *model s. 0.2* has a significantly better crash reproduction rate in 3 crashes. Also, other configurations of model-seeding are significantly better than no seeding in 4 crashes. This improvement is achieved by model-seeding while 2 out of 4 configurations of model-seeding have a significant unfavorable impact only on one crash.

Crash reproduction efficiency (RQ2.3) Table 1.7 compares the number of the needed fitness function evaluations for crash reproduction in model-seeding and no-seeding. As we can see in this table,

Table 1.7: Evaluation results for comparing model-seeding and no-seeding in the number of fitness evaluations $\overline{\text{rate}}$ and σ designate average fitness function evaluations needed for crash reproduction and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Fitness		Comparison to no s.					
			large		medium		small	
	$\overline{\text{evaluations}}$	σ	< 0.5	> 0.5	< 0.5	> 0.5	< 0.5	> 0.5
no s.	18,713.1	28,023.93	-	-	-	-	-	-
model s. 0.2	18,016.1	27,699.61	2	1	1	1	2	1
model s. 0.5	17,646.9	27,463.02	2	1	2	-	2	1
model s. 0.8	17,564.5	27,400.27	3	1	2	-	1	3
model s. 1.0	17,268.8	27,190.73	3	1	2	-	1	2

the average effort is reduced by using model-seeding. On average *mode s. 1.0* achieves the fastest crash reproduction.

According to this table, and in contrast to test-seeding, model-seeding's efficiency is slightly positive. Model-seeding has a large adverse effect size (as Vargha Delaney measures are higher than 0.5) on one crash while this number is higher for test-seeding (e.g., 13 for *test s. 1.0*). However, the number of crashes that model-seeding has a positive large or medium influence (as Vargha Delaney measures are lower than 0.5) on varies between 3 to 5.

Table 1.7 does not include the cost of model generation for seeding. However, we explain in the discussion section that since model generation is a one-time and affordable (it takes a few minutes) process, this task does not impose a notable burden on the crash reproduction efficiency.

Influencing factors (RQ2.4) We manually analyzed the crashes which lead to the significant differences between different configurations of model seeding and no seeding. In doing so, we have identified 4 influencing factors in model-seeding on the search-based crash reproduction, namely: (i) using **dissimilar call sequences** (2 frames) for guided initialization, (ii) having **multiple information sources** (18 frames) to infer the behavioral models, (iii) **prioritizing the call sequences** (3 frames) for seeding by focusing on the classes involved in the stack trace, and (iv) having a **fixed size for the abstract object behavior selection** from usage models.

Dissimilar call sequences Using *dissimilar call sequences* to populate the object pool in model seeding seems particularly useful for search efficiency compared to test seeding. In particular, if the number of test cases is large, model seeding enables to (re)capture the behavior of those tests in the model and regenerate a smaller set of call sequences which maximize diversity, augmenting the probability to have more diverse objects used during the initialization. For instance, Botsing with model-seeding is statistically more efficient than other strategies for replicating crash XWIKI-13141. Through our manual analysis we observed that model-seeding could replicate crash XWIKI-13141 in the initial population in 100% of cases, while the other seeding strategies replicate it after a couple of iterations. In this case, despite the large size of the target class model (35 transitions and 17 states), the diversity of the selected abstract object behaviors guarantees that Botsing seeds the reproducing test cases to the initial population.

Multiple information sources Having *multiple sources* to infer the model from helps to select diversified call sequences compared to test seeding. For instance, the sixth frame of the crash XWIKI-14556 points to a class called `HqlQueryExecutor`. No seeding cannot replicate this crash because it does not have any guidance from existing solutions. Also, since the test carver could not detect any existing test which is using the related classes, this seeding strategy does not have any knowledge to achieve reproduction. In contrast, the knowledge required for reproducing this crash is available in the source code, and model-seeding learned it from static analysis of this resource. Hence, this seeding strategy is successful in accomplishing crash reproduction.

Prioritizing the call sequences By *prioritizing classes* involved in the stack trace for the abstract object behaviors selection, the object pool contains more objects likely to help to reproduce the crash. For instance, for the 10th frame of the crash LANG-9b, model seeding could achieve reproduction in the majority of runs, compared to 0 for test and no seeding, by using the class `FastDateParser` appearing in the stack trace.

Fixed size for the abstract object behavior selection The last factor points to the fixed number of the generated abstract object behaviors from each model. In some cases, we observed that model-seeding was not successful in crash reproduction because the usage models of the related classes were large, and it was impossible to cover all of the paths with 100 abstract object behaviors. As such, this seeding strategy missed the useful dissimilar paths in the model. As an example, model-seeding was not successful in replicating crash XWIKI-8281 (which is replicated by no-seeding and test-seeding). In this crash, the unfavorable generated abstract object behaviors for the target class misguided the search process in model seeding.

Summary (RQ2) Model seeding achieves a better search initialization ratio compared to no seeding. With respect to the best achievement of model seeding (*model s. 0.8 & 1.0*), they decrease the number of not started searches in 3 crashes. Also, compared to no seeding, model seeding increases the number of crashes that can be reproduced in the majority of times to 6%. Additionally, it reproduces 9 new crashes which are unreproducible with no-seeding. This is notable since using test seeding leads to more unfavorable impacts rather than beneficial ones. Model seeding takes on average less fitness function evaluations, compared to no seeding, while this number slightly increases in test seeding. Also, in contrast with test seeding, model seeding delivers more positive significant impact on the efficiency of the search process compared to no seeding.

The helping factors include the *seeding of call sequences* (as for test seeding), the diversity of the sources used to build the behavioral model and the selection of dissimilar call sequences as well as their prioritization, based on the stack trace.

In general, model seeding outperforms test seeding in all of the aspects of search-based crash reproduction. Also, there are some crashes that Botsing can only reproduce when using model seeding. However, the achieved improvement is not a dramatically departure from the no seeding approach.

1.1.7 Discussion

Practical implications

Cost Generating seeds comes with a cost. For our worst case, XWIKI-13916, we collected 286K call sequences from static and dynamic analysis and generated 7,880 models from which we selected 6K abstract object behaviors. We repeated this process 10 times and found the average time for call sequence collection to be 14.2 seconds; model inference took 77.8 seconds; and abstract object behavior selection and concretization took 51.5 seconds. We do note however that the model inference is a one-time process that could be done offline (in a continuous integration environment). After the initial inference of models, any search process can utilize model seeding. To summarize, the total initial overhead is $\sim$ 2.5 minutes, and the total nominal overhead is around $\sim$ 1.25 minute. We argue that **the overhead of model seeding is affordable giving its increased effectiveness.** The initial model inference can also be incremental, to avoid complete regeneration for each update of the code, or limited to subparts of the application (like in our evaluation where we only applied static and dynamic analysis for classes involved in the stack trace). Similarly, abstract object behavior selection and concretization may be prioritized to use only a subset of the classes and their related model. In our current work, this prioritization is based on the content of the stack traces. Other prioritization heuristics, based for instance on the size of the model (reflecting the complexity of the behavior), is part of our future work.

Applicability and effectiveness In general, test seeding alone does not help in making crash reproduction more effective. Furthermore, it also causes more negative impacts on the search-based crash reproduction. We did not observe this behavior with model seeding as it always performs better than no seeding with different configurations. As such, we observe that **model seeding can reproduce more crashes than other strategies**. Also, test seeding's performance depends on the existence and quality of the existing test cases. As we observed, EvoSuite's test carver could not find any relevant test in half of the crashes. Since model seeding also exploits test cases, thereby subsuming test seeding regarding the observed behavior of the application that is reused during the search, greater performance can be attributed to the analysis of the source code translated in the model. In our experiments, we did not have any crash without relative usage model.

In our experiments, various configurations of model seeding reproduced 8 new crashes that neither test seeding nor no seeding strategies could reproduce. Additionally, **only model seeding could reproduce stack traces with more than seven frames** (*e.g.*, LANG-9b). Still, model seeding missed the reproduction of one crash which is reproduced by no seeding. Despite the achieved improvements by model seeding, this seeding strategy could not outperform no-seeding dramatically (crash reproduction improved by 6%). To better understand the reasons for the results, we manually analyzed the logs of Botsing executions on the crashes for which model seeding could not show any improvements. Through this investigation, we noticed that the generated usage models in these cases are limited and they do not contain the beneficial call sequences for covering the particular path that we need for crash reproduction. The average size of the generated model in this study is 7 states and 14 transitions. We believe that by collecting more call sequences from different sources (*i.e.,* log files), model seeding can increase the number of crash reproductions.

Extendability The usage models can be inferred from any resource providing call sequences. In this study, we used the call sequences derived from the source code and existing test cases. However, we can extend the models with extra resources (*e.g.*, execution logs). Also, the abstract object behavior selection approach can be adapted according to the problem. In this study, we used the dissimilarity strategy to increase the diversity of the generated tests.

Recommendation We, therefore, **recommend to use behavioral model seeding** as the seeding overhead is compensated by quantitative (more reproduced crashes and fewer failures) and qualitative (*e.g.*, by using relevant objects in the test cases) improvements.

Model seeding configuration

Model seeding can be configured with different $Pr[pick\ init]$ and $Pr[pick\ mut]$ probabilities. Like many other parameters in search-based test case generation [14], the values of those parameters could influence our results. Although a full investigation of the effect of $Pr[pick\ init]$ and $Pr[pick\ mut]$ on the search process is beyond the scope of this deliverable, we set up a small experiment on a subset of crashes (10 crash in total) with 15 new configurations, each one run 10 times.

Tables 1.8 and 1.9 present the configurations used for $Pr[pick\ init]$ and $Pr[pick\ mut]$ with, for each one, the crash reproduction effectiveness (Table 1.8), and the crash reproduction efficiency (Table 1.9). In general, we observe that changing the probability of picking an object during guided initialization ($Pr[pick\ init]$) as an impact on the search and leads to more reproduced crashes with a fewer number of fitness evaluations. Changing the probability of picking an object during mutation ($Pr[pick\ mut]$) does not seem to have a large impact on the search. A full investigation of the effects of $Pr[pick\ init]$ and $Pr[pick\ mut]$ on the search process is part of our future work.

Threats to validity

Internal validity We selected 124 crashes from 5 open source projects: 33 of crashes have previously been studied [96] and we added additional crashes from Xwiki and Defects4J (see Sec-

Table 1.8: Evaluation results for comparing different configurations of model seeding in crash reproduction. $\overline{\text{rate}}$ and σ designate average crash reproduction rate and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Reproduction		Comparison to other conf.		
	$\overline{\text{rate}}$	σ	better	no diff.	worse
Pr[init]=0.0 Pr[mut]=0.3	18.8	13.81	0	0	11
Pr[init]=0.0 Pr[mut]=0.6	19.0	13.64	0	0	10
Pr[init]=0.0 Pr[mut]=0.9	19.0	13.55	0	0	13
Pr[init]=0.2 Pr[mut]=0.0	20.4	12.42	2	0	2
Pr[init]=0.2 Pr[mut]=0.3	19.6	12.87	0	0	7
Pr[init]=0.2 Pr[mut]=0.6	19.8	12.88	1	0	5
Pr[init]=0.2 Pr[mut]=0.9	19.4	13.15	0	0	7
Pr[init]=0.5 Pr[mut]=0.0	20.8	12.17	3	0	1
Pr[init]=0.5 Pr[mut]=0.3	20.6	12.29	3	0	2
Pr[init]=0.5 Pr[mut]=0.6	19.4	13.24	0	0	7
Pr[init]=0.5 Pr[mut]=0.9	20.0	12.58	1	0	5
Pr[init]=0.8 Pr[mut]=0.0	21.8	11.46	8	0	0
Pr[init]=0.8 Pr[mut]=0.3	21.6	11.53	6	0	0
Pr[init]=0.8 Pr[mut]=0.6	21.8	11.77	8	0	0
Pr[init]=0.8 Pr[mut]=0.9	20.8	11.96	3	0	2
Pr[init]=1.0 Pr[mut]=0.0	21.6	11.53	6	0	0
Pr[init]=1.0 Pr[mut]=0.3	23.0	11.31	12	0	0
Pr[init]=1.0 Pr[mut]=0.6	21.6	11.82	8	0	0
Pr[init]=1.0 Pr[mut]=0.9	22.6	11.30	11	0	0

tion 2.1.4). Since we focused on the effect of seeding during guided initialization, we fixed the $Pr[pick\ mut]$ value (which, due to the current implementation of Botsing, is also used as $Pr[pick\ init]$ value in test seeding) to 0.3, the default value used in EvoSuite for unit test generation. The effect of this value for crash reproduction, as well as the usage of test and model seeding in guided initialization, is part of our future work. We cannot guarantee that our extension of Botsing is free of defects. We mitigated this threat by testing the extension and manually analyzing a sample of the results. Finally, each frame has been run 30 times for each seeding configuration to take randomness into account and we derive our conclusion based on standard statistical tests [13, 82].

External validity We cannot guarantee that our results are generalizable to all crashes. However, we recall the diversity of the applications and associated crashes. We collect the crashes from Defects4j, as state-of-the-art, and extend it by more crashes from XWiki, an industrial and complex open source application. Variations in the performance of the approaches also suggest mitigation of this threat.

Verificability A replication package of our empirical evaluation is available at https://github.com/STAMP-project/ExRunner-bash/tree/master. The complete results and analysis scripts are also provided in this package. Our extension of Botsing is released under a LGPL 3.0 license and available at https://github.com/STAMP-project/botsing.

1.2 Extraction and seeding of common software behaviors for unit test generation

Similarly to crash-reproduction, model seeding can be used in search-based unit test generation. The following section presents the research questions and the empirical evaluation protocol that will be applied in our future work to evaluate the impact of model seeding on search-based unit test generation.[3]

[3]The evaluation infrastructure is available at https://github.com/STAMP-project/evosuite-model-seeding-empirical-evaluation.

Table 1.9: Evaluation results for comparing different configurations of model seeding in the number of fitness evaluations rate and σ designate average fitness function evaluations needed for crash reproduction and standard deviation, respectively. The numbers in the comparison only count the statistically significant cases.

Conf.	Fitness		Comparison to other configurations					
			large		medium		small	
	$\overline{\text{evaluations}}$	σ	< 0.5	> 0.5	< 0.5	> 0.5	< 0.5	> 0.5
Pr[init]=0.0 Pr[mut]=0.3	23,456.5	30,105.20	-	5	-	3	-	3
Pr[init]=0.0 Pr[mut]=0.6	23,066.3	29,976.23	-	2	-	5	-	3
Pr[init]=0.0 Pr[mut]=0.9	23,030.9	30,001.82	-	7	-	4	1	2
Pr[init]=0.2 Pr[mut]=0.0	20,179.0	29,012.80	-	-	1	2	1	-
Pr[init]=0.2 Pr[mut]=0.3	21,803.0	29,620.34	-	2	2	5	-	1
Pr[init]=0.2 Pr[mut]=0.6	21,448.9	29,441.74	-	2	-	3	1	2
Pr[init]=0.2 Pr[mut]=0.9	22,214.6	29,752.12	-	2	2	5	1	1
Pr[init]=0.5 Pr[mut]=0.0	19,371.3	28,668.58	-	-	2	1	3	-
Pr[init]=0.5 Pr[mut]=0.3	19,766.8	28,849.00	-	-	1	2	2	-
Pr[init]=0.5 Pr[mut]=0.6	22,245.2	29,729.80	-	-	-	4	-	3
Pr[init]=0.5 Pr[mut]=0.9	21,030.0	29,302.03	-	2	-	3	1	2
Pr[init]=0.8 Pr[mut]=0.0	17,329.0	27,693.98	2	-	6	-	-	1
Pr[init]=0.8 Pr[mut]=0.3	17,710.5	27,919.28	1	-	4	-	3	-
Pr[init]=0.8 Pr[mut]=0.6	17,327.0	27,694.60	2	-	6	-	-	-
Pr[init]=0.8 Pr[mut]=0.9	19,383.3	28,659.38	-	-	1	1	2	2
Pr[init]=1.0 Pr[mut]=0.0	17,730.5	27,906.92	1	-	4	-	3	1
Pr[init]=1.0 Pr[mut]=0.3	14,863.9	26,275.53	7	-	3	-	-	-
Pr[init]=1.0 Pr[mut]=0.6	17,692.5	27,930.17	2	-	5	-	1	-
Pr[init]=1.0 Pr[mut]=0.9	15,656.9	26,798.15	7	-	5	-	1	-

In their work, Rojas *et al.* [87] evaluate the impact of various seeding strategies, including test seeding (see Section 1.1.2). Therefore, in this study, we focus on model seeding to provide answers to the following research questions:

RQ1 What is the influence of *behavioral model seeding used during initialization* on search-based unit test generation effectiveness?

RQ2 Does *behavioral model seeding used during initialization* produce test cases similar to the manually written test suites?

The first research question compares the fault detection capabilities of test cases generated using no seeding, test seeding, and model seeding. The second research question compares the similarity of the test cases generated using no seeding, test seeding, and model seeding with manually written test cases.

Setup

Classes under test selection For the classes under test, we consider Defects4J [57], a state-of-the-art benchmark containing 438 bugs with their fixes from 6 different Java projects. Additionally, we generate test cases for 120 classes from `sat4j-core` and 23 classes from `proactive-common-api` containing at least one non-trivial method (with a cyclomatic complexity above 3).

Model inference To generate the models, we collected all the call sequences observed for all the classes in `proactive-common-api` and `sat4j-core`.

Configuration parameters We use DynaMOSA [81] as it is the best state-of-the-art algorithm for unit test case generation [79], with the default set of objectives. We used a budget of 5 minutes for each EvoSuite run with a population size of 50 individuals (the default value in EvoSuite). All other configuration parameters have their default value [87]. To take randomness into account, we repeated each execution 100 times.

Test seeding As for search-based crash reproduction, we leave the default value of 0.3 for `p_object_pool`, and we only change the value of $Pr[clone]$ in different configurations to allow a comparison with model seeding.

Model seeding We use the same values as for crash reproduction: 0.2, 0.5, 0.8, 1.0 for $Pr[pick\ init]$ and 0. for $Pr[pick\ mut]$.

Data analysis procedure

To check if the search process can achieve a better fault detection capability, we use *Pit* [4], a state-of-the-art mutation analysis tool, and compare the number of mutants hardly killed bu the different test suites.

We measure the effectiveness of the generated test suite using *mutation analysis* on the test suites generated using no seeding, test seeding, and model seeding. Mutation analysis is a high-end coverage criterion, and mutants are often used as substitutes for real faults since previous studies highlighted its significant correlation with fault-detection capability [58, 11]. Besides, mutation analysis provides a better measure of the test effectiveness compared to more traditional coverage criteria [113] (e.g., branch coverage).

For the mutation analysis, we used PIT[5], which is a state-of-the-art mutation testing tool for Java code, with the default mutation operators: `Conditionals Boundary`, `Increments`, `Invert Negatives`, `Math`, `Negate Conditionals`, `Return Values`, and `Void Method Calls` operators.

To answer **RQ1**, we compute the mutation scores achieved by the test suites generated for each class under test. To measure the impact of each strategy on the mutation score, we use the Odds Ratio (OR) because the distribution of the related data in this aspect is binary distributed (i.e., whether a particular mutant is killed or not). Also, we apply Fisher's exact test, with $\alpha = 0.05$ for the Type I error, to evaluate the significance of results.

To answer **RQ2**, we compare the similarity between the manually written test cases and the generated ones. For that, we collect the sequences of methods called on each object in the generated test cases and compare those sequences with the one collected from the manually written test cases using the Jaccard index [53].

[4] http://pitest.org
[5] http://pitest.org

Chapter 2

Code instrumentation extension

This chapter presents the extension of the code instrumentation used by Botsing to monitor the execution of a class under test. Code instrumentation consists of the injection of probes into the bytecode of a Java application to monitor the runtime behavior of specific classes. Once executed, the instrumented code will produce additional information, collected by Botsing for various tasks. For instance, in Chapter 1, the source code is instrumented to perform dynamic analysis of the system and produce log messages, used to learn a transition system representing the usages of the different classes. In this chapter, we present our extension of the current code instrumentation mechanism and its usage during the search process for (i) class integration testing in Section 2.1, and (ii) search-based crash reproduction in Section 2.2.

2.1 Class integration testing

Search-based approaches have been applied to a variety of white-box testing activities [47], among which test case and data generation [69] and crash reproduction [95]. In white-box testing, most of the existing work has focused on the unit level, where the goal is to generate test cases/suites that achieve high structural (e.g., branch) coverage for a single class. Prior work has shown that search-based unit test generation can achieve high code coverage [8, 25, 79], detect real-bugs [40, 92], and help developers during debugging activities [27].

Despite these undeniable advantages, in recent years, researchers have investigated the limitations of the generated unit tests [42, 92, 90]. Prior studies have questioned the effectiveness of the generated unit tests with high code coverage in terms of their capability to detect real faults or to kill mutants when using mutation coverage. For example, Gay *et al.* [42] have highlighted how traditional code coverage could be a poor indicator of test effectiveness (in terms of fault detection rate and mutation score). Shamshiri *et al.* [92] have reported that around 50% of faults remain undetected when relying on generated tests with high coverage. Similar results have also been observed for large industrial systems [9].

Gay *et al.* [42] have observed that traditional unit-level adequacy criteria measure only whether certain code elements are reached, but not *how* each element is covered. The quality of the test data and the paths from the covered element to the assertion play an essential role for better test effectiveness. As such, they have advocated the need for more reliable adequacy criteria for test case generation tools. While these results hold for generated unit tests, other studies on manually-written unit tests have further highlighted the limitation of unit-level code coverage criteria [113, 90].

In this deliverable, we explore the usage of the integration code between coupled classes as guidance for the test generation process. The idea is that, by exercising the behavior of a class under test E (the calleE) through another class R (the calleR) calling its methods, R will handle the creation of complex parameter values and exercise valid usages of E. In order words, the caller R might contain

integration code that (1) enables to create better test data for the callee E, and (2) allows to better validate the data returned by E.

Integration testing can be approached from many different angles [56, 76]. In our case, we focus on *class integration testing* between a caller and a callee [91]. Class integration testing aims to assess whether two or more classes work together properly by thoroughly testing their interactions [91]. Our idea is to complement unit test generation for a class under test by looking at its integration with other classes. To that end, we define a novel structural adequacy criterion we call *Coupled Branches Coverage* (CBC), targeting specific integration points between two classes. Coupled branches are pairs of branches $\langle r, e \rangle$, with r a branch of the caller, and e a branch of the callee, such that an integration test that exercises branch r also exercises branch e.

Furthermore, we implement a search-based approach that generates integration-level test suites, based on the CBC criterion. We coin our approach CLING (for class integration testing). CLING uses a state-of-the-art many-objective solver that generates test suites maximizing the number of covered coupled branches. For the guidance, CLING uses novel search heuristics defined for each pair of coupled branches (the objectives).

We conducted an empirical study on 140 well-distributed pairs of caller and callee classes extracted from five open-source Java projects. Our results show that CLING can achieve up to 99% CBC scores, with an average CBC coverage of 50% across all classes. We analyzed the benefits of the integration-level test cases generated by CLING compared to unit-level test generated by EVOSUITE, a state-of-the-art generator of unit-level tests. In particular, we assess whether integration-level tests generated by CLING allow to kill mutants and detect faults that would remain uncovered when relying on generated unit tests.

According to our results, on average, CLING allows killing 10% of mutants per class that cannot be detected by unit tests generated with EVOSUITE for both the caller and the callee. The improvements in mutation score are up to 60% for certain classes, such as the `Period` class in the Joda Time subject system. Finally, we found 29 integration faults that were detected only by the integration tests generated with CLING (and not through unit testing with EVOSUITE).

2.1.1 Background

McMinn defined search-based software testing (SBST) as *"using a meta-heuristic optimizing search technique, such as a genetic algorithm, to automate or partially automate a testing task"*. Within this realm, test data generation at different testing levels (such as *unit testing*, *integration testing*, *etc.*) has been actively investigated [69]. This section provides an overview of earlier work in this area.

Search-based approaches for unit testing

SBST algorithms have been extensively used for unit test generation. Previous studies on search-based unit testing confirmed that thus generated tests achieve a high code coverage [80, 24], real-bug detection [8], and debugging cost reduction [99, 83], complementing manually-written tests.

From McMinn *et al.*'s [69] survey about search-based test data generation, we observe that most of the current approaches rely on the control flow graph (CFG) to abstract the source code and represent possible execution flows. The $CFG_m = (N_m, E_m)$ represents a method m as a directed graph of *basic blocks* of code (the nodes N_m), while E_m is the set of the control flow edges. In a CFG, an edge connects a basic block n_1 to another one n_2 if the control may flow from the last statement of n_1 to the first statement of n_2.

Listing 2.1 presents the source code of Person class, representing a person and her transportation habits. A $Person$ can drive home (lines 4-10), or add energy to her car (lines 12-18). The right hand side of Figure 2.2 presents the CFG of the different methods, with the labels of the nodes representing the line numbers in the code.

Many structural-based approaches combine two common heuristics to reach a high branch and statement coverage in unit-level testing. These two heuristics are *approach level* and *branch distance*.

Listing 2.1: Class Person

```
class Person{
    private Car car = new Car();
    protected boolean lazy = false;
    public void driveToHome(){
        if (car.fuelAmount < 100) {
            addEnergy();
        } else {
            car.drive();
        }
    }

    protected void addEnergy(){
        if (this.lazy) {
            takeBus();
        } else {
            car.refuel();
        }
    }   }
```

Branch distance is a heuristic (based on a set of rules) measuring, for a branching node, the distance to true and the distance to false for a particular execution of the program. *Approach level* is a heuristic for measuring the distance between the execution path and a target node in a CFG. To describe how this heuristic measures this distance, we rely on the concepts of *post-dominance* and *control dependency* [7].

As an example, in Figure 2.2, $node8$ is control dependent on $node5$ and $node8$ post-dominates edge $\langle 5, 8\rangle$. *Approach level* is the minimum number of control dependencies between a target node and an executed path by a test case.

In search-based unit testing, each generated test case is a sequence of method calls to a target class. This call sequence can be generated randomly, or it can be generated using existing resources. Goffi *et al.* [45] leverage existing documentation in this process, but for various reasons it does not allow to detect all bugs [86, 20, 118]. Rojas *et al.* [87] collect the usages of classes in the existing test cases to generate the call sequences. To reach that goal, they need to execute each of the existing tests to find the call sequences; this may be a time taking process.

In this study, we focus on utilizing the usage of a class by the other classes. For this purpose, we use the Class-level Control Flow Graph (CCFG) of a target class and another class, which uses it, to generate integration tests between these two classes.

Search-based approaches for integration testing

Integration testing aims at finding faults that relate to the interaction between components. We discuss existing integration testing criteria and explain the search-based approaches that use these criteria to define fitness functions for automating integration level testing tasks.

Integration testing criteria Jin *et al.* [56] categorize the connections between two procedures into four levels for testing: *call couplings* occur when one procedure calls another one; *parameter couplings* happen when a procedure passes a parameter to another one; *shared data couplings* occur when two procedures refer to the same data objects; *external device coupling* happens when two procedures access the same storage device. They introduce integration testing criteria according to the data flow graph (containing the definitions and usages of variables at the integration points) of procedure-based

software. Their criteria, called *coupling-based testing criteria*, require that the developed tests execute paths in the CFG of a procedure (the *caller procedure*) which starts from the definition of a variable to a node (the *call site*) which calls another procedure.

Harrold *et al.* [48] introduced data flow testing for classes focusing on method integration. They define three levels of testing: *intra-method testing*, which tests an individual method (= unit testing); *inter-method testing*, in which a public method is tested that (in)directly calls other methods of a class, and *intra-class testing*, in which the interactions between calls to public methods in various sequences are tested. For data flow testing of the last two levels, they defined the *Class-level Control Flow Graph* (CCFG). The CCFG of class C is a directed graph $CCFG_C = (N_{Cm}, E_{Cm})$ which is a composition of the control flow graphs of methods in C; the CFGs are connected through their call sites to methods in the same class [48]. This graph demonstrates all paths that might be crossed within the class by calling its methods or constructors.

In our approach, we also rely on the CCFG. As an example of its construction, the CCFG of class $Person$ is created by merging the CFGs of its method as demonstrated in Figure 2.2. For example, in the CFG of the method `Person.driveToHome()`, the $node10c$ is a call site to `Person.addEnergy()`.

A special case is represented by the polymorphic interactions that need to be tested. Alexander *et al.* [2, 3, 5, 4] have used the data flow graph to define testing criteria for integrations between classes which are extending each other.

Search-based approaches Search-based approaches are widely used for test ordering [112, 103, 49, 110, 17, 55, 21, 67, 46, 1, 32, 22, 110], typically with the aim of executing those tests with the highest likelihood of failing earlier on. However, search-based approaches have been rarely used for generating integration tests. Ali Khan *et al.* [6] have proposed an evolutionary approach which detects the coupling paths in the data flow graph of classes and have used it to define the fitness function for the genetic algorithm. Then, this defined fitness function aids the genetic algorithm to generate tests for the detected coupling paths. Moreover, they proposed another approach for the same goal, which uses Particle Swarm Optimization [59]. However, they did not perform any evaluation for examining the quality of the generated tests by this approach. Also, they did not check if the generated tests by these approaches can complement the generated tests of existing search-based unit testing approaches.

In this study, we propose a novel approach for class integration test generation. Instead of using the data flow graph, which is relatively expensive to construct as it needs to find the coupling paths, we use the information available in the class call graph of the classes to calculate the fitness of the generated tests. Also, we assess the influence of the generated tests by our approach by different metrics.

Search-based approaches for other testing levels

Arcuri [12] proposed an evolutionary-based white-box approach for system-level test generation for RESTful APIs. A test for a RESTful web service is a sequence of HTTP requests. The proposed approach (EvoMaster) generates these tests to cover three types of targets: (i) all of the statements in the System Under Test (SUT); (ii) all of the branches in the SUT; (iii) different returned HTTP status codes. Although this approach tests different classes in the SUT, it does not systematically target different integration scenarios between classes.

In contrast to EvoMaster, other proposed approaches in literature perform black-box fuzz testing. As defined by Holler *et al.* [52] *"Fuzz testing is an automated technique providing random data as input to a software system in the hope to expose a vulnerability "*. These approaches use information like grammar specifications [52, 18, 30, 44] or feedback from the program during the execution of tests [77]. These approaches do not have any knowledge about classes in the SUT. Hence, their search processes is not guided by the integration of classes.

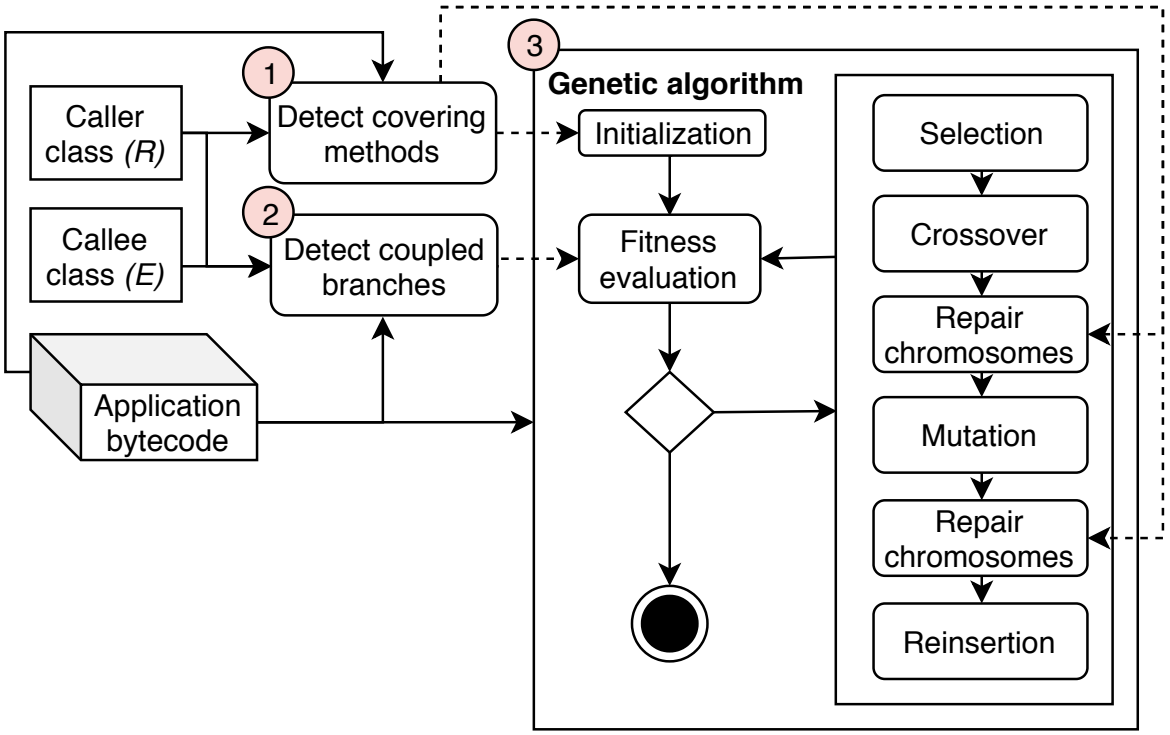


Figure 2.1: General overview of CLING

Our proposed approach performs white-box testing. It monitors the interaction between the target classes and strives to cover different integration scenarios between them.

2.1.2 Class integration testing

The main idea of our class integration testing approach (hereinafter referred to as CLING) is to test a class by leveraging its usage in another class. More specifically, we focus on the call between the former, the callee (E), and the latter, the caller (R). By doing so, we benefit from the additional context setup by R before calling E (*e.g.*, initializing a complex input parameter), and the additional post-processing after E returns (*e.g.*, using the return value later on in R), thus implicitly adding assertions on the behavior of E.

Figure 2.1 presents the general overview of CLING. CLING takes as input a couple of caller-callee $\langle R, E \rangle$ classes with at least one call (denoted *call site* hereafter) from R to E. Since the goal of CLING is to generate test cases covering E by calling methods in R, the first step (①) collects the list of *covering methods* in R that, when called, may directly or indirectly cover statements in E. This list is later used during the generation process to ensure that test cases contain calls to covering methods. The second step (②) analyses the CCFGs of R and E to identify the coupled branches between R and E used later on to guide the search. Finally, the generation of the test cases (③) uses a genetic algorithm with two additional *repair* steps, ensuring that the crossover and mutation only produce test cases able to cover lines in E. The result is a test suite for E, whose test cases call methods in R covering method call interactions with E.

The remainder of this section describes our novel underlying Coupled Branches Criterion, the corresponding search-heuristics, and test case generation in CLING.

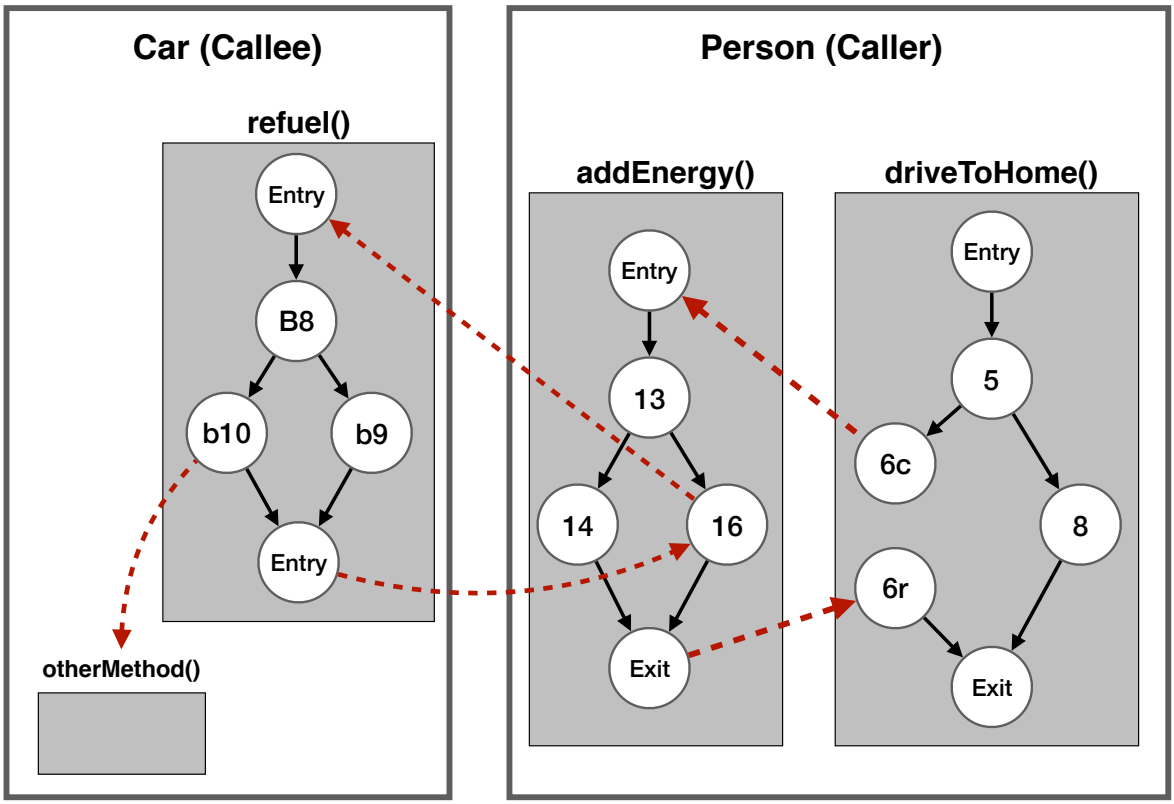


Figure 2.2: Class-level CFG for the two classes `Person` (caller) and `Car` (callee)

Coupled Branch testing criterion

To test the integration between two classes E and R, we need to define a coverage criterion that help us to measure how thoroughly a test suite T exercises the interaction calls between E and R. One possible coverage criterion would consist in testing all possible paths (*inter-class path coverage*) that start from the entry node of the caller R, execute the integration calls to E and terminate in one of the exit points of R. However, such a criterion will be affected by the *path explosion problem* [23]: the number of paths increases exponentially with the cyclomatic complexity of E and R, and thus the number of interaction calls between the two classes.

To avoid this issue, we define an integration-level coverage criterion, namely Coupled Branch Criterion (CBC), where the number of coverage targets remains polynomial to the complexity of E and R. More precisely, CBC focuses on call coupling between a caller class R and a callee class E. Intuitively, let $s \in R$ be a call site, i.e., a call statement to a method of the class E. Our criterion requires to cover all pairs of branches (b_r, b_e), where b_r is a branch in R that leads to s (the method call), and b_e is a branch of the callee E that is not trivially covered by every execution of E.

Before introducing the formal definition of the criterion, let us consider the example of caller and callee in Figure 2.2. The code for the class `Person` is reported in Figure 2.1. The class `Person` contains two methods, `addEnergy()` and `driveToHome()`, with the latter invoking the former (line 6 in Listing 2.1). The method `Person.addEnergy()` invokes the method `refuel()` of the class `Car` (line 16 in Listing 2.1). The method `Person.driveToHome()` invokes the method `Car.drive()` (line 8 in Listing 2.1). Therefore, the class *Person* is the caller, while `Car` is the callee.

Figure 2.2 shows an excerpt of the Class-level Control Flow Graphs (CCFGs) for the two classes.

In the figure, the names of the nodes are labelled with the line number of the corresponding statements in the code of Listing 2.1. Node 16 in `Person.addEnergy()` is a call site to `Car.refuel()`; it is also control dependent on nodes 5 (`Person.driveToHome()`) and 13 (`Person.addEnergy()`). Furthermore, node 16 only post-dominates branch $\langle 13, 16\rangle$. Instead, the branch $\langle 5, 6c\rangle$ is not post-dominated by node 16 as covering $\langle 5, 6c\rangle$ does not always imply covering node 16 as well. Therefore, the branches in the caller `Person.addEnergy()` that always lead to the callee are $B_{\texttt{Person}}$ (`Car.refuel()`) $= \{\langle 13, 16\rangle\}$. Hence, among all branches in the caller class (`Person` in our example), we are interested in covering the branches that, when executed, always lead to the integration call site (i.e., calling the callee class). We refer to these branches as *target branches* for the caller.

Definition 2.1 (*Target branches for the caller*) *For a call site s in R, the set of target branches $B_R(s)$ for the caller R contains the branches having the following characteristics: (i) the branches are outgoing edges for the node on which s is control dependent (i.e., nodes for which s post-dominates one of its outgoing branches but does not post-dominate the node itself); and (ii) the branches are post-dominated by s (i.e., branches for which all the paths through the branch to the exit point pass trough s).*

Let us consider the example of Figure 2.2 again. This time, let us look at the branches in the callee (`Car`) that are directly related to the integration call. In the example, executing the method call `Car.refuel()` (node 16 of the method `Person.addEnergy()`) leads to the execution of the branching node $b8$ of the class `Car`. Hence, the set of branches affected by the interaction calls is $B_{\texttt{Car}}$(`Car.refu-el()`) $= \{\langle b8, b9\rangle; \langle b8, b10\rangle\}$. In the following, we refer to these branches as *target branches* for the callee. Note that, for a call site s in R calling E, the set of target branches for the callee does include branches that are trivially executed by any execution of s.

Definition 2.2 (*Target branches for the callee*) *The set of target branches $B_E(s)$ for the caller E contains branches satisfying the following properties: (i) the branches are among the outgoing branches of branching nodes (i.e., the nodes having more than one outgoing edge); and (ii) the branches are accessible from the entry node of the method called in s.*

Given the sets of target branches for both the caller and callee, an integration test case should exercise at least one target branch for the caller (branch affecting the integration call) and one target branch for the callee (i.e., the integration call should lead to cover branches in the callee). In the following, we define pairs of target branches $(b_r \in B_R(s), b_e \in B_E(s))$ as *coupled branches* because covering b_r can lead to cover b_e as well. In our example of Figure 2.2, we have two coupled branches: the branches $(\langle 9, 10c\rangle, \langle b8, b9\rangle)$ and the branches $(\langle 9, 10c\rangle, \langle b8, b10\rangle)$.

Definition 2.3 (*Coupled branches*) *Let $B_R(s)$ be the set of target branches in the caller class R; let $B_E(s)$ be the set of target branches in the callee class E; and let s be the call site in R to the methods of E. The set of coupled branches $CB_{R,E}(s)$ is the cartesian product of $B_R(s)$ and $B_E(s)$:*

$$CB_{R,E}(s) = CB_{R,E}(s) = B_R(s) \times B_E(s) \tag{2.1}$$

Definition 2.4 *Let $S = (s_1, \ldots, s_k)$ be the list of call sites from a caller R to a callee E, the set of coupled branches for R and E is the union of the coupled branches for the different call sites S: $CB_{R,E} = \cup_{s \in S} CB_{R,E}(s)$.*

Coupled Branches Criterion (CBC) Based on the definition above, the CBC criterion requires that for all the call sites S from a caller R to a callee E, a given test suite T covers all the coupled branches:

$$CBC_{R,E} = \frac{|\{(r_i, e_i) \in CB_{R,E} | \exists t \in T : t \; covers \; r_i \; and \; e_i\}|}{|CB_{R,E}|}$$

As for classical branch pair coverage, $CB_{R,E}$ may contain incompatible branch pairs. However, detecting and filtering those incompatible pairs is an undecidable problem.

Listing 2.2: Class GreenPerson

```
Class GreenPerson extends Person{
    private HybridCar car = new HybridCar();
    @override
    public void addEnergy(){
        if(this.lazy){
            takeBus();
        }else if (chargerAvailable()){
            car.recharge()
        }else{
            car.refuel();
        }
    }

    private void chargerAvailable(){
        if(ChargingStation.takeavailableStations().size > 0){
            return true;
        }
        return false;
    }
}
```

Inheritance and polymorphism In the special case where the caller and callee classes are in the same inheritance tree, we use a different procedure to build the CCFG of the super-class and find the call sites S. The CCFG of the super-class is built by merging the CFGs of the methods that are not overridden by the sub-class. As previously, the CCFG of the sub-class is built by merging the CFGs of the methods defined in this class, including the inherited methods overridden by the sub-class (other non-overridden inherited methods are not part of the CCFG of the sub-class).

For instance, the class `GreenPerson` in Listing 2.2, representing owners of hybrid cars, extends class `Person` from Listing 2.1. For adding energy, a green person can either refuel or recharge her car (lines 7 to 11). `GreenPerson` overrides the method `Person.addEnergy()` and defines an additional method `GreenPerson.chargerAvailable()` indicating if the charging station is available. Only those two methods are used in the CCFG of the class `GreenPerson` presented in Figure 2.3, inherited methods are not included in the CCFG; the CCFG of the super-class `Person` does not contain the method `Person.addEnergy()`, redefined by the sub-class `GreenPerson`.

The call sites S are identified according to the CCFGs, depending on the caller and the callee. If the caller R is the super-class, S will contain all the calls in R to methods that have been redefined by the sub-class. For instance, nodes 6 and 13 in Figure 2.2 with `Person` as caller. If the caller R is the sub-class, S will contain all the calls in R to methods that have been inherited but not redefined by R. For instance, node 6 in Figure 2.3 with `GreenPerson` as caller.

2.1.3 CLING

In this section, we describe CLING, the tool that we developed to generate integration-level test suites that maximize the proposed CBC adequacy criterion. The inputs of CLING are the (1) *application's bytecode*, (2) a *caller class* R, and (3) and *callee class* E. As presented in Figure 2.1, it first detects the covering methods (step ①) and identifies the coupled branches $CB_{R,E}(s)$ for the different call sites (step ②), before starting the search-based test case generation process (detailed in the following subsections). CLING produces a test suite that maximizes the CBC criterion for R and E.

Satisfying the CBC criterion is essentially a many-objective problem where integration-level test cases have to cover pairs of coupled branches separately. In other words, each pair of coupled branches

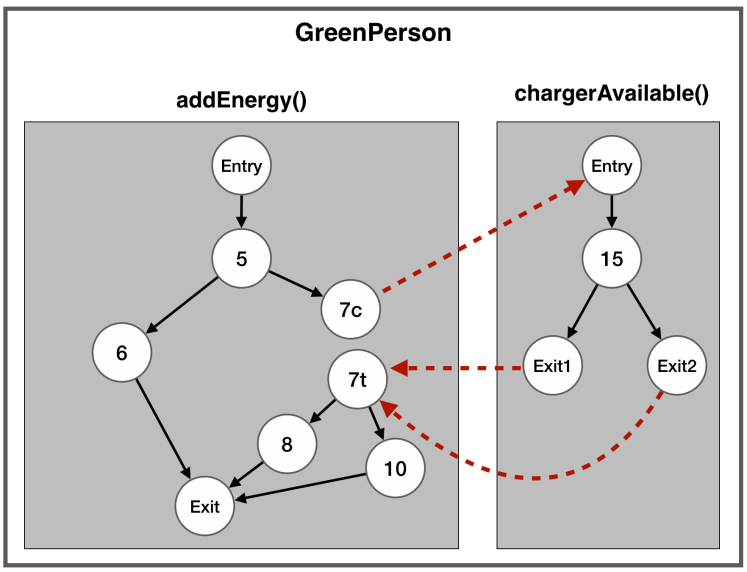


Figure 2.3: CCFG of $GreenPerson$ as subclass

corresponds to a search objective to optimize. The next subsection describes our search objectives.

Search objectives

In our approach, each objective function measures the distance of a generated test from covering one of the coupled branch pairs. The value ranges between $[0, +\infty)$ (zero denoting that the objective is satisfied). Assuming that $CB_{R,E} = \{c_1, c_2, \ldots, c_n\}$ is the set of coupled branches $< r_i, e_i >$ between R and E. Then, the fitness for a test case t is:

$$\begin{cases} d(c_1, t) = D(r_1, t) + D(e_1, t) \\ \ldots \\ d(c_n, t) = D(r_n, t) + D(e_n, t) \end{cases} \tag{2.2}$$

Where $D(b, t) = al(b, t) + bd(b, t)$ computes the distance between the test t to the branch b using the classical approach level ($al(b, t)$) and normalized branch distance ($bd(b, t)$) [69].

Test Case Generation

To solve such a many-objective problem, we tailored the Many-Objective Sorting Algorithm (MOSA) [78] to generate test cases trough class integration. MOSA has been introduced and assessed in the context of unit test generation [78] and security testing [54]. Besides, previous studies showed that MOSA is very competitive compared with alternative algorithms (e.g., [24, 78]) when handling hundreds and thousands of testing objectives. Interested readers can find more details about the original MOSA algorithm in [78]. Although a more efficient variant of MOSA has been recently proposed [81], such a variant (DynaMOSA) requires to have a hierarchy of dependencies between coverage targets that exists only at unit-level.

Therefore, in CLING, we tailored MOSA to work at integration level, targeting pairs of coupled branches rather than unit-level coverage targets (e.g., statements). In the following, we describe the main modifications we applied to MOSA to generate integration-level test cases.

Initial population

The search process starts by generating an initial population of test cases. A random test case is a sequence of statements (*objects instantiations*, *primitive statements*, *method calls*, and *constructors*) of variable lengths. More precisely, the random test cases include *method calls* and *constructors* for the caller R, which directly or indirectly invoke methods of the callee E (*covering methods*). Although CLING generates these test cases randomly, it extends the initialization procedure of Soltani *et al.* [99]. In particular, the initialization procedure in CLING gives a higher priority to methods in the caller class R that invoke methods of the callee class E. While calls to other methods of R are also inserted, their insertion has a lower probability. This prioritization ensures to generate tests covering call sites to the callee class. Instead, in the original MOSA algorithm, all methods of the CUT are inserted in each random test case with the same probability (no prioritization).

Mutation and crossover

CLING uses the traditional single-point crossover and mutation operators (adding, changing and removing statements) [36] with an additional procedure to repair broken chromosomes. The initial test cases are guaranteed to contain at least one *covering methods* (a method of R that invokes directly of indirectly methods of E). However, mutation and crossover can lead to generating *offspring* tests that do not include any *covering method*. We refer to these chromosomes as *broken chromosomes*. To fix the broken chromosomes, the *repairing procedure* works in two different ways, depending on whether the broken chromosome is created by the crossover or by the mutation.

If the broken chromosome is the result of the mutation operator, then the repairing procedure works as follows: let t be the broken chromosome and let M be the list of covering methods; then, CLING applies the mutation operator to t in an attempt to insert one of the covering methods in M. If the insertion is not successfully, then the mutation operator is invoked again within a loop. The loop terminates when either a covering method is successfully injected in t or when the number of unsuccessfully attempts is greater than $h = 50$. In the latter case, t is not inserted in the new population for the next generation.

If the broken chromosome is generated by the crossover operator then the broken child is replaced by one of its parents used by single-point crossover operator.

Polymorphism

If the caller and callee are in the same hierarchy and the caller is the super-class, CLING cannot generate tests for the caller class that will cover the callee class (since the methods to cover are not defined in the super-class). In this particular case, CLING generates tests for the callee class. However, it selects the covering methods only from the inherited methods which are not overridden by the callee (sub-class). A covering method should be able to cover calls to the methods that have been redefined by the sub-class. With this slight change, CLING can improve the CBC coverage, as described in Section 2.1.2.

2.1.4 Empirical evaluation

Our evaluation aims to answer the following research questions:

- **RQ1**: *To what extent is* CLING *able to achieve high Coupled Branch Coverage?*
- **RQ2**: *What is the effectiveness of the integration-level tests compared to unit-level tests?*
- **RQ3**: *What integration faults does* CLING *detect?*

Table 2.1: Projects in our empirical study. `#` indicates the number of caller-callee couples. `CC` indicates the cyclomatic complexity of the caller and callee classes. `Calls` indicates the number of calls from the caller to the callee. `Coupled branches` indicates the number of coupled branches.

Project	#	Caller		Callee		Calls		Coupled branches			
		$\overline{cc}$	σ	$\overline{cc}$	σ	count	σ	min	count	σ	max
closure	26	1,221.3	1,723.0	377.2	472.5	70.3	101.0	4	10,542	17,080	60,754
lang	18	145.0	177.8	235.3	242.7	12.4	14.6	2	409	598	1,826
math	25	79.2	88.4	57.5	64.4	18.8	34.5	2	294	613	2,682
mockito	20	115.3	114.4	127.8	113.2	39.5	64.9	0	1,185	1,974	6,929
time	51	68.7	84.0	87.2	92.3	23.9	50.5	0	494	1,093	5,457
Total	140	301.1	859.5	160.6	257.7	32.4	62.8	0	2,412	8,294	60,754

Implementation

We implemented CLING as an open-source tool written in Java. The tool implements the code instrumentation for pairs of classes, builds the CCFGs at the byte-code level, and derives the coverage targets (pairs of branches) according to the CBC criterion introduced in Section 2.1.2. The tool also implements the search heuristics, which are applied to compute the objective scores as described in Section 2.1.2. For the search algorithms, CLING re-uses the algorithms available in EVOSUITE [36], which is an external maven dependency. Besides, CLING implements the repair procedure described in Section 2.1.3, which extends the interface of the genetic operators in EVOSUITE. This allows us to re-use the original implementation of state-of-the-art search algorithms (e.g., WS [25], DynaMOSA [81], and MIO [12]) and customize it for our test case generation problem. More information about the user documentation and development status are available in Chapter 3.

Study Setup

Subjects Selection The subjects of our studies are five Java projects, namely *Closure compiler*, *Apache commons-lang*, *Apache commons-math*, *Mockito*, and *Joda-Time*. These projects have been used in prior studies to assess the coverage and the effectiveness of unit-level test case generation (e.g., [66, 81, 57, 92]), program repair (e.g., [94, 68]), fault localization (e.g., [84, 15]), and regression testing (e.g., [75, 65]).

To sample the classes under test, we first extract pairs of `caller` and `callee` classes (i.e., pairs with interaction calls) in each project. Then, we remove pairs that contain trivial classes, i.e., classes where the caller and callee methods have no decision point (i.e., with cyclomatic complexity equal to one). This is because methods with no decision points can be covered with single method calls at the unit testing level. Note that similar filtering based on code complexity has been used and recommended in the related literature [25, 71, 81]. From the remaining pairs, we sampled 140 classes from the five projects in total. We performed the sampling to have classes with a broad range of complexity and coupling. The most and least complex classes in the selected class pairs have 5,034 and one branching nodes, respectively. Also, the caller class of the most and least coupled class pairs contain 453 and one call sites to the callee class, respectively. The numbers of pairs selected from each project are reported in Table 2.1. Each pair of caller and callee classes represents a target for CLING.

Evaluation Procedure To answer **RQ1**, we ran CLING 20 times on each class pair. In each run, we collected the generated test suites and the corresponding number of pair branches covered by the suite. Each run was configured with a search budget of five minutes. Then, we analyzed the average CBC coverage achieved by CLING across the 20 independent runs.

For **RQ2**, we measure the effectiveness of the generated test suite using *mutation analysis* on

the callee classes (only). Mutation analysis is a high-end coverage criterion, and mutants are often used as substitutes for real faults since previous studies highlighted its significant correlation with fault-detection capability [58, 11]. Besides, mutation analysis provides a better measure of the test effectiveness compared to more traditional coverage criteria [113] (e.g., branch coverage).

For the mutation analysis, we used PIT[1], which is a state-of-the-art mutation testing tool for Java code, to mutate the callee classes. PIT has been used in literature to assess the effectiveness of test case generation tools [66, 71], and it has also been applied in industry[2]. In our study, we used PIT v. 1.4.9 with the default mutation operators: `Conditionals Boundary`, `Increments`, `Invert Negatives`, `Math`, `Negate Conditionals`, `Return Values`, and `Void Method Calls` operators.

To answer **RQ2**, we compute the mutation scores achieved by the test suite generated with CLING (T_{CLING}) for the callee class in each target class pair. Then, we compare it with the mutation scores achieved by the unit-level test suites produced by EVOSUITE when executed against the caller (T_R) and the callee classes (T_E) of the target pair. We choose EVOSUITE as the state-of-the-art unit test generation tool because it won the last editions of the unit testing tool competition [89, 60].

For each class pair targeted with CLING, we ran EVOSUITE on both the caller and the callee separately. This results in having two unit-level test suites, one for the caller (T_R), and one for the callee (T_E). To answer **RQ2**, we analyzed the orthogonality of the sets of mutants in the callee that were strongly killed by the integration-level tests (T_{CLING}), and those killed by the two unit-level tests (T_R and T_E) individually. In other words, we look at whether T_{CLING} allows to kill mutants that are not killed at unit level (strong mutation). To allow a fair comparison, EVOSUITE was run for five minutes (the same search budget used for CLING) on each caller/callee class. Furthermore, EVOSUITE was configured to use the branch coverage criterion, using DynaMOSA as the search algorithm. To address the random nature of DynaMOSA, EVOSUITE was launched 20 times on each class.

For **RQ3**, we collected and analyzed the exceptions triggered by both integration and unit-level test suites. In particular, we extracted unexpected exceptions, *i.e.,* exceptions that are triggered by the test suites but that are not handled by the SUT (caller and callee) causing application crashes. Unhandled exceptions are not declared in the signature of the caller and callee methods, not caught with `try-catch` blocks, `throws` clauses, and not documented in the `Javadoc`. To answer **RQ3**, we then manually analyze those unexpected exceptions that are triggered by the integration-level test cases (i.e., by CLING), but not by the unit-level tests.

Flaky tests. The test suites generated by CLING and EVOSUITE may contain flaky tests, *i.e.,* test cases that exhibit intermittent failures if executed with the same configuration. To detect and remove flaky tests, we ran each generated test suite five times. Then, we removed those tests that fail in at least one of the independent runs. Therefore, the test suites used to answer our three research questions likely do not contain flaky tests.

Infrastructure We used a cluster (with 20 CPU-cores, 384 GB memory, and 482 GB hard drive) for our evaluation. We executed CLING and EVOSUITE (against caller and callee class) on each of the 140 detected subjects. To address the random nature of the evaluated search approaches, we repeated each execution 20 times. In total, we performed 8,400 independent executions.

2.1.5 Evaluation Results

This section presents the results of the evaluation and answers the research questions. To ease readability, we report our results at the project level and provide examples at the class level.

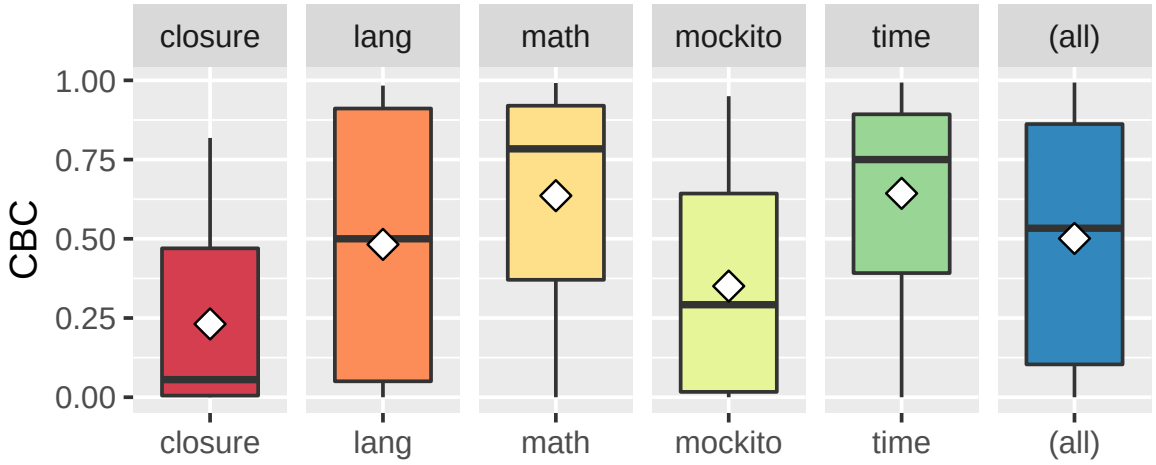


Figure 2.4: Coupled branches coverage of T_{Cling}.

Coupled branches coverage (RQ1)

Figure 2.4 presents the coupled branch coverage of the T_{Cling} test suites. On average (the diamons in Figure 2.4), the test suites generated by CLING achieve a coupled branch coverage of 50.10% for all the projects. The most covered couples are in the `time` project ($\overline{CBC} = 64.34\%$), followed by `math` ($\overline{CBC} = 63.60\%$) and `lang` ($\overline{CBC} = 48.23\%$). The least covered couples are in the `closure` ($\overline{CBC} = 23.12\%$) and `mockito` projects ($\overline{CBC} = 37.07\%$), which are also the projects with the highest number of coupled branches in Table 2.1 (10,542 coupled branches on average for all the class pairs in `closure` and 1,185 coupled branches on average in `mockito`).

As reported in Table 2.1, CLING did not identify any coupled branches for four pairs of classes (one in `mockito` and three in `time`). This is due to the absence of target branches in either the caller or the callee. Those four couples have been excluded from the results presented in Figure 2.4.

In total, CLING could generate at least one test suite achieving a coupled branches coverage higher than 50% for 81 out of 140 pairs: 37 for `time`, 18 for `math`, 10 for `lang`, 9 for `closure`, and 7 for `mockito`. For 23 caller-callee pairs, CLING could not generate a test suite able to cover at least one coupled branch out of 20 executions: 7 for `closure`, 4 for `math`, `mockito` and `time`, and 2 for `lang`. Those 23 pairs cannot be explained solely by the complexities of the caller (with a cyclomatic complexity ranging from 8 to 5,034 for those classes) and the callee (with a cyclomatic complexity ranging from 1 to 2,186) or the number of call sites (ranging from 1 to 177) and call for a deeper understanding of the interactions between caller and callee around the call sites. In our future work, we plan to refine the caller-callee pair selection used in our evaluation protocol (for which we looked at the global complexity of the classes) to investigate the local complexity of the classes around the call sites.

Summary For 81 out of 140 (58%) of the pairs, CLING can generate test suites achieving a coupled branches coverage above 50%. For 23 pairs out of 140 (16.4%) no coupled branch pairs were covered by the generated tests. For the remaining 36 pairs (25.6%), CLING was able to generate test suites covering coupled branches, but none could achieve a CBC higher than 50%.

[1] http://pitest.org
[2] http://pitest.org/sky_experience/

Table 2.2: Status (for T_R and T_E) of the mutants killed only by T_{Cling}. `Not-covered` denotes that the mutant is not covered by any test case and `survived` denotes that the mutant is covered but not killed.

Test Suite	closure		lang		math	
	not-covered	survived	not-covered	survived	not-covered	survived
T_E	5,128 (0.01)	1,596 (0.00)	4,505 (0.01)	600 (0.00)	12,655 (0.07)	2,626 (0.01)
T_R	3,799 (0.00)	1,651 (0.00)	2,652 (0.00)	1,528 (0.00)	11,482 (0.07)	4,541 (0.03)
	mockito		**time**			
	not-covered	survived	not-covered	survived		
T_E	10,188 (0.06)	4,468 (0.03)	33,436 (0.08)	7,003 (0.02)		
T_R	9,626 (0.06)	5,865 (0.03)	16,979 (0.04)	13,877 (0.03)		

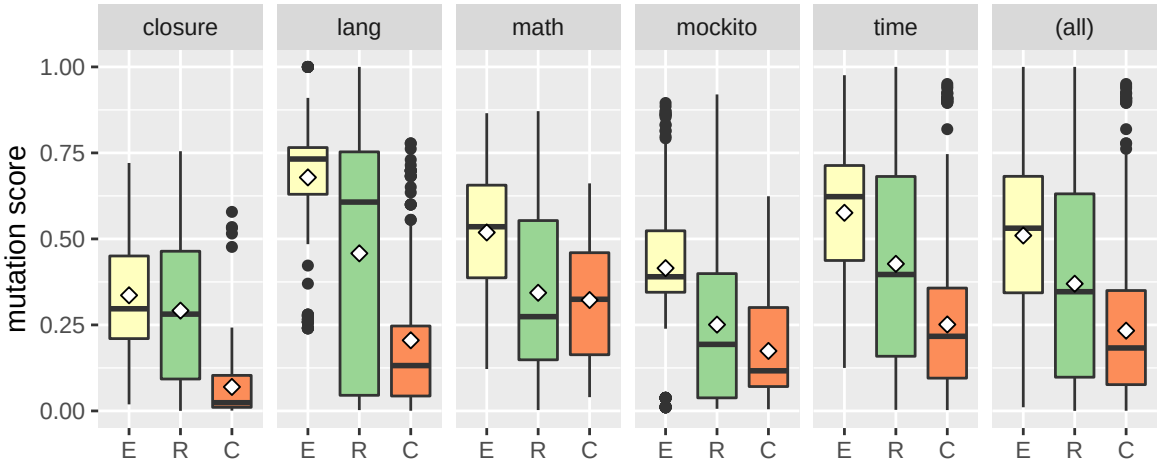


Figure 2.5: Mutation coverage for T_E, T_R, and T_{CLING} when mutating E.

Comparison of Mutation Coverage (RQ2)

To understand the impact of CLING on mutation coverage, we first show the overall mutation scores, in Figure 2.5. For each of our subject systems, we show the mutation scores when we mutate class E, and apply the test suite T_E, T_R, and T_{CLING}.

As expected, test suites optimized for overall branch (line) coverage (T_E), achieve a total higher mutation score (50.49% on average), simply because a mutant that is on a line that is never executed cannot be killed. Thus, the yellow (T_E) bars score highest in Figure 2.5. Likewise, the orange (T_{CLING}) bars are lowest (20.46% on average), since CLING searches for dedicated interaction pairs, but does not try to optimize overall line coverage. Note that for `lang` it is the "easiest" to kill mutants, and that this is the hardest for the complex `closure` project.

Figure 2.5 also shows that almost half of the mutants are *not* killed by unit test suites T_E and T_R. It is those unkilled mutants that are the target of CLING. Thus, Figure 2.6 shows increase in the percentage of the total number of mutants strongly killed by T_{CLING}, compared to T_E, T_R, or their union T_{E+R}. The main findings are:

1. On average, 11.80% of the mutants are killed only by T_{CLING}, compared to T_E, the unit test suite optimized for E itself.
2. This difference becomes slightly less, 9.21%, if we use T_R, the unit test suite exercising E via the caller class R. This is natural, since both T_R and T_{CLING} seek to exercise E via R.
3. The difference with traditional unit testing is 6.81% when we compare CLING with the combined test suites of E and R, exercising E directly as much as possible as well as indirectly via

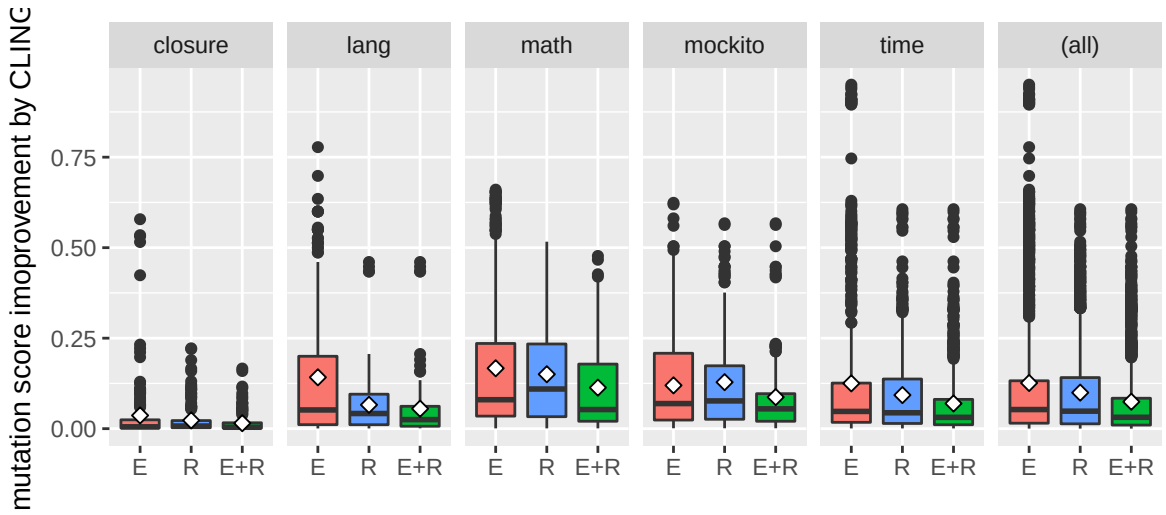


Figure 2.6: Increases in mutation coverage of T_{CLING} compared to T_E, T_R and T_{E+R}

Listing 2.3: Exception captured only by CLING in Math

```
java.lang.ArithmeticException: / by zero
  at org.apache.commons.math.fraction.ProperFractionFormat.format(
      ProperFractionFormat.java:95)
  at org.apache.commons.math.fraction.FractionFormat.format(FractionFormat.java
      :206)
  at java.base/java.text.Format.format(Format.java:158)
  at java.base/java.text.Format.formatToCharacterIterator(Format.java:207)
```

call sites in R.

Note that the success of CLING is related to the level of CBC achieved for the subject systems, as shown in Figure 2.4 when answering RQ1. Higher CBC levels help to exercise a larger amount of different behaviors, which in turn help to kill mutants.

The outliers in Figure 2.6 are also of interest: Out of the 140 classes, there are 24 for which CLING was able to generate a test suite where more than half of the mutants were killed *only* by T_{CLING}, compared to T_E. Even when compared to T_{E+R}, there were three classes for which this was the case, further emphasizing the complementarity between unit and integration testing.

Table 2.2 presents the status of the mutants that are killed by T_{CLING} but not by unit-level test cases. What stands out is that many mutants are in fact covered, but not killed by T_E or T_R. Here CLING leverages the context of caller, not only to reach a mutant, but also to *propagate* the (modified) values inside the caller's context, so that the mutants can be eventually killed.

Summary The test suite generated by CLING for a caller R and callee E, can kill *different* mutants than unit test suites for E, R or their union, increasing the mutation coverage on average with 11.81%, 9.21%, and 6.81%, respectively, with outliers well above 50%. Our analysis indicates that this is not just due to the fact that different mutants are *reached*, but also because the mutated outcomes are better *propagated* in the caller context, causing the mutant to be killed by CLING.

Listing 2.4: CLING test case triggering the crash in Listing 2.3

```
public void testFraction() {
    ChoiceFormat cf = new ChoiceFormat("");
    ProperFractionFormat pf = new ProperFractionFormat(cf);

    ParsePosition ps = new ParsePosition(0);
    Object obj = cf.parseObject("", ps);

    pf.formatToCharacterIterator(obj);
}
```

New Bugs Found Using CLING (RQ3)

In our experiments, CLING generated 29 test cases that triggered an unexpected exception in one of the subject systems. These remained undetected by any of the corresponding test cases generated by the 40 executions of EVOSUITE (20 for each caller, and 20 for each callee class, for the pairs tested by CLING). Of the 29 crash-inducing test cases, 21 were found in *Closure*, four in *Time*, and another four in *Math*. For six of the 29 integrations tested, the caller and callee belonged to the same class hierarchy (shared a superclass other than Object).

To get an intuition for the type of problem detected by CLING, consider the test case it generated in Listing 2.4 and the induced stack trace (for a division by zero) in Listing 2.3. These are from the Commons Math `Fraction` class, which can be used to represent fractions like $1/3$ or $3/4$. There are several ways to *format* fractions, represented by the `FractionFormat` class, and its subclass `ProperFractionFormat`.

The various classes and methods involved in fractions make assumptions about denominators being zero or not; one class assumed an invariant that the denominator can never be zero. This was indeed ensured by most constructors, but unfortunately not by all. The CLING integration testing approach brought these conflicting assumptions together, triggering the stack trace of Listing 2.3.

The method pair under test by CLING in this case is ⟨`FractionFormat.format`, `ProperFractionFormat.format`⟩, which indeed recurs on lines 3 and 2 of the stack trace (Listing 2.3). The test case obtained by CLING implicitly creates a fraction by parsing an empty string. Under the hood this leads to the creation of a fraction from `Double.NaN`. Fractions can be created from any double, and the `Fraction` class then finds the corresponding minimal nominator and denominator (e.g., 3 and 4 for 0.75). For `Double.NaN` this approximation leads to a denominator that is zero. At the same time, the `format` method seeks to display the fraction value, and therefore computes the actual division, triggering the failure.

As is typical for integration faults, this problem can be fixed in multiple ways. The most consistent would be to adjust the `Fraction(double)` constructor, to align it with all other constructors (which raise an exception if the denominator is zero). This then would ensure the invariant that the denominator is never zero, aligning the assumptions of all classes and methods involved.

We are in the process of conducting a root cause analysis for all 29 issues. We have not yet contacted the maintainers of the open source subject systems to avoid exposing ourselves in light of double blind reviewing, but will offer findings (and proposed fixes) at a later stage.

Summary CLING-based automated testing of ⟨caller, callee⟩ class pairs exposes actual problems that are not found by unit testing either the caller or callee class individually. These problems relate to conflicting assumptions on the safe use of methods across classes.

2.1.6 Discussion

Test generation cost One of the challenges in automated class integration testing is detecting the integration points between classes in SUT. The number of code elements (e.g., branches) that are related to the integration points increases with the complexity of the involved classes. Finding and testing a high number of integration code targets increases the time budget that we need for generating integration-level testing.

With CBC, the number of coupled branches to exercise is upper bounded to the cartesian product between the branches in the caller R and the callee E. Let B_R be the set of branches in R and B_E the set of branches in E, the maximum number of coupled branches $CB_{R,E}$ is $B_R \times B_E$. In practice, the size of $CB_{R,E}$ is much smaller than the upper bound as the targets branches in the caller and callee are subsets of R and E, respectively. Besides, CBC is defined for pairs of classes and not for multiple classes together. This substantially reduces the number of targets we would incur when considering more than two classes at the same time.

Effectiveness To answer **RQ2**, we analyzed the set of mutants that are killed by CLING (integration tests) but not by the two unit-test suites generated by EVOSUITE for the caller and callee separately (boxes labeled with $C - E - R$ in Figure 2.6). Note the test suite C was generated by CLING using a search budget of five minutes. Similarly, the unit-level suites E and R by EVOSUITE were generate with a search budget of five minutes for each class separately. Therefore, the total search budgets for unit test generation ($E + R$) is 10 minutes. Despite the larger search budget spent on unit testing, there are still mutants and faults detectable only by CLING and in less time.

Note that CLING is not an alternative tool to unit testing tools like EVOSUITE. In fact, integration test suites do not subsume unit-level suites as the two types of suites focus on different aspects of the SUT. Our results (**RQ2**) confirm that integration and unit testing are complementary. Indeed, some mutants can be killed exclusively by unit-test suites: the overall mutation scores for the unit tests E, and R are larger than the overall mutation scores of CLING. This higher mutation score is expected due to the larger unit-level branch coverage achieved by the unit tests (coverage is a necessity but not a sufficient condition to kill mutant).

Instead, CLING focuses on a subset of the branches in the units (caller and callee) but exercises the integration between them more extensively. In other words, the search is less broad (few branches) but more in-depth (the same branches are covered multiple times within different pairs of coupled branches). This more in-depth search allows killing mutants that could not be detected by satisfying unit-level criteria. Our results further indicate that it also allows us finding bugs that are not detectable by unit tests.

Applicability CLING considers pairs of classes and exercises the integration between them. We did not propose any procedure for selecting pairs of classes to give in input to CLING. However, CLING can be applied to any pair of classes in which at least one of the classes calls the other one. Besides, our approach can be further extended by incorporating integration test ordering approaches and selecting the classes to integrate with a given ordering.

In this deliverable, we consider only the integration call type of integration between classes, although other types of integration exist between classes [76] (i.e., integration through external data). However, our results are very encouraging because they show *how integration-level tests based on CBC coverage complement unit-level tests generated with* EVOSUITE *in terms of test effectiveness*. Further research is needed to incorporate other types of integrations in CLING. This is part of our future agenda.

2.1.7 Threats to validity

Internal validity Our implementation of CLING may contain bugs. We mitigated this threat by reusing standard algorithms implemented in EVOSUITE, a widely used state-of-the-art unit test gen-

eration tool. And by unit testing the different extensions (described in Section 2.1.4) we developed. To take the randomness of the search process into account, we followed the guidelines of the related literature [13] and executed CLING and EVOSUITE 20 times to generate the different test suites (T_{CLING}, T_E, and T_R) for the 140 caller-callee classes pairs. We described how we parametrized CLING and EVOSUITE in Sections 2.1.3 and 2.1.4. We left all other parameters to their default value, suggested by related literature [14, 78, 93].

External validity We acknowledge that we report our results for only five open-source projects. However, we recall here their diversity and broad adoption by the software engineering community. The identification and categorization of the integration faults done in **RQ3** have been performed by the first author and confirmed independently by the last author of the deliverable. Disagreements were solved by a discussion between the two authors.

Reproducibility We provide CLING as an open-source publicly available tool as the data and the processing scrips used to present the results of this deliverable.[3] Including the subjects of our evaluation (inputs) and the produced test cases (outputs).

2.2 Search-based crash reproduction

The previous versions of Botsing use a fitness function that relies on the instrumentation of the target class only (see Chapter 2 of D3.3). Using the extension of the instrumentation, developed for class integration testing (see Section 2.1, we can now instrument more than one class appearing in the stack trace. This instrumentation allows to redefine the fitness function to improve the guidance of the search-based crash reproduction process.

2.2.1 Background

The initial fitness function from crash reproduction in Botsing is defines as [98]:

$$f(t) = \begin{cases} 3 \times d_s(t) + 2 \times max(d_{except}) + max(d_{trace}) & \textit{if the line is not reached} \\ 3 \times min(d_s) + 2 \times d_{except}(t) + max(d_{trace}) & \textit{if the line is reached} \\ 3 \times min(d_s) + 2 \times min(d_{except}) + d_{trace}(t) & \textit{if the exception is thrown} \end{cases} \tag{2.3}$$

where $d_s(t) \in [0, 1]$ denotes how far t is from executing the target statement using two well-known heuristics, *approach level* and *branch distance* [97]. The approach level measures the minimum number of control dependencies between the path of the code executed by t and the target statement s. The branch distance scores how close t is to satisfying the branch condition for the branch on which the target statement is directly control dependent [69]. In Equation 2.3,

In D3.3, applying multi-objectivization was straightforward as the fitness function in Equation 2.3 is defined as the weighted sum of three components. Therefore, our multi-objectivized version of the crash replication problem consists of optimizing the following three objectives:

$$\begin{cases} f_1(t) = d_s(t) \\ f_2(t) = d_{except}(t) \\ f_3(t) = d_{trace}(t) \end{cases} \tag{2.4}$$

Fitness functions defined in equations 2.3 and 2.4 consider only the target class and provide little guidance (other than checking that the exception is thrown) once the target line is reached. For instance, by setting the target frame to the third frame in Listing 2.5, the $d_s(t)$ component of the fitness function will consider the approach level and branch distance to reach the line 614 in the

[3] Link is omitted due to double-blind (as per ICSE policy).

Listing 2.5: Stack trace of the XWIKI-13372 crash

```
java.lang.NullPointerException: null
  at com[...]BaseProperty.equals([...]:96)
  at com[...]BaseStringProperty.equals([...]:57)
  at com[...]BaseCollection.equals([...]:614)
  at com[...]BaseObject.equals([...]:235)
  at com[...]XWikiDocument.equalsData([...]:4195)
  [...]
```

`BaseCollection` class but will (unlike other approaches like RECORE [88]) not provide any guidance to reach the lines specified in the lower frames.

2.2.2 Using integration testing for crash reproduction

As explained in Section 2.1.2, the extension of the class instrumentation mechanisms allows to build CCFGs for multiple classes. In the case of search-based crash reproduction, this means that we can now define a fitness function that uses the approach level and branch distance on multiple classes. We define a new fitness function where the $d_s(t)$ component is refined such that it takes multiple frames into account:

$$f(t) = \begin{cases} d_{s_n}(t) + d_{s_{n-1}}(t) + \ldots + d_{s_1}(t) + max(d_{trace}) & \textit{if the line of the deepest frame is not reached} \\ d_{trace}(t) & \textit{otherwise} \end{cases} \quad (2.5)$$

Where $d_{s_n}(t) \in [0, 1]$ denotes how far t is from executing the target statement in frame at frame level n using the approach level and branch distance. Additionally, a statement in frame $k-1$ can only be reached if the statement at frame k has already been reached. For a target frame n, we have:

$$\forall k \in [1; n] : d_{s_k}(t) > 0 \Rightarrow d_{s_{k-1}}(t) = 1$$

2.2.3 Empirical evaluation

Out future work include a full empirical evaluation of this new fitness function on JCrashPack (defined in D3.2) and a comparison with fitness functions defined in equations 2.3 and 2.4. This evaluation will follow the standard protocol defined in our previous work [95].

Chapter 3

The Botsing framework

Botsing is a framework for Java crash reproduction and test generation using runtime information of the software under test, developed in the context of the STAMP project.

3.1 Developer documentation

3.1.1 Maven modules

Botsing is designed as a framework for crash reproduction and test case generation. The Maven projects is organized as follows:

- `botsing`, the parent module that contains only a `pom.xml` file with the configuration common to all the modules:
  - `botsing-reproduction`, the reproduction engine (see D3.3).
  - `botsing-preprocessing`, a tool to preprocess and clean stack traces before calling the reproduction engine (see D 3.2).
  - `botsing-maven`, the Botsing Maven plugin (see WP4).
  - `botsing-examples`, code examples that could be reused to try Botsing and that are also used for Botsing integration tests.
  - `botsing-commons`, containing Java classes common to multiple sub-modules.
  - `botsing-model-generation`, containing the model generation engine used in model seeding (see Chapter 1).
  - `botsing-parsers`, containing utility classes to parse a stack trace.

3.1.2 Architecture

Figure 3.1 presents the packages of the crash reproduction engine (`botsing-reproduction`). The main packages are the `eu.stamp.botsing.ga` and `eu.stamp.botsing.fitness-function` packages. They contain the classes used to execute the guided genetic algorithm and compute the fitness function, both presented in Chapter 2.1.1.

Figure 3.2 details the classes implementing the guided genetic algorithm. A `GeneticAlgorithm` object uses a `TestFitnessFunction` to drive the test case generation process. In Botsing, the `GuidedGeneticAlgorithm` uses a `WeightedSum` fitness function and `GuidedSinglePointCrossover` and `GuidedMutation` operators to perform the crash reproduction search.

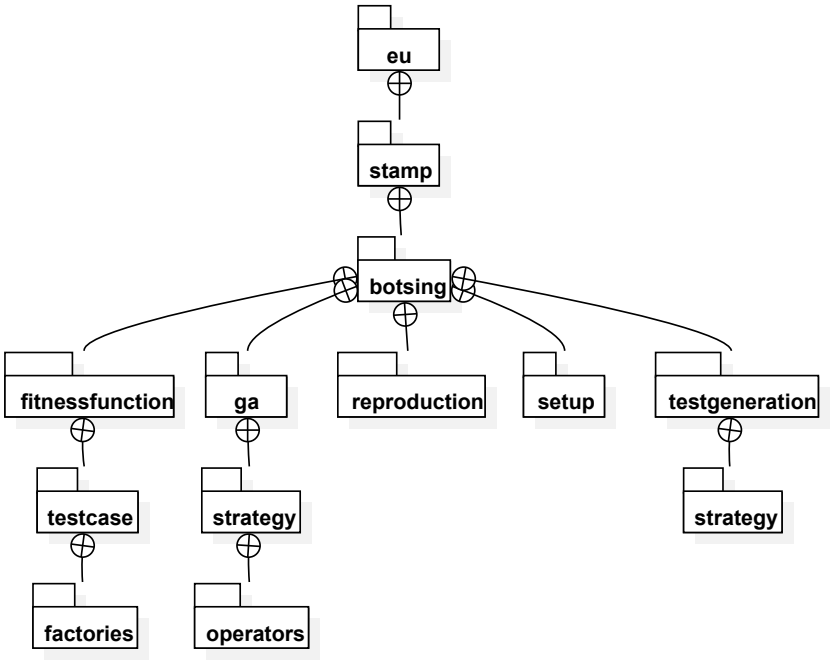


Figure 3.1: Package hierarchy of `botsing-reproduction`

The guided genetic algorithm is implemented in the `GuidedGeneticAlgorithm` class by the following method:

```
@Override
public void generateSolution() {
  currentIteration = 0;

  // generate initial population
  initializePopulation();

  LOG.debug("Starting evolution");
  int starvationCounter = 0;
  double bestFitness = getBestFitness();
  double lastBestFitness = bestFitness;

  LOG.debug("Best fitness in the initial population is: {}", bestFitness);
  while (!isFinished()){
    // Create next generation
    evolve();
    sortPopulation();

    bestFitness = getBestFitness();
    LOG.debug("New fitness is: {}", bestFitness);

    // Check for starvation
    if (Double.compare(bestFitness, lastBestFitness) == 0) {
      starvationCounter++;
    } else {
      LOG.debug("Reset starvationCounter after {} iterations", starvationCounter);
```

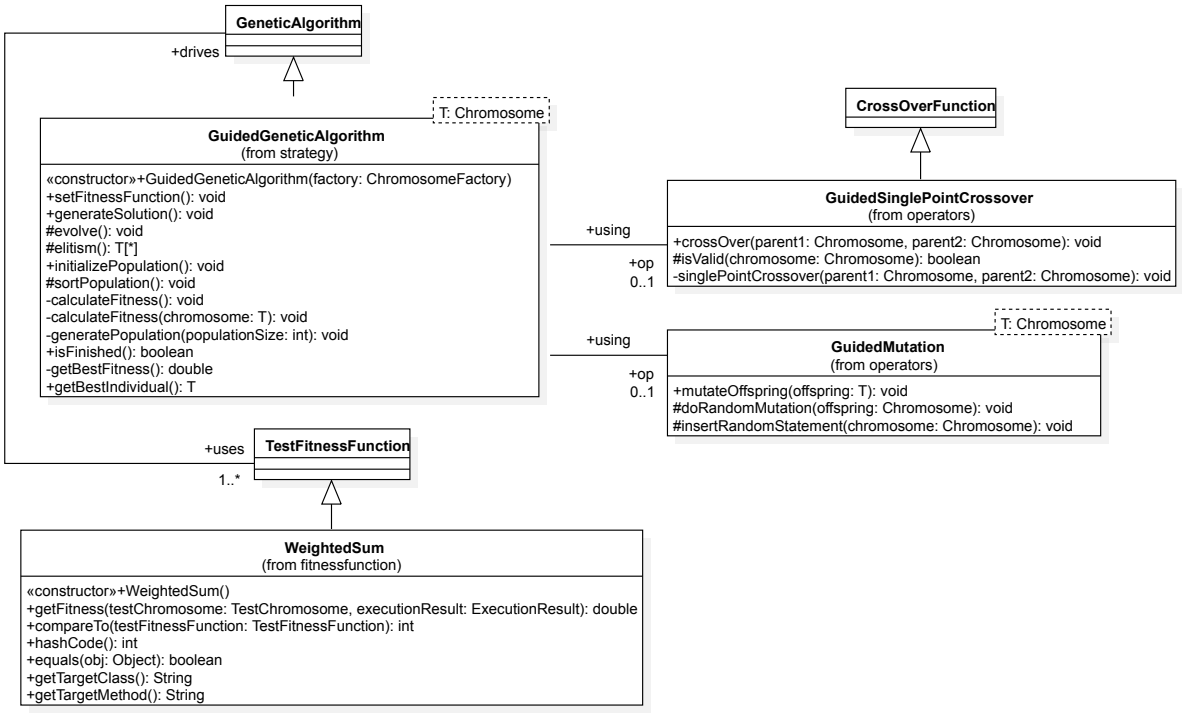


Figure 3.2: Core classes of `botsing-reproduction`

```
27          starvationCounter = 0;
            lastBestFitness = bestFitness;
29        }
          updateSecondaryCriterion(starvationCounter);
31
          LOG.debug("Current iteration: {}", currentIteration);
33        this.notifyIteration();
        }
35      LOG.debug("Best fitness in the final population is: {}", lastBestFitness);
    }
```

The population is initialized at line 6. The algorithm then loops until the fitness is 0.0 or until the budget is exhausted. The next generation is created at lines 16 and 17. The algorithm has an additional check to prevent starvation at line 23, before starting the next iteration.

3.1.3 Building Botsing

Currently, Botsing using a customized version of the EvoSuite-client. Hence, the building process contains two steps: 1) installing the customized version of EvoSuite-client:

```
mvn install:install-file -Dfile=botsing-reproduction/evosuite-client-botsing
    -1.0.7.jar -DgroupId=org.evosuite -DartifactId=evosuite-client-botsing -
    Dversion=1.0.7 -Dpackaging=jar
```

And, 2) build the Botsing project:

```
mvn package
```

3.1.4 Coding style

The coding style is described in a `checkstyle.xml` file available at the root of the project. The command `mvn checkstyle:check` must succeed for any pull request to be accepted.

3.1.5 Adding dependencies

The Botsing reproduction engine (`botsing-reproduction`) relies on EvoSuite to instrument the Java code during the evolutionary search. EvoSuite and other dependencies are managed at the module level. Each module declares a list of Maven dependencies, if you want to add one, simply add it to the list. However, dependency version must be declared as a property in the parent `pom.xml` file using the following syntax:

```
<properties>
  ...
  <!-- Dependencies versions -->
  <depdencendy-artifactId.version>1.1.1</depdencendy-artifactId.version>
  ...
</properties>
```

And referenced in the dependencies of the module using the following syntax:

```
<dependencies>
  <dependency>
    <groupId>com.groupId</groupId>
    <artifactId>depdencendy-artifactId</artifactId>
    <version>${depdencendy-artifactId.version}</version>
  </dependency>
</dependencies>
```

Please check in the list of properties that the dependency version is not already there before adding a new one.

3.1.6 License

Botsing is available under a business friendly license: *Apache-2.0*.

3.2 User documentation

We provide hereafter the user documentation to use the Botsing crash reproduction engine with the command line interface, the Maven plugin, and the Eclipse IDE plugin.

3.2.1 Crash reproduction command line interface

The latest version of Botsing crash reproduction engine with a command line interface (`botsing-reproduction-X-X-X.jar`) is available at https://github.com/STAMP-project/botsing/releases. The interface has three mandatory parameters:

- `-crash_log` the file with the stack trace. The stack trace should be clean (no error message) and cannot contain any nested exceptions.
- `-target_frame` the target frame to reproduce. This number should be between 1 and the number of frames in the stack trace.
- `-projectCP` the classpath of the project and all its dependencies. The classpath can be a folder containing all the .jar files required to run the software under test.

Additional parameters can be set. By default, the engine uses the following parameter values:

- `-Dsearch_budget=1800`, a time budget of 30 min. This value can be modified by specifying an additional parameter in format `-Dsearch_budget=60` (here, for 60 seconds).
- `-Dpopulation=100`, a default population with 100 individuals. This value may be modified using `-Dpopulation=10` (here, for 10 individuals).
- `-Dtest_dir=crash-reproduction-tests`, the output directory where the tests will be created (if any test is generated). This value may be modified using `-Dtest_dir=new-outputdir`.

To check the list of options, use `java -jar botsing-reproduction.jar -help`:

```
$ java -jar botsing-reproduction.jar -help
usage: java -jar botsing-reproduction.jar -crash_log stacktrace.log -target_frame
    2 -projectCP dep1.jar;dep2.jar
 -crash_log <arg>       File with the stack trace
 -D <property=value>    use value for given property
 -help                  Prints this help message.
 -projectCP <arg>       classpath of the project under test and all its
                        dependencies
 -target_frame <arg>    Level of the target frame
```

Example

```
java -jar botsing-reproduction.jar -crash_log LANG-1b.log -target_frame 2 -
    projectCP ~/bin
```

3.2.2 Stack trace preprocessing command line interface

The latest version of Botsing preprocessing command line tool (`botsing-preprocessing-X-X-X.jar`) is available at https://github.com/STAMP-project/botsing/releases.

Botsing preprocessing has these mandatory parameters (key/value):

- `-i` represents the input file path (`crash_log`) with the stack trace to clean. For example `-i=path-name-of-crash-log`. `-o` represents the output file path (`output_log`) cleaned of the error message and/or nested exceptions. For example `-o=path-name-of-output-log`.

The actions to perform (clean) in the input log file are:

- `-f` to flatten the stack trace. This action needs to use `-p` parameter (key/value) to set the package has frames pointing to the software under test (as `regexp`). For example `-p=my.package.*`.
- `-e` to remove the error message.

Example To clean the nested stack trace:

```
java -jar botsing-preprocessing.jar -i=crash_log.txt -o=output_log.log -f -p=com.
    example.*
```

Or to remove the error message in the log file:

```
java -jar botsing-preprocessing.jar -e -i=crash_log.txt -o==output_log.log
```

Note that you can use also both actions (`-f` and `-e`).

3.2.3 Model seeding command line interface

Behavioral models represent the usages of the different objects involved in the project. To learn (i.e., generate) behavioral models for an application, one can use `botsing-model-generation` that (i) statically analyses the source code of the project and (ii) executes the test cases to capture usages of the objects.

The latest version of Botsing model generation in command line (`botsing-model-generation-X-X-X.jar`) is available at https://github.com/STAMP-project/botsing/releases. Usage:

```
java -d64 -Xmx10000m -jar bin/botsing-model-generation-X.X.X.jar \
  -project_cp $CLASSPATH \
  -project_prefix "my.package" \
  -out_dir "results/my-project"
```

Where

- `-project_cp $CLASSPATH` provides the class path;
- `-project_prefix "my.package"` tells Botsing to run the test cases that are in the package my.package or one of its sub-packages; and
- `-out_dir "results/"` specifies the output directory for the models (models are generated in results/models).

Since model generation can consume a lot of memory, we strongly recommend to use the options `-d64 -Xmx10000m` when executing the tool.

A compete tutorial on how to use model seeding for crash reproduction is available at https://github.com/STAMP-project/botsing-demo.

Crash reproduction using model seeding

After model inference, the reproduction engine needs to access the directory where models have been generated, specified by the parameter `-model`. Additionally, it requires the probability of usage of model seeded object behaviors during the search, set using `-Dp_object_pool`.

Example

```
java -jar botsing-reproduction-1.0.7.jar -project_cp applications/LANG-9b -
    crash_log crashes/LANG-9b.log -targ
```

Test case generation using model seeding

The generation of new test cases using model seeding is performed by our extended version of EvoSuite.[1] EvoSuite is a unit test generator for Java applications. It implements several evolutionary algorithms and benefits from many advances over the years. One of the key functionalities of EvoSuite when generating test for a Class Under Test (CUT) is to be able to generate and initialize complex objects used by the CUT. This generation however is random or based solely on the existing test cases by carving (i.e., copy-pasting) objects and methods called on those objects from the existing test cases. With model seeding, we seek to enhance this functionality by using the models learned from the source code and the existing tests.

We can run EvoSuite using the following (quite long) command line:

```
java -d64 -Xmx4000m -jar bin/evosuite-master-1.0.7-SNAPSHOT.jar \
  -class "my.package.MyClass" \
  -projectCP "$myapp_classpath" \
  -generateMOSuite \
  -Dalgorithm=DynaMOSA \
  -Dsearch_budget=60 \
  -Dseed_clone="0.5" \
  -Donline_model_seeding=TRUE \
  -Dmodel_path="results/myapp-core/models" \
  -Dtest_dir="results/myapp-core/evosuite-tests" \
  -Dreport_dir="results/myapp-core/evosuite-report" \
  -Dno_runtime_dependency=true
```

Where

- `-class "my.package.MyClass"` defines the class under test for which test cases will be generated. It is also possible to generate a test suite for a whole package using the `-prefix "my.package"` option;
- `-projectCP "$myapp_classpath"` provides the class path of the software under test (to be set by using the `export` bash command);
- `-generateMOSuite` and `-Dalgorithm=DynaMOSA` indicate to EvoSuite to use a multi-objectives evolutionary strategy witht the DynaMOSA algorithm;
- `-Dsearch_budget=60` indicates the search budget in seconds allocated to EvoSuite to generate tests;
- `-Dseed_clone="0.5"` is the probability to use objects generated from the models. In our case, there is one chance out of two when an object is required to generate it from the models or to randomly generate a new one during the search;
- `-Donline_model_seeding=TRUE` indicates to generate objects from the models during the search. If this option is set to false, EvoSuite will initialize a pool of objects (by generating them from the models) during the initialization phase and pick objects only from that pool during the search. This option should be set to FALSE only if there is a large amount of objects required during the search when generating test cases for a given class;
- `-Dmodel_path="results/myapp-core/models"` indicates the path where to find the models. Each file should be named after the class it models;

[1] Available for now at https://github.com/STAMP-project/evosuite-model-seeding-tutorial/blob/master/bin/evosuite-master-1.0.7-SNAPSHOT.jar.

- `-Dtest_dir="results/myapp-core/evosuite-tests"` indicates where EvoSuite will generate the test cases;
- `-Dreport_dir="results/myapp-core/evosuite-report"` indicates where EvoSuite will generate the report on the generated tests;
- `-Dno_runtime_dependency=true` indicates to EvoSuite to generate plain JUnit test without a dependency to `org.evosuite:evosuite-standalone-runtime`. By default, EvoSuite generates JUnit tests relying on the EvoSuite framework to, among other things, decrease the risk of flakiness.

The generated tests are available in `results/myapp-core/evosuite-tests` and can be reviewed to improve or correct the oracle (mainly, the assertions) before being added to the test suite of the project.

A compete tutorial on how to use model seeding for test case generation is available at https://github.com/STAMP-project/evosuite-model-seeding-tutorial.

3.2.4 Botsing parallel processing

Botsing parallel allows to run an arbitrary number of parallel instances of the Botsing reproduction engine. The tool requires the following parameters:

- `-project_cp` is the classpath of the project under test and all its dependencies.
- `-crash_log` is the parameter to tell Botsing where the log file to analyze is.
- `-target_frame` is the max level of the target frame. The tool tries to reproduce the crash from the `target_frame` up to the first frame of the stack trace.
- `-N` is the number of reproduction engine instances running in parallel.
- `-X` is the maximal number of times the tool will try to reproduce a frame.

Example

```
java -jar botsing-parallel-jar-with-dependencies.jar -project_cp ~/bin -crash_log
    LANG-20b.log -target_frame 3 -N 2 -X 4
```

All results will be stored in the `test_dir` folder (default value is `crash-reproduction-tests`). This value can be overridden using the additional `-Dtest_dir=new-folder/` parameter.

A full example is available at https://github.com/STAMP-project/botsing-parallel.

3.3 Development status

3.3.1 Fitness function refinement

For now, the Botsing reproduction engine uses the weighted sum fitness function. The multi-objectivization search described in D3.3 has been redeveloped by an external contributor and needs to be included in Botsing (see pull request https://github.com/STAMP-project/botsing/pull/51).

3.3.2 Behavioral model seeding

Behavioral model seeding is available for crash reproduction and test case generation. See https://github.com/STAMP-project/botsing/releases/tag/botsing-1.0.7 for the latest release of the model generation (documentation available at https://stamp-project.github.io/botsing/pages/modelseeding.html). Tutorials on how to use model seeding for crash reproduction and unit test generation are available at:

- https://github.com/STAMP-project/botsing-demo, and
- https://github.com/STAMP-project/evosuite-model-seeding-tutorial.

Future developments

Future developments include the usage and comparison of other learning algorithms.

3.3.3 Stack trace preprocessing

Preprocessing of the stack traces is available in the latest release:

- https://github.com/STAMP-project/botsing/releases/tag/botsing-1.0.7.

Documentation is available at https://stamp-project.github.io/botsing/pages/preprocessing.html.

3.3.4 Code instrumentation extension

The code instrumentation extension for crash reproduction is currently under test and will be merged in the master branch. See branch https://github.com/STAMP-project/botsing/tree/integration_testing.

3.3.5 Botsing parallel processing

The parallel execution of Botsing is released at https://github.com/STAMP-project/botsing-parallel/releases/tag/v1.0.

Conclusion

3.4 Test amplification for common behaviors

Manual crash reproduction is labor-intensive for developers. A promising approach to alleviate them from this challenging activity is to automate crash reproduction using search-based techniques. In this work, we augment this technique using both test and behavioral model seeding. We implement both test seeding and the novel model seeding in Botsing.

For practitioners, the implication is that more crashes can be automatically reproduced, with a small cost. In particular, our results show that behavioral model seeding outperforms test seeding and no seeding without a major impact on efficiency. The different behavioral model seeding configurations reproduce 6% more crashes compared to no seeding, while test seeding reduces the number of reproduced crashes. Also, behavioral model seeding could significantly increase the search initialization rate for 3 crashes compared to no seeding, while test seeding performs worse than no seeding in this aspect. We hypothesize that the achieved improvements by model seeding can be further extended by using more resources (*i.e.,* execution logs) for collecting the call sequences which are beneficial for the model generation.

From the research perspective, by abstracting behavior through models and taking advantage of the advances made by the model-based testing community, we can enhance search-based crash reproduction. Our analysis reveals that (1) using collected call sequences, together with (2) the dissimilar selection, and (3) prioritization of abstract object behaviors, as well as (4) the combined information from source code and test execution, enable more search processes to get started, and ultimately more crashes to be reproduced.

Similarly to search-based crash reproduction, model seeding can be applied to search-based unit test generation. The usage of models can help in initializing complex objects and mimic common object behaviors during the search process. Our future work includes the evaluation of model seeding for unit test generation.

3.5 Code instrumentation extension

Previous studies have introduced many automated unit and system-level testing approaches for helping developers to test their software projects. However, there is no approach to automate the process of testing the integration between classes, even though this type of testing is one of the fundamental and labor-intensive tasks in testing. Therefore, in this deliverable we have introduce a testing criterion for integration testing, called the *Coupled Branches Criterion* (CBC). Furthermore, we have presented an evolutionary-based class integration testing approach called CLING that uses the CBC criterion to generate these kinds of tests with a low budget.

In our investigation of 140 branch pairs that we collected from 5 open source Java projects, we found that CLING has reached an average CBC score of 50% across all classes, while for some classes we reached 90% coverage. More tangibly, if we consider mutation coverage and compare automatically generated unit tests with automatically generated integration tests using the CLING

approach, we find that our approach allows to kill 10% of mutants per class that cannot be killed by unit tests generated with EvoSuite. Finally, we observed 29 crashes of our subject systems, which we could not reproduce using unit test approaches.

The results indicate a clear potential application perspective, more so because our approach can be integrated into any integration testing practice. Additionally, it can be applied in conjunction with other automated unit and system-level test generation approach in a complementary way.

From a research perspective, our study shows that CLING is not an alternative for unit testing. However, it can be used for complementing unit testing for reaching higher mutation coverage and capturing additionally crashes which materialize during the integration of classes. These improvements of CLING are achieved by the key idea of using existing usages of classes in calling classes in the test generation process.

Applied to search-based crash reproduction, integration testing allows to provide more guidance during the search to reach the location where the exception is thrown. Our future work includes a full evaluation of the new fitness function working on multiple frames.

Bibliography

[1] A. Abdurazik and J. Offutt. Using Coupling-Based Weights for the Class Integration and Test Order Problem. *The Computer Journal*, 52(5):557–570, aug 2009.

[2] R. Alexander and A. Offutt. Criteria for testing polymorphic relationships. In *Proceedings 11th International Symposium on Software Reliability Engineering. ISSRE 2000*, pages 15–23. IEEE Comput. Soc, 2000.

[3] R. Alexander and A. Offutt. Analysis techniques for testing polymorphic relationships. pages 104–114, 2003.

[4] R. Alexander, J. Offutt, and A. Stefik. Testing Coupling Relationships in Object-Oriented Programs. pages 1–43, 2009.

[5] R. T. Alexander and J. Offutt. Coupling-based Testing of O-O Programs. *Journal of Universal Computer Science*, 10(4):391–427, 2004.

[6] S. Ali Khan and A. Nadeem. Automated Test Data Generation for Coupling Based Integration Testing of Object Oriented Programs Using Evolutionary Approaches. In *2013 10th International Conference on Information Technology: New Generations*, pages 369–374. IEEE, apr 2013.

[7] F. E. Allen. Control flow analysis. volume 5, pages 1–19, New York, NY, USA, 1970. ACM.

[8] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds. An Industrial Evaluation of Unit Test Generation: Finding Real Faults in a Financial Application. In *2017 IEEE/ACM 39th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, pages 263–272. IEEE, may 2017.

[9] M. M. Almasi, H. Hemmati, G. Fraser, A. Arcuri, and J. Benefelds. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering in Practice Track*, pages 263–272. IEEE Press, 2017.

[10] N. Alshahwan and M. Harman. Automated web application testing using search based software engineering. In *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pages 3–12. IEEE, nov 2011.

[11] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *Proceedings of the 27th international conference on Software engineering*, pages 402–411. ACM, 2005.

[12] A. Arcuri. RESTful API automated test case generation with Evomaster. *ACM Transactions on Software Engineering and Methodology*, 28(1):1–37, 2019.

[13] A. Arcuri and L. Briand. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability*, 24(3):219–250, 2014.

[14] A. Arcuri and G. Fraser. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering*, 18(3):594–623, jun 2013.

[15] T.-D. B Le, D. Lo, C. Le Goues, and L. Grunske. A learning-to-rank based fault localization approach using likely invariants. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 177–188. ACM, 2016.

[16] C. Baier and J.-P. Katoen. *Principles of Model Checking*. MIT Press, 2007.

[17] P. Bansal, S. Sabharwal, and P. Sidhu. An investigation of strategies for finding test order during Integration testing of object Oriented applications. In *2009 Proceeding of International Conference on Methods and Models in Computer Science (ICM2CS)*, pages 1–8. IEEE, dec 2009.

[18] M. Beyene and J. H. Andrews. Generating string test data for code coverage. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 270–279. IEEE, 2012.

[19] F. A. Bianchi, M. Pezzè, and V. Terragni. Reproducing concurrency failures from crash stacks. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering - ESEC/FSE 2017*, pages 705–716. ACM Press, 2017.

[20] A. Blasi, A. Goffi, K. Kuznetsov, A. Gorla, M. D. Ernst, M. Pezzè, and S. D. Castellanos. Translating code comments to procedure specifications. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 242–253. ACM, 2018.

[21] L. Borner and B. Paech. Integration Test Order Strategies to Consider Test Focus and Simulation Effort. In *2009 First International Conference on Advances in System Testing and Validation Lifecycle*, pages 80–85. IEEE, sep 2009.

[22] L. Briand, Y. Labiche, and Yihong Wang. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594–607, jul 2003.

[23] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[24] J. Campos, Y. Ge, N. Albunian, G. Fraser, M. Eler, and A. Arcuri. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information and Software Technology*, 104(August):207–235, 2018.

[25] J. Campos, Y. Ge, G. Fraser, M. Eler, and A. Arcuri. An empirical evaluation of evolutionary algorithms for test suite generation. In T. Menzies and J. Petke, editors, *Symposium on Search Based Software Engineering (SSBSE '17)*, volume 10452 of *LNCS*, pages 33–48, Cham, 2017. Springer International Publishing.

[26] E. G. Cartaxo, P. D. L. Machado, and F. G. O. Neto. On the use of a similarity function for test case selection in the context of model-based testing. *Software Testing, Verification and Reliability*, 21(2):75–100, 2011.

[27] M. Ceccato, A. Marchetto, L. Mariani, C. D. Nguyen, and P. Tonella. Do automatically generated test cases make debugging easier? an experimental assessment of debugging effectiveness and efficiency. *ACM Trans. Softw. Eng. Methodol.*, 25(1):5:1–5:38, Dec. 2015.

[28] N. Chen and S. Kim. STAR: Stack trace based automatic crash reproduction via symbolic execution. *IEEE Trans. on Software Engineering*, 41(2):198–220, 2015.

[29] T. Chen, M. Li, and X. Yao. On the Effects of Seeding Strategies: A Case for Search-based Multi-Objective Service Composition. In *Proceedings of the Genetic and Evolutionary Computation Conference on - GECCO '18*, pages 1419–1426. ACM Press, 2018.

[30] D. Coppit and J. Lian. Yagg: an easy-to-use generator for structured test inputs. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 356–359. ACM, 2005.

[31] M. Črepinšek, S.-H. Liu, and M. Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Computing Surveys (CSUR)*, 45(3):35, 2013.

[32] R. da Veiga Cabral, A. Pozo, and S. R. Vergilio. A Pareto Ant Colony Algorithm Applied to the Class Integration and Test Order Problem. In *IFIP International Conference on Testing Software and Systems*, volume 6435 LNCS of *ICTSS 2010*, pages 16–29. Springer, 2010.

[33] X. Devroey, G. Perrouin, M. Cordy, H. Samih, A. Legay, P.-Y. Schobbens, and P. Heymans. Statistical prioritization for software product line testing: an experience report. *Software & Systems Modeling*, 16(1):153–171, feb 2017.

[34] X. Devroey, G. Perrouin, A. Legay, P.-Y. Schobbens, and P. Heymans. Search-based Similarity-driven Behavioural SPL Testing. In *Proceedings of the Tenth International Workshop on Variability Modelling of Software-intensive Systems - VaMoS '16*, pages 89–96, Salvador, Brazil, jan 2016. ACM Press.

[35] W. Dulz and Fenhua Zhen. MaTeLo - statistical usage testing by annotated sequence diagrams, Markov chains and TTCN-3. In *Third International Conference on Quality Software, 2003. Proceedings.*, pages 336–342. IEEE, 2003.

[36] G. Fraser and A. Arcuri. EvoSuite: automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419. ACM Press, 2011.

[37] G. Fraser and A. Arcuri. The Seed is Strong: Seeding Strategies in Search-Based Software Testing. In *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 121–130. IEEE, apr 2012.

[38] G. Fraser and A. Arcuri. Whole Test Suite Generation. *IEEE Transactions on Software Engineering*, 39(2):276–291, feb 2013.

[39] G. Fraser and A. Arcuri. A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. *ACM Transactions on Software Engineering and Methodology*, 24(2):1–42, dec 2014.

[40] G. Fraser and A. Arcuri. 1600 faults in 100 projects: automatically finding faults while achieving high coverage with evosuite. *Empirical Software Engineering*, 20(3):611–639, 2015.

[41] G. Fraser and A. Zeller. Exploiting common object usage in test case generation. *Proceedings - 4th IEEE International Conference on Software Testing, Verification, and Validation, ICST 2011*, pages 80–89, 2011.

[42] G. Gay, M. Staats, M. Whalen, and M. P. Heimdahl. The risks of coverage-directed test case generation. *IEEE Transactions on Software Engineering*, 41(8):803–819, 2015.

[43] C. Ghezzi, M. Pezzè, M. Sama, and G. Tamburrelli. Mining Behavior Models from User-intensive Web Applications. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE '14, pages 277–287, Hyderabad, India, 2014. ACM Press.

[44] P. Godefroid, A. Kiezun, and M. Y. Levin. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, volume 43, pages 206–215. ACM, 2008.

[45] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè. Automatic generation of oracles for exceptional behaviors. pages 213–224, 2016.

[46] G. Guizzo, G. M. Fritsche, S. R. Vergilio, and A. T. R. Pozo. A Hyper-Heuristic for the Multi-Objective Integration and Test Order Problem. In *Proceedings of the 2015 on Genetic and Evolutionary Computation Conference - GECCO '15*, pages 1343–1350, Madrid, Spain, 2015. ACM Press.

[47] M. Harman, S. A. Mansouri, and Y. Zhang. Search-based software engineering. *ACM Computing Surveys*, 45(1):1–61, nov 2012.

[48] M. J. Harrold and G. Rothermel. Performing data flow testing on classes. *SIGSOFT Softw. Eng. Notes*, 19(5):154–163, Dec. 1994.

[49] N. Hashim, H. Schmidt, and S. Ramakrishnan. Test Order for Class-based Integration Testing of Java Applications. In *Fifth International Conference on Quality Software (QSIC'05)*, volume 2005, pages 11–18. IEEE, 2005.

[50] H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology*, 22(1):1–42, feb 2013.

[51] S. Herbold, P. Harms, and J. Grabowski. Combining usage-based and model-based testing for service-oriented architectures in the industrial practice. *International Journal on Software Tools for Technology Transfer*, 19(3):309–324, jun 2017.

[52] C. Holler, K. Herzig, and A. Zeller. Fuzzing with code fragments. In *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*, pages 445–458, Bellevue, WA, 2012. USENIX.

[53] P. Jaccard. Étude comparative de la distribution florale dans une portion des Alpes et des Jura. *Bulletin del la Société Vaudoise des Sciences Naturelles*, 37:547–579, 1901.

[54] S. Jan, A. Panichella, A. Arcuri, and L. Briand. Search-based multi-vulnerability testing of xml injections in web applications. *Empirical Software Engineering*, pages 1–34, 2019.

[55] S. Jiang, M. Zhang, Y. Zhang, R. Wang, Q. Yu, and J. W. Keung. An Integration Test Order Strategy to Consider Control Coupling. *IEEE Transactions on Software Engineering*, 5589(c):1–1, 2019.

[56] Z. Jin and A. J. Offutt. Coupling-based criteria for integration testing. *Software Testing, Verification and Reliability*, 8(3):133–154, sep 1998.

[57] R. Just, D. Jalali, and M. D. Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 International Symposium on Software Testing and Analysis*, pages 437–440. ACM, 2014.

[58] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser. Are mutants a valid substitute for real faults in software testing? In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 654–665. ACM, 2014.

[59] S. A. Khan and A. Nadeem. Automated Test Data Generation for Coupling Based Integration Testing of Object Oriented Programs Using Particle Swarm Optimization (PSO). In J.-S. Pan, P. Krömer, and V. Snášel, editors, *Proceedings of the Seventh International Conference on Genetic and Evolutionary Computing, ICGEC 2013*, volume 238 of *Advances in Intelligent Systems and Computing*, pages 115–124, Cham, 2014. Springer International Publishing.

[60] F. Kifetew, X. Devroey, and U. Rueda. Java unit testing tool competition: seventh round. In *Proceedings of the 12th International Workshop on Search-Based Software Testing*, pages 15–20. IEEE Press, 2019.

[61] I. Krka, Y. Brun, D. Popescu, J. Garcia, and N. Medvidovic. Using dynamic execution traces and program invariants to enhance behavioral model inference. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering - Volume 2*, ICSE '10, pages 179–182, New York, NY, USA, 2010. ACM.

[62] M. Leemans, W. M. P. van der Aalst, and M. G. J. van den Brand. The Statechart Workbench: Enabling scalable software event log analysis using process mining. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 502–506. IEEE, mar 2018.

[63] R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, A. Egyed, and E. Alba. Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of Software Product Lines. In *2014 IEEE Congress on Evolutionary Computation (CEC)*, pages 387–396. IEEE, jul 2014.

[64] D. Lorenzoli, L. Mariani, and M. Pezzè. Automatic generation of software behavioral models. In *Proceedings of the 13th international conference on Software engineering - ICSE '08*, page 501. ACM Press, 2008.

[65] Y. Lu, Y. Lou, S. Cheng, L. Zhang, D. Hao, Y. Zhou, and L. Zhang. How does regression test prioritization perform in real-world software evolution? In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 535–546. IEEE, 2016.

[66] L. Ma, C. Artho, C. Zhang, H. Sato, J. Gmeiner, and R. Ramler. Grt: Program-analysis-guided random testing (t). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 212–223. IEEE, 2015.

[67] T. Mariani, G. Guizzo, S. R. Vergilio, and A. T. Pozo. Grammatical Evolution for the Multi-Objective Integration and Test Order Problem. In *Proceedings of the 2016 on Genetic and Evolutionary Computation Conference - GECCO '16*, pages 1069–1076, Madrid, Spain, 2016. ACM Press.

[68] M. Martinez and M. Monperrus. Astor: A program repair library for java. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, pages 441–444. ACM, 2016.

[69] P. McMinn. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability*, 14(2):105–156, jun 2004.

[70] P. McMinn, M. Shahbaz, and M. Stevenson. Search-Based Test Input Generation for String Data Types Using the Results of Web Queries. In *IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST '12)*, pages 141–150. IEEE, apr 2012.

[71] U. R. Molina, F. Kifetew, and A. Panichella. Java unit testing tool competition-sixth round. In *2018 IEEE/ACM 11th International Workshop on Search-Based Software Testing (SBST)*, pages 22–29. IEEE, 2018.

[72] D. Mondal, H. Hemmati, and S. Durocher. Exploring Test Suite Diversification and Code Coverage in Multi-Objective Test Case Selection. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, ICST '15, pages 1–10. IEEE, apr 2015.

[73] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson. JCHARMING: A bug reproduction approach using crash traces and directed model checking. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 101–110. IEEE, mar 2015.

[74] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson. A bug reproduction approach based on directed model checking and crash traces. *Journal of Software: Evolution and Process*, 29(3):e1789, mar 2017.

[75] T. B. Noor and H. Hemmati. A similarity-based approach for test case prioritization using historical failure data. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*, pages 58–68. IEEE, 2015.

[76] A. Offutt, A. Abdurazik, and R. Alexander. An analysis tool for coupling-based integration testing. In *Proceedings Sixth IEEE International Conference on Engineering of Complex Computer Systems. ICECCS 2000*, pages 172–178. IEEE Comput. Soc, 2000.

[77] R. Padhye, C. Lemieux, K. Sen, M. Papadakis, and Y. Le Traon. Semantic fuzzing with zest. pages 329–340, 2019.

[78] A. Panichella, F. M. Kifetew, and P. Tonella. Reformulating branch coverage as a many-objective optimization problem. 04 2015.

[79] A. Panichella, F. M. Kifetew, and P. Tonella. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology*, 104(June):236–256, 2018.

[80] A. Panichella, F. M. Kifetew, and P. Tonella. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology*, 104(August):236–256, 2018.

[81] 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 Transactions on Software Engineering*, 44(2):122–158, 2018.

[82] A. Panichella and U. R. Molina. Java unit testing tool competition - Fifth round. *Proceedings - 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing, SBST 2017*, pages 32–38, 2017.

[83] S. Panichella, A. Panichella, M. Beller, A. Zaidman, and H. C. Gall. The impact of test case summaries on bug fixing performance: An empirical investigation. *Proceedings - International Conference on Software Engineering*, 14-22-May-2016:547–558, 2016.

[84] S. Pearson, J. Campos, R. Just, G. Fraser, R. Abreu, M. D. Ernst, D. Pang, and B. Keller. Evaluating and improving fault localization. In *Proceedings of the 39th International Conference on Software Engineering*, pages 609–620. IEEE Press, 2017.

[85] S. Prowell and J. Poore. Computing system reliability using Markov chain usage models. *Journal of Systems and Software*, 73(2):219–225, oct 2004.

[86] M. P. Robillard and R. Deline. A field study of api learning obstacles. *Empirical Software Engineering*, 16(6):703–732, 2011.

[87] J. M. Rojas, G. Fraser, and A. Arcuri. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability*, 26(5):366–401, aug 2016.

[88] J. Rößler, A. Zeller, G. Fraser, C. Zamfir, and G. Candea. Reconstructing core dumps. In *Proc. International Conference on Software Testing, Verification and Validation (ICST)*, pages 114–123. IEEE, 2013.

[89] U. Rueda, R. Just, J. P. Galeotti, and T. E. Vos. Unit testing tool competition—round four. In *2016 IEEE/ACM 9th International Workshop on Search-Based Software Testing (SBST)*, pages 19–28. IEEE, 2016.

[90] A. Schwartz, D. Puckett, Y. Meng, and G. Gay. Investigating faults missed by test suites achieving high code coverage. *Journal of Systems and Software*, 144:106–120, 2018.

[91] A. Scott. Building object applications that work, your step-by-step handbook for developing robust systems using object technology, 1997.

[92] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges. *Proceedings - 2015 30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015*, pages 201–211, 2016.

[93] S. Shamshiri, J. M. Rojas, L. Gazzola, G. Fraser, P. McMinn, L. Mariani, and A. Arcuri. Random or evolutionary search for object-oriented test suite generation? *Software Testing, Verification and Reliability*, 28(4):e1660, jun 2018.

[94] E. K. Smith, E. T. Barr, C. Le Goues, and Y. Brun. Is the cure worse than the disease? overfitting in automated program repair. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pages 532–543. ACM, 2015.

[95] M. Soltani, P. Derakhshanfar, X. Devroey, and A. van Deursen. A benchmark-based evaluation of search-based crash reproduction. *Empirical Software Engineering*, (731529), aug 2019.

[96] M. Soltani, P. Derakhshanfar, A. Panichella, X. Devroey, A. Zaidman, and A. van Deursen. Single-objective versus Multi-Objectivized Optimization for Evolutionary Crash Reproduction. In *Proceedings of the 10th Symposium on Search-Based Software Engineering (SSBSE '18)*. Springer, 2018.

[97] M. Soltani, A. Panichella, and A. van Deursen. Evolutionary testing for crash reproduction. In *Proceedings of the 9th International Workshop on Search-Based Software Testing - SBST '16*, pages 1–4, 2016.

[98] M. Soltani, A. Panichella, and A. van Deursen. A Guided Genetic Algorithm for Automated Crash Reproduction. In *International Conference on Software Engineering (ICSE)*, pages 209–220. IEEE, may 2017.

[99] M. Soltani, A. Panichella, and A. Van Deursen. A guided genetic algorithm for automated crash reproduction. In *Proceedings of the 39th International Conference on Software Engineering*, pages 209–220. IEEE Press, 2017.

[100] M. Soltani, A. Panichella, and A. van Deursen. Search-Based Crash Reproduction and Its Impact on Debugging. *Software Engineering, IEEE Transactions on*, 2018.

[101] S. Sprenkle, L. Pollock, and L. Simko. A Study of Usage-Based Navigation Models and Generated Abstract Test Cases for Web Applications. In *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, pages 230–239. IEEE, mar 2011.

[102] S. E. Sprenkle, L. L. Pollock, and L. M. Simko. Configuring effective navigation models and abstract test cases for web applications by analysing user behaviour. *Software Testing, Verification and Reliability*, 23(6):439–464, 2013.

[103] M. Steindl and J. Mottok. Optimizing Software Integration by Considering Integration Test Complexity and Test Effort. In *Proceedings of the 10th International Workshop on Intelligent Solutions in Embedded Systems*, pages 63–68. IEEE, 2012.

[104] L. D. Toffola, C.-A. Staicu, and M. Pradel. Saying 'Hi!' is not enough: Mining inputs for effective test generation. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 44–49. IEEE, oct 2017.

[105] P. Tonella, A. Marchetto, C. D. Nguyen, Y. Jia, K. Lakhotia, and M. Harman. Finding the optimal balance between over and under approximation of models inferred from execution logs. *IEEE 5th International Conference on Software Testing, Verification and Validation, ICST 2012*, pages 21–30, 2012.

[106] P. Tonella, R. Tiella, and C. D. Nguyen. Interpolated n-grams for model based testing. In *Proceedings of the 36th International Conference on Software Engineering - ICSE 2014*, pages 562–572. ACM Press, 2014.

[107] J. Tretmans. Model based testing with labelled transition systems. *Formal methods and testing*, pages 1–38, 2008.

[108] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007.

[109] A. Vargha and H. D. Delaney. A critique and improvement of the CL common language effect size statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[110] S. R. Vergilio, A. Pozo, J. C. G. Árias, R. da Veiga Cabral, and T. Nobre. Multi-objective optimization algorithms applied to the class integration and test order problem. *International Journal on Software Tools for Technology Transfer*, 14(4):461–475, aug 2012.

[111] S. Verwer and C. A. Hammerschmidt. flexfringe: A Passive Automaton Learning Package. In L. O'Conner, editor, *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 638–642. IEEE, sep 2017.

[112] Z. Wang, B. Li, L. Wang, M. Wang, and X. Gong. Using Coupling Measure Technique and Random Iterative Algorithm for Inter-Class Integration Test Order Problem. In *2010 IEEE 34th Annual Computer Software and Applications Conference Workshops*, volume 1, pages 329–334. IEEE, jul 2010.

[113] Y. Wei, B. Meyer, and M. Oriol. Is branch coverage a good measure of testing effectiveness? In *Empirical Software Engineering and Verification*, pages 194–212. Springer, 2012.

[114] M. White, M. L. Vásquez, P. Johnson, C. Bernal-Cárdenas, and D. Poshyvanyk. Generating reproducible and replayable bug reports from android application crashes. In A. D. Lucia, C. Bird, and R. Oliveto, editors, *Proceedings of the 2015 IEEE 23rd International Conference on Program Comprehension, ICPC 2015, Florence/Firenze, Italy, May 16-24, 2015*, pages 48–59. IEEE Computer Society, 2015.

[115] J. Xuan, X. Xie, and M. Monperrus. Crash reproduction via test case mutation: let existing test cases help. In *Proceedings of the Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 910–913. ACM, 2015.

[116] A. Zeller. *Why Programs Fail - A Guide to Systematic Debugging, 2nd Edition*. Academic Press, 2009.

[117] Y. Zhang, M. Harman, Y. Jia, and F. Sarro. Inferring Test Models from Kate's Bug Reports Using Multi-objective Search. In M. Barros and Y. Labiche, editors, *Search-Based Software Engineering, SSBSE '15*, pages 301–307, Cham, 2015. Springer International Publishing.

[118] Y. Zhou, R. Gu, T. Chen, Z. Huang, S. Panichella, and H. Gall. Analyzing apis documentation and code to detect directive defects. In *Proceedings of the 39th International Conference on Software Engineering*, pages 27–37. IEEE Press, 2017.