The Journal of Systems & Software 193 (2022) 111442

Contents lists available at ScienceDirect

The Journal of Systems & Software

journal homepage: www.elsevier.com/locate/jss

Diversity-driven unit test generation ☆

Marcus Kessel *, Colin Atkinson

University of Mannheim, 68159 Mannheim, Germany

ARTICLE INFO

Article history:
Received 26 April 2021
Received in revised form 27 May 2022
Accepted 14 July 2022
Available online 11 August 2022

Keywords:
Diversity
Test generation
Test amplification
Automation
Behavior
Experiment
Evaluation
Test quality

ABSTRACT

The goal of automated unit test generation tools is to create a set of test cases for the software under test that achieve the highest possible coverage for the selected test quality criteria. The most effective approaches for achieving this goal at the present time use meta-heuristic optimization algorithms to search for new test cases using fitness functions defined on existing sets of test cases and the system under test. Regardless of how their search algorithms are controlled, however, all existing approaches focus on the analysis of exactly one implementation, the software under test, to drive their search processes, which is a limitation on the information they have available. In this paper we investigate whether the practical effectiveness of white box unit test generation tools can be increased by giving them access to multiple, diverse implementations of the functionality under test harvested from widely available Open Source software repositories. After presenting a basic implementation of such an approach, DivGen (Diversity-driven Generation), on top of the leading test generation tool for Java (EvoSuite), we assess the performance of DivGen compared to EvoSuite when applied in its traditional, mono-implementation oriented mode (MonoGen). The results show that while DivGen outperforms MonoGen in 33% of the sampled classes for mutation coverage (+16% higher on average), MonoGen outperforms DivGen in 12.4% of the classes for branch coverage (+10% higher average).

1. Introduction

Automated unit test generation (AUTG) approaches have evolved steadily over the years and can today routinely achieve branch coverage scores of well over 70% per class (Fraser and Arcuri, 2014; Panichella et al., 2017). This has largely been achieved through the use of search-based software engineering techniques (Harman and Jones, 2001) which employ random, meta-heuristic optimization algorithms (Deb, 2011) to search for new test cases using fitness functions defined on existing test cases and a single system under test (SUT) (McMinn, 2011). Given the superiority of search-based techniques over pure random generation approaches, recent research on improving AUTG performance has primarily focused on improving the deployed fitness function(s) and/or optimization algorithms. Key examples are the introduction of the "whole suite" approach (Fraser and Arcuri, 2012b) and the "many-objective optimization" approach, which delivered significant improvements in average branch coverage of between 6% and 8%, respectively (Panichella et al., 2017). However, given the challenging fitness landscape faced by AUTG tools (Wolpert and Macready, 1997; Arcuri and Fraser, 2011), obtaining similar improvements in the future is likely to become more difficult as the enhancements needed to tease out extra performance become increasingly subtle, complex and case-specific.

Although they introduced a radical new strategy for generating test cases, search-based AUTGs are still founded on the underlying premise that there is exactly one implementation of the functionality under test. This is a natural assumption to make, given that the basic goal of testing is to improve the quality of the implementation in hand, developed as part of a software engineering project, but has the consequence that the deployed search algorithms and fitness functions are limited to the information that can be extracted from the single implementation under test. However, several fundamental developments in the software engineering landscape over the last decade have opened up the prospect of mitigating this limitation —

1. The emergence of large, online software repositories containing pre-existing, Open Source implementations of frequently used functionality,
2. the emergence of sophisticated "semantic" code search engines that are able to efficiently find existing implementations of specific functionality within the aforementioned repositories, and,
3. the growing popularity of continuous development practices (e.g., continuous integration etc.) and supporting computing platforms, which means that powerful resources are

☆ Editor: Raffaela Mirandola.

* Corresponding author.
E-mail addresses: kessel@informatik.uni-mannheim.de (M. Kessel), atkinson@informatik.uni-mannheim.de (C. Atkinson).

https://doi.org/10.1016/j.jss.2022.111442
0164-1212/

regularly available (e.g., overnight) to exploit the aforementioned resources. This opens up the prospect of running AUTG tools, such as EvoSuite, on multiple alternative implementations of the SUT (or more precisely, alternative implementations of the functionality realized by the SUT), rather than just the single implementation usually available.

This paper explores the hypothesis that the performance of AUTG tools like EvoSuite can be enhanced by harvesting existing implementations of the functionality (of the system) under test and exploiting the information they contain to deliver higher quality tests. More specifically, by giving such tools access to multiple, diverse implementations of the functionality under test, and thus access to more structural properties and domain information, we believe they will be able to routinely deliver better quality test sets than when working with the SUT alone. Instead of focusing on enhancing the algorithms and fitness functions used by the tools, therefore, the rationale of the proposed approach is to expand the information they have available to work on.

The motivation is similar to that behind the N-version programming approach to verification advocated in the 70s (Chen and Avizienis, 1978; Avizienis, 1985). The idea is that multiple, diverse implementations of a piece of functionality are likely to have different properties (e.g., structure, algorithms, identifiers) which can be exploited to increase the overall reliability of the system. While the focus of the original N-version programming approach was on dynamic behavior comparison rather than test generation, we believe the same abundance of implementation know-how and domain knowledge embedded within the alternative implementations should also be exploitable for test generation.

To explore this hypothesis we constructed a rudimentary (although unavoidably complex) implementation of a diversity-driven AUTG tool using EvoSuite, the most effective AUTG available for Java, and then compared its performance to standard EvoSuite when executed with the same resources and time budget. Given our hypothesis that extra information alone, rather than improvements in the search algorithms and/or fitness functions, is sufficient to improve test generation performance, all executions of EvoSuite were performed using the default parameters except for the time budget, which was varied in a limited way. Our "diversity-driven" AUTG approach, which we call *DivGen*, first harvests alternative implementations of the functionality under test using a traditional interface-based, semantic search of the kind supported by online code search engines (e.g., Hummel et al. (2008), Bajracharya et al. (2014) and De Paula et al. (2016)). Then, after applying certain selection criteria, it applies EvoSuite to each alternative implementation in turn, using its default configuration and time budget, before combining the results into a single, unified test set.

As well as the effectiveness of the test generation strategy per se, the overall effectiveness of the approach is predicated on the availability and harvestability of useful numbers of alternative implementations for a sufficiently high proportion of the kinds of classes developers typically run AUTG tools on. In other words, there are three underlying reasons why our hypothesis might not be valid in practical software engineering settings —

1. The software repository used to harvest alternative implementations may not contain enough alternative implementations for a sufficient numbers of SUTs to supply useful amounts of additional domain information,
2. the search algorithm used to retrieve alternative implementations from the repository may lack the precision and recall needed to do so effectively,
3. the test-generation algorithm (e.g., *DivGen* in our case) may be unable to use the additional domain information embedded in the retrieved alternative implementations to generate higher-quality tests than those generated by just running the AUTG tool on the SUT alone.

The goal of the study presented in this paper, therefore, is to establish the practical feasibility of the overall approach rather than to judge the effectiveness of any one ingredient, including the diversity-driven test generation approach applied (e.g., *DivGen*). For this reason, the underlying strategy used to realize *DivGen* in our prototype implementation is conceptually straightforward, although implementationally complex, and we make no claim that it is particularly sophisticated. In fact, by demonstrating that the idea of diversity-driven AUTG is feasible and potentially beneficial, even when implemented in a rudimentary way, we hope the work will encourage further research into this approach. We believe there is significant scope for optimizing the individual ingredients of the approach as well as the way they are integrated, including the definition of new search algorithms and fitness functions designed to exploit multiple, alternative implementations. In addition, since the focus is on the practical effectiveness of the overall approach, including the first two ingredients mentioned above, our study was designed to illuminate the actual results that practitioners are likely to experience in real projects, rather than on evaluating the effectiveness of the diversity-driven test generation algorithm alone. For this reason, our study samples classes completely randomly from a mainstream online software repository, Maven Central (Sonatype, 2021), which has not been "curated" or sanitized in any way for executability or measurability. Our results show that *DivGen* is indeed able to outperform EvoSuite, run on the SUT alone, but only in terms of strong mutation score. For the structural code metric, branch coverage, EvoSuite outperforms *DivGen* but by a smaller margin.

The rest of the paper is structured as follows. In Section 2 we discuss the background to the presented approach, describe the related work and discuss the different ways it can be characterized. Section 3 then describes the approach and its prototype realization, while Section 4 describes the computing platform and repository we used to implement and evaluate it. Section 5 then describes the design of our evaluation before Section 6 presents the results. Thereafter, Section 7 continues with a discussion of the insights gained by the experiment and the potential for future research. Finally, Section 8 concludes with some closing remarks.

2. Background and related work

The approach presented in this paper combines results from, and contributes to, several areas of software engineering research. In this section we characterize that research and describe how it relates to the proposed approach.

2.1. Automated unit test generation

First and foremost, of course, the paper presents a novel approach for AUTG. The field of automated test generation broadly encompasses all approaches that aim to create tests for a software artifact without input from human developers (Anand et al., 2013). Although tests can be created just from a knowledge of the interface of the system (e.g., by random testing) the majority of approaches require some kind of specification of the functionality under investigation (e.g., source code). As their name implies, AUTGs focus on the generation of tests for SUTs at the granularity of compilation units of programming languages, such as classes in modern, object-oriented programming languages.

The leading AUTGs currently use meta-heuristic search algorithms to improve coverage, such as search-based software testing (SBST) (McMinn, 2011), or (dynamic) symbolic execution (Baldoni et al., 2018) to explore the potential execution paths. EvoSuite, the most effective AUTG tool for Java, uses a hybrid combination of these and other approaches (Galeotti et al., 2013). Since they rely on randomly generated "seed tests" as the starting point for the optimization process, search-based AUTGs are inherently randomized and may deliver different outputs for the same inputs. This complicates the experimental evaluation process (Arcuri and Briand, 2011, 2014) because multiple executions of tools are required to diminish the role of luck in comparisons. Although our approach does not focus on, or aim to enhance, the underlying algorithms and/or fitness functions used in the optimization process per se, it inherits the underlying non-determinism of the search-based AUTG it is using — EvoSuite in our case.

A key challenge when using randomized tools in practice is deciding on what parameter settings to use (Arcuri and Fraser, 2011). EvoSuite, for example, has dozens of parameters, ranging from the configuration of the "search process" (e.g., stopping criteria, fitness criteria) to the configuration of the techniques used. The role of these parameters is difficult for practitioners to understand, both individually and in terms of their interactions. Most practitioners, therefore, choose the default parameters of EvoSuite for practical applications since these have been shown to be a reasonable choice (Arcuri and Fraser, 2011). Our implementation of *DivGen* adheres to this practice when using EvoSuite (i.e., all executions of EvoSuite within *DivGen* use the default parameters), but our experiment includes invocations of EvoSuite with larger time budgets than the default in order to achieve a fair comparison of the diversity-driven approach and the mono-implementation driven approach.

2.2. Software reuse

The presented approach also falls under the general research area of software reuse since it exploits preexisting software from code repositories (Mili et al., 1995). In general, any technique or activity that exploits existing software in the development of new applications can be regarded as a form of software reuse (Krueger, 1992; Frakes and Kang, 2005). This ranges from the use of exemplary snippets of software from forums such as Stack Overflow (Abdalkareem et al., 2017) and the identification of successful patterns and architecture from repositories, to the testing of tools, techniques and hypothesis using large software data sets (Dyer et al., 2015). However, the canonical form of reuse is the incorporation of pre-existing, self-contained software components into new applications to reduce the amount of code that has to be written from scratch. Although the component-assembly vision of software engineering this suggests is attractive, this form of reuse remains rare in practice because of the legal, logistic and psychological issues involved in reusing third party software (e.g., "not invented here syndrome") (Mili et al., 1998). At the present time, therefore, software reuse is usually focused on exploiting the information extractable from existing code to create other useful software engineering artifacts (e.g., tests, as in our approach).

2.2.1. Code search engines

For practitioners, software reuse technologies at the level of classes and methods are usually manifest in two main ways. The first takes the form of so-called "code search engines" which aim to support more "intelligent" ways of finding relevant software than pure "Google-like" queries over their text. This field of research received a major boost at the turn of the century when large scale, Open Source software repositories started to become accessible on the Internet and efficient, full-text search tools such as Apache Lucene[1] became available to index and analyze their contents using NLP techniques. These spawned a range of platforms, referred to as code or software "search engines" (Sim and Gallardo-Valencia, 2013) which aimed to help developers find existing software entities that match the needs of new applications or use cases. Well known examples include Sourcerer (Bajracharya et al., 2014) and Merobase (Hummel, 2008). It is important to note that the term "search" has a different meaning in meta-heuristic search algorithms than in code search engines. The former refers to the use of meta-heuristic optimization engines to cover a theoretical search space, while the latter refers to the use of natural language processing (NLP) and behavioral sampling techniques (Podgurski and Pierce, 1993) to retrieve software components with desired properties from a code repository.

The basic challenge addressed by these platforms was to allow users to express queries that reflect the functionality or behavior of the desired software rather than just its textual properties — that is, to support so-called "semantic code searches". Since code is a form of semi-structured, text-based data, the majority of code search engines are ultimately based on full-text search. However, their query languages assign a special meaning to the core components of source code such as methods, classes etc. This can extend to the level of allowing users to specify the interfaces of the software abstractions they are looking for in terms of one or more operation signatures (Hummel, 2008; De Paula et al., 2016). The goal of such "interface-based searches" is to find all (collections of) code units that implement the specified interface.[2]

The semantic matching performed in such searches essentially applies what are now called NLP techniques to discern the semantics of code modules based on their profiles and the "meaning" of the identifiers used to label their methods and parameters etc. However, since this can be imprecise, several code search engines enhanced the basic interface-driven search technology with an additional testing step to check whether potential candidates actually deliver the desired functionality when executed. The idea of using tests to characterize the functionality of a software implementation was first proposed in the early 90s under the name "behavior sampling" (Podgurski and Pierce, 1992), where it was used to filter routines from a Unix C-function library. However, since it requires test cases to be included in the search query alongside interface information, this capability is usually referred to as test-based search (or test-case based search). Essentially, code search engines supporting test-driven code searches, such as S6 (Reiss, 2009), Merobase (Hummel, 2008), Sourcerer (Bajracharya et al., 2014) and Satsy (Stolee et al., 2016), augment the text-based semantics embedded in the code with behavior semantics embedded in the tests to provide a richer and more precise form of semantic code search.

The harvesting ingredient of the *DivGen* approach presented in this work essentially uses the interface-based search technology mentioned above. The specific variant we use evolved from Merobase's interface-based search technology, and uses the Merobase Query language (MQL) to express search request.[3] However, *DivGen* uses a significantly enhanced implementation provided by our LASSO platform (Kessel and Atkinson, 2019c,a).

[1] https://lucene.apache.org/ (retrieved 2022-07-01).

[2] Note that the use of the term interface is rather abstract and should not be confused with concrete interfaces like in Java, even though they may materialize as such.

[3] The authors were involved in the development of the Merobase search engine at the University of Mannheim.

2.2.2. Recommendation engines

The second main way in which software reuse is manifested to practitioners is through so-called "recommendation systems" which aim to "suggest" opportunities for exploiting the contents of software repositories, either by explicit code reuse, or by the creation of other useful software engineering artifacts derived from the code (e.g., tests) (Robillard et al., 2010). At the level of statements and code snippets, recommendations systems are already widely used and are embedded seamlessly within development platforms. However, at the level of whole compilation units (e.g., classes) this is still largely a research topic and usually complements research on code search engines. For example, the CodeGenie recommendation tool (Lemos et al., 2007) was driven by the Sourcerer search engine (Bajracharya et al., 2014) while the CodeConjurer recommendation tool (Hummel et al., 2008) was driven by the Merobase code search engine (Hummel, 2008).

The distinction between "recommendation" tools and "automatic generation" tools is obviously a fuzzy one, and essentially boils down to the way the products of the tools are delivered to users. Tools are typically characterized as "recommendation" tools if they provide their results in a suggestive way, at opportune moments without disturbing a developer's normal work, and create the results unobtrusively in the background. On the other hand, tools are typically characterized as "automated generation" tools if they are intended to be invoked explicitly by users and only take up resources when explicitly allocated to them. Since classic AUTG tools like EvoSuite have traditionally been used in the latter way they are rarely referred to as "test recommendation" tools. However, the *DivGen* approach presented in this work lends itself to characterization as a recommendation tool for two main reasons. First, like the CodeGenie and CodeConjurer recommendation tools mentioned above it uses the services of a semantic search engine, and the code redundancy they provide access to, to improve developers' productivity. Second, as mentioned previously, since the effective exploitation of exiting code in the generation of tests will likely require greater resources (i.e., time budgets) than those traditionally used with AUTG tools (e.g., EvoSuite's standard), *DivGen* is most suitable for execution in continuous execution environments at times when their resources are idle (e.g., overnight). It is interesting to note that the developers of EvoSuite have also started to explore the application of EvoSuite in this background mode within continuous integration environments. Their continuous test generation approach (Campos et al., 2014) aims to enhance the way AUTG tools are exploited in continuous development environments, and there are already commercial tools that integrate AUTG into build tools like AgitarOne (Agitar, 2021).

2.3. Search-based vs. search-driven software engineering

As explained before, the AUTG approach presented in this work combines approaches from search-based test generation (e.g., EvoSuite) and search-driven software engineering (e.g., code search engine), hence it can be considered a search-enhanced approach (Atkinson et al., 2013). This inevitably leads to potential overloaded terms that create confusion. Most importantly the meaning of the term diversity needs to be further clarified. Whereas the search-driven meaning of diversity reflects the availability of alternative implementations of some class and its functionality (i.e., retrieved from a large code repository), the search-based meaning of diversity is typically attributed to evolutionary algorithms, particularly genetic algorithms, that are often used to realize optimization strategies.

AUTGs such as EvoSuite use genetic algorithms for test generation, but a property which has not been well explored (so far) in this context is "population diversity" and its effect on the quality of the generated tests (Morrison and De Jong, 2002; Albunian, 2017). The goal in this case is to increase the diversity in the population sampled from the search space in order to avoid premature convergence to local optima that may lead to lower-quality tests (i.e., miss global optima). Since diversity-driven test generation focuses on search-driven diversity in terms of assessing multiple alternative implementations, our work does not attempt to explore/solve population diversity in this regard.

3. Studied test generation approaches

In this section we describe the conceptual basis for our proposed diversity-driven test generation approach, define the signatures and algorithms of the component functions and outline how they are realized in our prototype implementation.

Since our approach involves multiple alternative implementations of the "same" system, we need terminology to distinguish between the abstract functionality that a piece of software provides and concrete realizations of that functionality. We refer to the former as a "functional abstraction" (e.g., the functionality of a stack as manifested via a specific interface) and to the latter as an "implementation" (e.g., a particular class that implements the stack functionality using some particular algorithm). In our approach, therefore, we deal with multiple implementations of a given functional abstraction. Also, because the basic input to our approach and EvoSuite is a Java class, we use the term "Class under Test" or CUT rather the more generic term "Software under Test" or SUT. A CUT has methods which can be invoked, including initializer methods like Java constructors. When we use the term "test", we actually refer to the concept of a test sequence, that is, a sequence of statements that invoke one or more operations of a class (e.g., methods or constructors). Optionally, a test sequence may also capture the observable behavior of the actual CUT executed in response to the executed test sequence (for the sake of regression testing). We adhere to the practice in the literature on automated test generation of using the term "test set" to refer to the results provided by AUTG tools.

The basic algorithms (i.e., functions) we use in our approach are summarized in Table 1 and described in greater detail below.

3.1. MonoGen

Although the prototype implementation studied in this paper uses EvoSuite as the underlying AUTG, in principle any mono-implementation based AUTG could be used instead since it is used as a "black box". To make it clear when the mono-implementation based nature of the utilized AUTG is important rather than the idiosyncrasies of EvoSuite we therefore introduce the term *MonoGen* as a generic name for the underlying AUTG. *MonoGen* therefore stands in contrast to *DivGen* which uses it to create unit tests using, in general, multiple implementations of the functional abstraction of interest. When playing the role of *MonoGen* in our prototype realization of *DivGen*, EvoSuite is always invoked with the default parameters since this has been shown to be a reasonable choice (Arcuri and Fraser, 2011).

However, when invoked to provide a baseline for comparison, EvoSuite's time budget is sometimes changed from the default time budget of 2 min, as explained in Section 5, although always in multiples of 2. All other parameters remain set to the default values. Formally, the function $MonoGen(c, b) \rightarrow T$ takes a class under test, c and a time budget, b, measured in minutes, and returns a test set T for class c. Since we use EvoSuite as a black box function, we do not provide any further details about its internal implementation in this paper. Further information can be found in Fraser and Arcuri (2011, 2014), Fraser et al. (2017) and Fraser (2018).

Table 1
Overview of test generation and helper functions.

Function	Description	Label
Single class		
$MonoGen(c, b) \rightarrow T$	EvoSuite which takes a CUT c and time budget b, and returns a test set T	*MonoGen*
Helpers		
$Harvest(c, m) \rightarrow C_{alt}$	Function for harvesting alternative implementations, which takes a class c and an integer m, and returns a set of up to m alternative implementations of c	*Harvest*
$Adapt(c, a, T) \rightarrow T_{adap}$	Function for adapting tests derived from harvested alternative implementations which takes a CUT c, an integer a and a test set T, and returns a test set T_{adap} adapted to the interface c using a maximum number of a adapter mappings	*Adapt*
$Sanitize(c, T) \rightarrow T$	Function for sanitizing test sets, which takes a test set T for a CUT c and returns a sanitized test set T_{san}	*Sanitize*
Multiple classes		
$DivGen(c, m, b, a) \rightarrow T, n$	Diversity-driven test generation function which takes a CUT c and three integers m, b and a and returns a test set T as well an integer n. T is created by applying $MonoGen(c, b)$ to c and up to m retrieved alternative implementations using up to a adapter mappings and a time budget of b. n is the total number of implementations used to generate the tests	*DivGen*
Invocation forms (experiment)		
$MonoGen(c, 2)$	*MonoGen* invoked with the standard EvoSuite time budget of 2 min	$MonoGen_2$
$MonoGen(c, n * 2)$	*MonoGen* invoked using an equivalent time budget to $DivGen(c, m, b, a)$ where n is the returned number of used implementations	$MonoGen_{2n}$
$DivGen(c, m = 15, b = 2, a = 1)$	*DivGen* invoked with the use of a maximum number of 15 alternative implementations, a maximum number of 1 adapter mapping and invocation of $MonoGen(c, b)$ with a time budget of 2 min	$DivGen_{2n}$

3.2. DivGen

The basic idea of *DivGen* (for "Diversity-driven Generation") is to exploit the domain knowledge encapsulated in harvested implementations of the functionality realized by the CUT to generate sets of tests using repeated executions of *MonoGen* on these alternative implementations (as well as the CUT itself). We hypothesize that the richer combined search space obtained through this extra domain information will enable *DivGen* to perform better (i.e., deliver higher quality test sets) than *MonoGen* applied to just c, with a corresponding overall time budget and the same computing resources.

As a black box, *DivGen* provides the same basic service to users as *MonoGen*, but "under the hood" it actually uses *MonoGen* to generate tests from the alternative implementations. The input parameters to $DivGen(c, m, b, a) \rightarrow T, n$ are the class under test c, the maximum number m of alternative implementations to be retrieved by invoking *Harvest*, the time budget b for running *MonoGen* on each available implementation and the maximum number of adapters a to be used to adapt generated tests to the interface of c. The output parameters are a merged test set T, and the number of implementations used in their generation (i.e., 1 plus the number of harvested implementations). The pseudo algorithm of *DivGen* is presented in Algorithm 1. Note that it uses *MonoGen* as well as three helper functions which are described in greater detail in subsequent subsections.

Input: Class c, maximum number m of alternative classes to retrieve, b time budget for MonoGen, a maximum number of adapters
Output: Generated test set T_{san} for c, and n, total number of implementations used to generate tests
Function `DivGen(c, m, b, a)` $\rightarrow T, n$**:**
  $C \leftarrow \{c\}$
  $C_{alt} \leftarrow$ `Harvest(c, m)`
  $C \leftarrow C \cup C_{alt}$
  $T \leftarrow \emptyset$
  for $c_i \in C$ **do**
    $T_{c_i} \leftarrow$ `MonoGen(`c_i`, b)`
    $T_{adap} \leftarrow$ `Adapt(c, a,` T_{c_i}`)`
    $T \leftarrow T \cup T_{adap}$
  end
  $T_{san} \leftarrow$ `Sanitize(c,` T_{adap}`)`
  $n \leftarrow size(C)$
  return T_{san}, n

Algorithm 1: Pseudo Algorithm of *DivGen*

DivGen starts with a pool (set) of classes C which at first only contains the CUT, c. In a second step, it retrieves up to m alternative classes which have similar signatures to that of c using helper function *Harvest*. Next, the retrieved set of alternative classes is added to the pool C of classes which are then used for test generation using *MonoGen*.

For each class $c_i \in C$, the default time budget b is used to apply $MonoGen(c_i, b)$ to each class c_i to generate tests (this is the CUT, c, as well as the alternative implementations). If tests are successfully generated for an alternative implementation they need to be post-processed, that is, each $t \in T_{c_i}$ needs to be adapted to c in order to map the calls to the alternative implementation to c. This involves the identification of a potential mappings between the used and the required interface signatures. Only those tests are returned as part of set T_{adap} (where $T_{adap} \subseteq T$) which could be "made" compatible to c's signatures and which are executable on it. Each test set T_{adap} which is returned $\forall c_i \in C : Adapt(c, a, T)$ is added to the test set, T.

Finally, the last post-processing step uses the helper function *Sanitize* to "clean" T (where $T_{san} \subseteq T$) in order to remove non-executable, flaky and redundant tests. The adapted, "sanitized" set T_{san} for c is then returned by function *DivGen* along with n, the number of total implementations used by *DivGen* to generate tests.

3.2.1. Harvest

The harvest function is the wrapper for invoking the semantic search performed on the underlying repository. As a black box it receives the CUT, c, extracts its interface definition and returns a "diverse", non-code clone set of classes which are judged to implement that interface. Formally, as illustrated in Algorithm 2, $Harvest(c, m) \rightarrow C_{alt}$ takes the class under test c as well as m, the maximum number of alternative classes to retrieve as input parameters and returns a set of alternative implementations, C_{alt}

by invoking the search engine's *Retrieve* function. To do this, it first derives the interface signature of c (i.e., method signatures) $\mathcal{I} = \{m_1, \ldots, m_n\}$ where $m_i \in \mathcal{I}$ denotes a method signature (i.e., an entry method into its functionality) which is used to retrieve alternative implementations. Each method signature $m_i \in \mathcal{I}$ is composed of a name, a list of input parameter types and optionally a return type (e.g., *sum(int,int):int* for a method signature whose body attempts to sum up two numbers). A method signature also includes initializers like Java constructors (e.g., *Stack():Stack*[4]).

The realization of the semantic search technology accessed by *Harvest* to retrieve alternative implementations from the software repository is explained in greater detail later in this section. Of course, alternative variants of *Harvest* could be used in the retrieval step which do not (just) use the information in the interface of the CUT to harvest alternative implementations (e.g., test-driven code search). However, in this first, prototype implementation we focus on interface-driven code searches to realize *Harvest*.

Input: Class c, m maximum number of alternative implementations to retrieve
Output: Alternative Implementations C_{alt} for c
Function `Harvest(`c, m`)` $\rightarrow C_{alt}$**:**
  $\mathcal{I} \leftarrow$ `ExtractInterface(`c`)`
  $C_{alt} \leftarrow$ `Retrieve(`$\mathcal{I}$, m`)`
  $C_{alt} \leftarrow$ `Filter(`C_{alt}`)`
  return C_{alt}

Algorithm 2: Pseudo Algorithm of *Harvest*

3.2.2. Adapt

The function *Adapt* is a key part of our diversity-driven test generation approach because it makes tests generated from alternative implementations applicable to the CUT, c. Since the *Harvest* function retrieves a set of alternative class candidates $c_i \in C_{alt}$ which are supposed to have "similar" (i.e., compatible) interface signatures c, it is necessary to verify that the signatures used in each generated test for $c_i \in C_{alt}$ can be mapped (i.e., adapted) to c. In other words, it is necessary to verify that any test can be transformed into a form where c's methods are invoked instead of c_i's methods.

As illustrated in Algorithm 3, *Adapt* takes in three input parameters, the class under test c, the set, T, of tests generated for each class $c_i \in C_{alt}$ using *MonoGen* (i.e., c plus the retrieved alternative classes), and a, the maximum number of potential adapter mappings to compute between c's signatures $\mathcal{I}_c$ and the signature $\mathcal{I}_{c_i}$ of each $c_i \in C_{alt}$. For each test $t \in T_{c_i}$, function $Adapt(c, T_{c_i}) \rightarrow T_{adap}$ attempts to map the method signatures used in a test to the sought-after method signatures of class c. Any test which does not apply to the predicate of *Adapt* (i.e., is not compatible to the signature of class c), is dropped and thus not added to the set of generated tests for c. The same applies to tests which cannot be made executable on class c (details about the strategies used are discussed later in this section). Eventually, *Adapt* returns a subset of tests $T_{adap} \subseteq T$, each of which is adapted to class c and has been verified to run on c. Naturally, if class c_i equals c, *Adapt* simply returns the unmodified set of tests for c_i (assuming that the tests are not "flaky").

Similar to Randoop's (Pacheco et al., 2007) and EvoSuite's (Fraser and Arcuri, 2011) internal abstraction of tests, we model

Input: Class c, a maximum number adapters, T set of tests to adapt
Output: Adapted set of tests T_{adapt}
Function `Adapt(`c, a, T`)` $\rightarrow T_{adap}$**:**
  $T_{adap} \leftarrow \emptyset$
  for $t_i \in T$ **do**
    $A \leftarrow$ `ComputeAdapters(`c, a, t_i`)`
    for $a_i \in A$ **do**
      $t_{adap} \leftarrow$ `Transform(`c, a_i, t_i`)`
      $T_{adap} \leftarrow T_{adap} \cup \{t_{adap}\}$
    end
  end
  return T_{adap}

Algorithm 3: Pseudo Algorithm of *Adapt*

a test as a sequence of statements. Each statement consists of an operation, a list of input parameters and an output. Formally, a test $t \in T$ is modeled as a sequence of ordered statements $\mathcal{S} = \langle s_1, \ldots, s_n \rangle$ where $s_i \in \mathcal{S}$ denotes a triple $s_i = [Op, I, O]$ at location i in the sequence. Op is the operation, I the list of inputs to the operation and O the output. For the adaptation, it is important to note that only those operations that are mappable to the CUT are considered, which can include constructors as well as ordinary methods. Any operations resolved to "third-party" classes like the Java API or third-party library classes such as unit testing support are not touched. The same applies to operation types other than methods and constructors like field or array element operations (i.e., set/get).

Therefore, in order to adapt $\mathcal{S}_{c_i}$ of class c_i to target CUT c, we need to define a mapping $a_i : \mathcal{S}_{c_i} \rightarrow \mathcal{S}_c$ which maps and transforms the signatures $\mathcal{I}_{c_i}$ of c_i in sequence $\mathcal{S}_{c_i}$ to the signatures $\mathcal{I}_c$ of c in $\mathcal{S}_c$. Since the test generation process is aimed at the mining of test inputs, we do not include the observed outputs of class c_i in $\mathcal{S}_{c_i}$. Instead, as part of the adaptation, the newly created sequence $\mathcal{S}_c$ of target CUT c is executed and its response (i.e., observed behavior) is encoded into the expected output of the sequence.

It is important to note that depending on the nature of the method signatures as well as the available adaptation operators, multiple mappings between the signatures of two classes can be identified.[5] The parameter a places a bound on the number of possible mappings to be considered. Potential adapters (i.e., mappings) are returned in order of their relevance, so the best match is put first, the second best match second and so on.

If more than one potential adapter mapping is identified and used as part of the process, it follows that for each test generated by *MonoGen*, multiple adapted tests may be added to the output test set. This means that we can invoke the CUT c differently (e.g., by switching parameters), resulting in different observable behavior (e.g., *subtract(2,1)* returns 1 whereas *subtract(1,2)* returns -1). So setting the maximum number of adapter mappings, a, to a value greater than one has the consequence that for each feasible adapter mapping we add additional tests. Moreover, if we need to deal with more than one method, the number of possible adapter mappings may grow exponentially. For example, the functional abstraction stack has methods such as *push(Object):Object*, *pop():Object*, *peek():Object* and *size():int*. Assuming that we need to map method signature *removeLast():Object* of some CUT to the aforementioned method signatures, there are two possible direct mappings based on the type signatures (input and output types) which is *pop* and *peek*.

[4] Since Java constructors return an instance of the type in which they are defined, we implicitly assume the return type "Stack", so we can omit the return type in the signature.

[5] e.g., the order of input parameters can be switched or types can be converted to other types.

Obviously, based on the semantics of the stack functional abstraction those mappings result in different observable behaviors since *pop* returns and removes an element at the top of the stack, whereas *peek* just returns an element, but does not remove it.

Finally, we need to stress that the adaptation process may not preserve the original semantics of the execution scenario defined in the tests. Since adaptation is a "best practice" approach based on a set of heuristics and adaptation operators, it is not guaranteed that the original semantics of the tests adapted to the target CUT are preserved. However, this is not a problem since output (i.e., oracle) values are not reliable products of current AUTG technology, and we generate the new output (i.e., oracle) values by running the test on the CUT. When the maximum number of adapters, a, is increased, it is possible to derive multiple tests from a single test generated by *MonoGen* which may lead to a larger variety of interesting tests (e.g., recall the subtraction example of subtracting two numbers with respect to the switching of parameters).

Our current implementation of *Adapt* generates adapters for each test independently. Another strategy would be to generate adapters for the entire set of tests, so that the adapter mapping is stable across tests of the same class. For the prototype realization, however, we opted to generate adapters per test since in this strategy the success rate is likely to be higher. This is because we only need to focus on the signatures of the test at hand, so there is no need to infer and aggregate the signatures from the entire set of tests beforehand.

3.2.3. Sanitizing tests

The generated set of tests collected from the available implementations (i.e., the CUT and the harvested alternative implementations) are further post-processed to check for non-executable tests, flaky tests and redundancy. Transforming tests for one class to another presents a non-trivial challenge that may result in non-executable tests. For example, unresolvable dependency issues can arise in which, for example, some methods or fields used as part of the execution scenario of a test are not available in the context of c. In such cases, these tests are not included in the output. Our approach needs to detect flaky tests, like any other AUTG tool, so all tests adapted for c are re-executed to detect flaky (i.e., probably non-deterministic) behavior. The collection of tests generated from the available implementations may also result in a subset of redundant tests (e.g., from our experience, EvoSuite often generates tests with "null" test inputs). The role of the sanitize function, $Sanitize(c, T) \rightarrow T_{san}$, is to perform this sanitization process. It takes in a test set T for the class under test c and cleans it using the aforementioned criteria.

3.3. Prototype implementation of DivGen

Although the basic idea of *DivGen* is intuitive, creating an effective implementation is challenging since there are a multitude of possible strategies for realizing the helper functions, particularly *Harvest* (e.g., what selection criterion achieves the most interesting coverage of the domain input space?) and *Adapt* (e.g., what is a good mapping strategy?). Indeed, EvoSuite itself (our instance of *MonoGen*), has a non-trivial number (dozens) of tuning parameters that may lead to good results in one particular scenario, and poor results in another (parameter tuning suffers from the consequences of the "no free lunch theorem" in optimization (Arcuri and Fraser, 2011)). *DivGen* adds even more tuning parameters which need further exploration such as the number of alternative implementations to retrieve, the time budget(s) to use for individual implementations as well as the number of adapter mappings to compute for each generated test.

Regardless of the strategy applied, and the parameter values picked, the infrastructure needed to realize the proposed approach in a practical setting requires a scalable, non-trivial, unified platform which can retrieve and process a large number of implementations in parallel in an automated manner (similar to continuous integration platforms). For this reason, we developed a rudimentary realization of *DivGen* on top of our platform for large-scale software observations, LASSO (Kessel and Atkinson, 2019c,b,a), which provides the repository and semantic search technology employed. The goal of the prototype realization is to evaluate the basic viability of a diversity-driven approach. The current realization of *DivGen* is therefore a rudimentary, non-optimized implementation that uses basic strategies for harvesting implementations and adapting generated tests for the CUT. We have not attempted to "tweak" the algorithm and the parameters identified to optimize performance and leave this to future work.

Fig. 1 summarizes *DivGen*'s process based on the example of a stack, called "Stack", which is the CUT for which we aim to generate tests. The realization of each of the four steps is explained in greater detail below.

3.3.1. Harvesting implementations using interface-driven code search

As described in Section 3.2.1, the first stage in the process is the harvesting of alternative implementations which involves three sub-steps —

1. Extraction of the "visible" interface signature of the CUT, c,
2. selection of (i.e., querying for) alternative implementation candidates based on the extracted signature,
3. filtering out class duplicates to increase implementation diversity.

In order to extract the interface signature of c, we identify all the "visible" signatures it defines. Since we focus on Java classes, visibility in this case refers to the methods (including constructors) which can be invoked by consumers of the classes (e.g., a JUnit test method[6]). The strategy we use to harvest alternative implementations for c is to retrieve alternative implementation candidates using an interface-driven code search. For this, the extracted interface signature of the class is represented in a UML-like DSL for code search[7] as illustrated by the signature specification of the Stack example in Fig. 1.

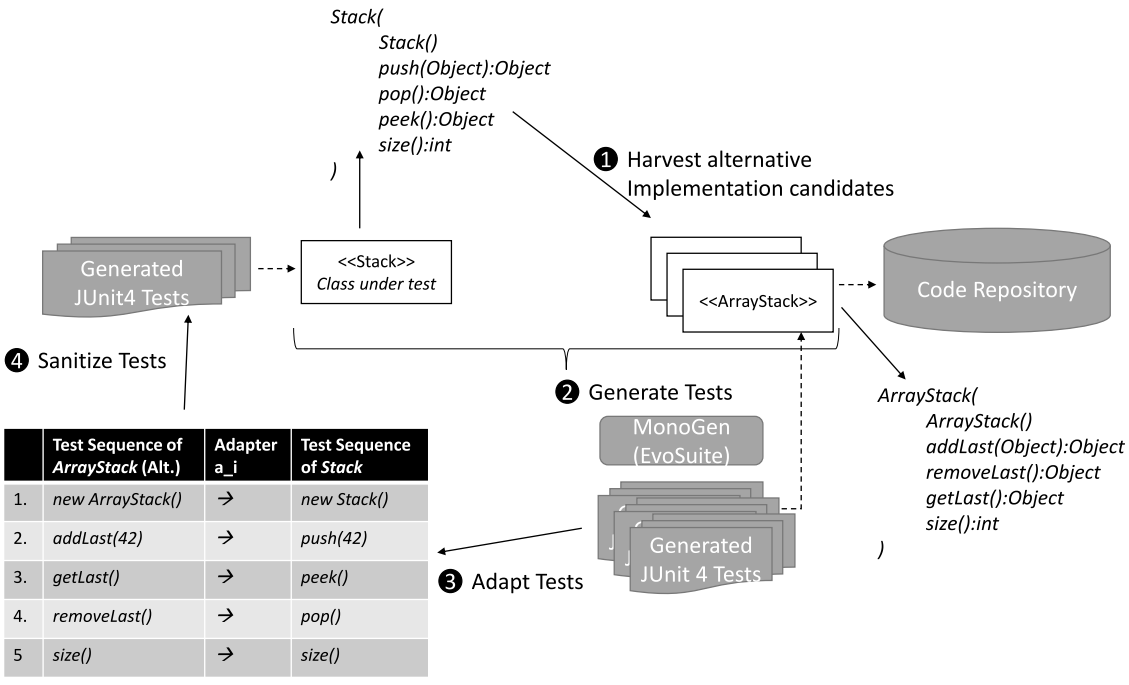
Based on the methods' signatures (input-, output types and naming), an MQL interface-driven code query (e.g., *Math (sqrt(double):double;)*) is formulated to request classes with similar (ideally identical) signatures. The basic similarity of signatures is computed using Solr/Lucene's[8] BM25-based retrieval mechanism, however LASSO makes further attempts to increase recall by employing two query expansion techniques: (1) method parameter type relaxations of common data types (e.g., *int* to *long*), and (2) simple thesaurus lookups of method name synonyms and antonyms based on the WordNet dictionary (Miller, 1995; Lemos et al., 2014).

In order to attain a minimal level of diversity in the returned candidate implementations, we apply type-2 code clone detection to ensure that the same class is not included multiple times. Since large repositories may contain exact, or very close, duplicates of classes, we reject duplicate classes. Otherwise, in the extreme case when all alternative implementations are identical, *DivGen*

[6] Note that methods declared as *private* cannot be invoked externally, however *protected* methods can be invoked by classes using the same package name.

[7] LASSO supports the Merobase Query Language (MQL) defined by the Merobase code search engine query (Hummel, 2008).

[8] https://solr.apache.org/ (retrieved 2022-07-01).



	Test Sequence of *ArrayStack* (Alt.)	Adapter a_i	Test Sequence of *Stack*
1.	*new ArrayStack()*	→	*new Stack()*
2.	*addLast(42)*	→	*push(42)*
3.	*getLast()*	→	*peek()*
4.	*removeLast()*	→	*pop()*
5	*size()*	→	*size()*

Fig. 1. DivGen: Stack example.

falls back to the running of *MonoGen* multiple times on the same CUT.

To detect duplicate classes, and thereby increase the diversity of the returned set of classes, we configured the clone detector Nicad[9] (Cordy and Roy, 2011) to reject all Type-1 and Type-2 clones which are true syntactical clones (cf. Roy and Cordy (2007)) of the reference- and/or alternative implementations' entry methods. For this, Nicad was given the source code of the class bodies. Whenever a group of clone classes were detected by Nicad, we simply picked the first representative of each clone group in the order in which they were returned by the interface-driven code search engine. The CUT, of course, is guaranteed to remain in the set of implementations.

3.3.2. Automated test generation

Having retrieved a pool of class implementations for the CUT, *DivGen* invokes *MonoGen* (in our case EvoSuite) to generate tests. To this end, we run EvoSuite[10] using its default settings (see Section 5.2.1 for details), including a search budget set to two minutes and the stopping condition set to the default setting "*MaxTime*." The test criteria to be optimized were left to EvoSuite's default settings. The output is a set of automatically generated JUnit4 classes.

3.3.3. Adapting JUnit test classes

Once JUnit4 classes have been generated using EvoSuite for all available implementations (i.e., the retrieved implementations plus the CUT), they are analyzed to derive their test sequences. For each JUnit4 class we construct an abstract syntax tree (AST) using JavaParser[11] and inspect all the declared test methods (i.e., annotated with *@Test*). The goal of this analysis is to extract an abstract test sequence model from each test method and to —

1. identify all calls to methods of the CUT,
2. create a template test sequence from which required method signatures can be identified in order to compute adapters for the target class c,
3. fill in the template test sequence by substituting required method signatures of the CUT with adapter mappings identified for the target class.

For each test sequence, we analyze each statement with respect to its operation (cf. Section 3.2.2). If the operation resolves to a call to the CUT for which the test was generated, we create a special template statement which carries the information about the method signature invoked (name, input types as well as output type). Ultimately, from each test sequence we construct a template sequence which is then the subject for adaptation to the target class c. Depending on the number of adapter mappings for the target class c identified for the template sequence, we create one or more concrete test sequence instances.

To make the test sequence mining process more flexible, we either use the entire sequence (if possible) or derive a subsequence in case only a partial adaptation was applicable (i.e., an ordered sub-sequence, starting from the first statement up to a particular statement). Note that technically this process is non-trivial since unresolvable operations sometimes occur, such as the use of types which are not available in the context of the target class (e.g., internal fields or constants of the original class etc.). A sequence is not successful if there is at least one statement which cannot be resolved. In this case, the test sequence is dropped. The executability of every successfully mined test sequence is verified on the target class, c, by running it. If the execution fails, the test sequence is dropped. Sub-sequences are often identified when the execution of a mined test sequence fails prematurely. If the sequence succeeded for at least one operation, we nevertheless add it to the list of successfully mined sequences.

As part of the successful execution of a mined test sequence, we observe the output (i.e., observable behavior) of target class, c, at each statement and record it for later assertion generation as part of the JUnit4 class transformation. Technically, since we use Java's reflection mechanism that allows inspection, instantiation

9 Nicad 6.2.1.
10 EvoSuite 1.10 which uses the "DynaMOSA" algorithm.
11 https://javaparser.org/ (retrieved 2022-07-01).

and invocation, we call each statement of a test sequence "reflectively" and record its returned output, including classic return outputs or exceptions thrown.

To overcome method signature mismatches between the target class interface and the alternative class implementations, we used LASSO's adaptation capabilities to compute a compatible mapping which is invocable with the inputs passed to the original statement of the test sequence at hand. For this purpose, we use a best practice process using a set of adaptation operators in order to obtain an ordered list of adaptation mappings based on their likely relevance based on their —

- *Assignability*: type hierarchy analysis for each method parameter (input/output) by walking up the inheritance hierarchy of a type to find assignable types up to the root type,
- *Casting/Relaxation*: primitive (wrapper) type casting and relaxation,
- *Switching*: switching parameter type positions,
- *Conversion*: converting a supported type to another (e.g., *string* value to *byte[]* array),
- *Producer*: an operator which attempts to identify an initializer of the class (or none if methods are static).

First, adapter mappings of the method's parameters are computed using Java's Reflection API (allowing for type relaxations such as *int* to *long* etc.) which are then ranked using a prioritization schema based on method name and type "closeness". Like the optimization approach in Wang et al. (2016), a loose weighting scheme is employed to compute the ranks of method permutations to prioritize the "best" methods for matches. Because we also need to adapt instance methods, the adaptation algorithm also attempts to find suitable initializers for the target class (e.g., constructor). In case the source methods are based on an instance, but the target class is defined as a corresponding static method, the producer adaptation operator creates a "no operation" statement which does nothing, but retains the structure of the test sequence.

3.3.4. Sanitizing test sequences

Eventually, once we have mined and adapted a collection of test sequences for the target class *c*, the final step is to ensure that the tests are runnable and "stable" on *c*. This process includes the transformation of the test sequences back to JUnit4 test classes and includes various checks. But first, we attempt to identify any test sequences which are redundant and drop those duplicates in order to obtain a non-redundant test set.

Since our test sequences were called "reflectively", there is a certain likelihood that this runtime environment differs from what is experienced when the test sequences are transformed back into JUnit4 classes. For this reason, we first translate the test sequence model back into a syntactically correct JUnit4 class and then attempt to run it in order to observe its execution of test methods. Each test sequence maps to a JUnit test method. Whenever we have previously observed outputs in the reflective execution of the statements, we generate corresponding assertions which verify the behavior observed when run as part of the JUnit testing framework.

Whenever we observe a discrepancy (i.e., assertion failure), we drop the corresponding test method. Sometimes the JUnit4 test classes which are generated cannot be compiled, since some of their dependent types cannot be resolved in the build context of the target class, *c*. We have therefore implemented a fallback mode which attempts to remove any portions of the test class which cannot be compiled based on the compiler's feedback, and we try to compile and run the test class again. Similarly, in the event that some types cannot be resolved "lately" at runtime (i.e., late-time binding mechanism), we try to remove any of those tests which exhibit this kind of failure.

Finally, since we attempt to construct JUnit4 classes, we also use this opportunity to identify flaky tests. If we observe non-deterministic runtime behavior, we also drop the corresponding tests. After all these checks and cleaning steps, the "sanitized" set of tests (i.e., JUnit4 classes) is returned. As explained in Section 5, we use the exported JUnit4 classes as input for the test coverage tools we use as part of the experiment.

4. Study platform and executable corpus

In this section we provide a brief overview of the platform used to realize the functions described in the previous section and to host the executable corpora of software implementations used to evaluate them.

4.1. LASSO platform

We performed our study using the LASSO platform, a "Large-Scale Software Observatorium" that supports the observation-based analysis of code repositories (Kessel and Atkinson, 2019c,b,a). This is built around a simple domain model of the code units comprising a software repository along with a simple workflow model of the actions that can be performed on them. Users can leverage these models using a dedicated domain specific language called the LASSO Scripting Language (LSL). Unlike other tools, such as BOA (Dyer et al., 2015), which also offer large-scale analysis services accessible via an abstract DSL, LASSO provides dynamic (e.g. behavior-based) as well as static analysis services.

The platform incorporates numerous techniques to enhance the realization diversity of harvested classes/methods, including code clone detection and elimination (Roy and Cordy, 2007), and build script synthesis to increase the number of executable, and hence testable, classes (Palsberg and Lopes, 2018). Most importantly, it provides an execution engine which supports the mining, adaptation, execution and observation of test sequences. LASSO also allows a range of additional selection constraints to be applied when retrieving and matching classes such as filtering based on (object-oriented) code attributes (e.g., members, dependencies, method invocations, Java keywords etc.), and properties characterized by metrics like LOC and cyclomatic complexity (McCabe, 1976).

4.1.1. Build script synthesis/automation

LASSO uses the Maven 3 ecosystem (Maven, 2021) to support scalable, automated build automation and build script synthesis for generating executable corpora (Dietrich et al., 2017). This allows LASSO to leverage Maven's rich dependency management mechanism, plugin ecosystem (including a variety of analysis plugins etc.) and established reporting system. For each implementation retrieved, LASSO basically synthesizes a project build script by generating a Maven compatible project object model (i.e., pom.xml). In order to manage parallel builds on each worker node, LASSO extends Maven with a custom "event spy" which is able to provide event-based reporting of currently running builds which worker nodes can start concurrently (multi-threading) to improve efficiency. Since implementations and builds are executed on a cluster of worker nodes, they not only run in parallel (i.e., scale vertically) but also scale horizontally.

Most of the available LSL actions (e.g., EvoSuite and test coverage measurements) are implemented using Maven's rich ecosystem of plugins. For example, for test generation with EvoSuite we rely on EvoSuite's Maven plugin.

Table 2
LASSO's Maven central corpus statistics (November 2020).

Unit	Total	Unique
Artifacts	184,464	184,464
Compilation units	8,884,430	6,947,672
Classes (non-abstract)	6,682,724	5,281,170
Classes (abstract)	531,732	397,691
Constructors	10,700,527	4,064,347
Methods (non-abstract)	75,335,199	28,916,079

4.1.2. Controllable execution environments

LASSO provides a sandbox environment for executing actions and implementations securely using "docker" containerization.[12] This not only facilitates the specification of controllable (e.g., Java version etc.), reusable and isolated runtime environments (Boettiger, 2015) it also enables fine-grained control over resource allocation and permissions. Moreover, using docker, execution environments can easily be constructed and shared in terms of downloadable container "images" through the docker hub repository. They can also be created through custom container configurations to integrate new execution environments and new analysis tools into LASSO on-the-fly.

4.2. Executable code corpus

The corpus used to drive the study was selected to meet two core requirements – (1) a high likelihood of "testability" in terms of executable implementations and (2) sufficient size to support a high degree of diversity. In general, the platform can be configured to operate on several external data sources, but our study used its default corpus of Java code harvested from the Maven Central repository (Sonatype, 2021). This contains a large range of popular third party Open Source libraries such as Spring, Apache Commons and Google Guava etc. which are developed by both Open Source and industry practitioners. Packaged Maven artifacts can contain a variety of types such as binary code artifacts (*.class), plain source code artifacts (*.java) and supplementary artifacts such as textual documentation containing rich metadata (Raemaekers et al., 2013, 2017).

The key statistics for LASSO's repository are shown in Table 2. At the time this study was conducted, the repository contained 184,464 indexed Maven artifacts, 8, 884, 430 indexed Java compilation units (i.e., stored class documents) and 75, 335, 199 indexed Java (non-abstract) methods (i.e., stored method documents). In addition to general source code information, the documents also contain a variety of metrics derived by static (i.e., structural) code analysis and static code measurement (e.g., structural properties like members, dependencies, method calls as well as static code analysis measurements). Note that the number of "unique" classes/methods is considerably lower that the "raw" number once trivially redundant artifacts have been rejected by simple string hash comparison (e.g., identical copy/paste clones of method bodies).

5. Study design

As explained previously, for diversity driven test generation to be effective in practice, it is not only necessary to have an approach and a platform that can exploit a diverse set of implementations of a functional abstraction when they are available, it is also necessary for a suitable level of software redundancy to actually be available in online repositories. In other words, the widespread software redundancy that underpins our approach not only needs to be exploitable, it also needs to exist and be accessible in practice. With the traditional mono-implementation approach used to realize current AUTG tools there is no question mark over the existence of the conditions needed to apply the approach, since the availability of an implementation of the CUT is a basic tenet in testing. Experimental evaluations of existing approaches have therefore typically used sanitized code corpora such as SF110 corpus (Fraser and Arcuri, 2014) where issues of dependency resolution and class execution have been pre-resolved by hand. However, since our goal is to evaluate the practical feasibility of diversity-driven test generation, our experiment uses the non-sanitized corpus described in the previous section containing code from a mainstream, online software repository (cf. Section 4.2). This provides a better indication of the results that practitioners can expect to experience in real-world software projects and thus increases the generalizability of the results.

Since the basic goal of our study is to establish whether there is a benefit from exploiting the diversity available in software repositories when generating tests using EvoSuite, we compare the quality of the test sets delivered by *DivGen* and *MonoGen* on randomly sampled classes under the same resources and overall time budget. More specifically, we compare $DivGen(c, m, b, a)$ to both $MonoGen(c, 2 * n)$ and $MonoGen(c, 2)$ for each subject class c. The inclusion of $MonoGen(c, 2)$ is to provide a basic baseline. The parameter, n, of $MonoGen(c, n * 2)$ is the number of class implementations processed by *DivGen* (returned as the output n), with a maximum value of 15+1 (i.e., the CUT plus the maximum number of alternative implementations, i.e., $m = 15$ for $DivGen(c, m, b, a)$). The metrics that we use to measure test quality are strong mutation score (MS) and branch coverage (BC), which are the most widely used in AUTG tool evaluations in the literature.

Our study therefore has two research questions, the first focusing on *DivGen's* relative performance with respect to mutation score and the second focusing on *DivGen's* relative performance with respect to branch coverage. Each of these is symmetrically divided into two sub-questions, one comparing $DivGen_{2n}$ to $MonoGen_{2n}$ and one comparing $DivGen_{2n}$ to $MonoGen_2$ (cf. Table 1). This gives a total of four concrete sub-questions —

- *RQ1 (a)* How does $DivGen_{2n}$ perform compared to $MonoGen_{2n}$ on mutation score?
- *RQ1 (b)* How does $DivGen_{2n}$ perform compared to $MonoGen_2$ on mutation score?
- *RQ2 (a)* How does $DivGen_{2n}$ perform compared to $MonoGen_{2n}$ on branch coverage?
- *RQ2 (b)* How does $DivGen_{2n}$ perform compared to $MonoGen_2$ on branch coverage?

5.1. Class sampling (subjects)

Unit testing technologies in the Java ecosystem, such as JUnit and EvoSuite, work at the granularity of classes and methods since they require self-contained executable modules with well-defined signatures and input/output parameters. It is important to note that our selection strategy differs from those used in related studies. The authors of EvoSuite, for instance, usually evaluate their tool on their well-curated SF110 corpus (benchmark) (Fraser and Arcuri, 2014) which was selected by randomly sampling 100 projects from SourceForge and adding the top 10 most popular projects. Our selection strategy, however, was optimized to randomly select a distinct set of classes from a huge variety of projects within a non-curated corpus. Another important difference is that the projects and classes in the SF110 corpus were manually transformed into an executable corpus, whereas

[12] https://www.docker.com/ (retrieved 2022-07-01).

LASSO's executable code corpus is fully automated (i.e., both the generation of the corpus and the sampling of executable classes). This has the advantage that our study was able to sample classes from a much larger search space. This applies both for the sampling of class subjects (i.e., functional abstractions and a reference implementation to serve as the CUT), of which we sampled 120 in our experiment, as well as for the retrieval of alternative class implementations which are obtained by *DivGen* through its *Harvest* function.

The automated sampling of "executable" classes from a large search space is challenging. Firstly, it requires a retrieval approach based on a database of classes which can be systematically queried for classes and their attributes (here using LASSO). Secondly, even though we take advantage of Maven's dependency resolution using Maven Central as the main repository, several problems can block the automated "project synthesis" of sampled classes, and thus their "executability" (e.g., wrong assumptions about versions and execution environment, "provided" missing or unresolvable artifacts etc.). LASSO's corpus, therefore, provides a best-effort approach which attempts to return executable classes based on a set of heuristics in an attempt to solve such problems.

The goal of the experiment was to evaluate diversity-driven test generation on a large number of subjects. To maximize generalizability, therefore, these subjects were selected completely at random from LASSO's code corpus. Table 2 shows the number of classes which were potentially retrievable for the experiment (i.e., 5, 281, 170 unique classes). For the sake of practicality, based on our past experience with LASSO and its corpus, we applied the following selection criteria —

- owner classes as well as their methods must be visible (i.e., non-private),
- classes and their methods must be production classes and not (unit) test classes (i.e., we rejected any classes which depend on common classes in popular unit testing frameworks including JUnit3/4/5 or TestNG; e.g., *org.junit.Test*),
- we rejected any classes from the default package name spaces of Java (e.g., *java.*, *javax.*, *com.sun* etc.) as well as EvoSuite's name space (since it is published in Maven Central as well),
- a class is required to have at least one invocable method,
- the minimum number of branches in a class must be at least 10[13] to rule out trivial functionality.

In addition to the selection criteria, the adaptation capability of the current incarnation of *DivGen* only supports Java compatible types (i.e., classes and types defined by the JDK). Complex, nested custom types specified by a particular class cannot be mapped and are thus unsupported. Any class which only contains custom type signatures is therefore not considered.

5.2. DivGen evaluation

To evaluate *DivGen* in this experiment we used the following configuration of $DivGen(c, m = 15, b = 2, a = 1)$ for each randomly sampled class. Parameter m allows the retrieval of up to 15 alternative implementations for class c. Since we set the time budget parameter b to 2 min, $MonoGen(c_i, b = 2)$ is applied to each retrieved alternative implementation and c to generate tests. To keep the experiment as manageable as possible, we set the maximum number of adaptation mappings to compute per test to 1. The choices of the parameters are not optimized for a particular setting since the intention is to understand and explore the overall feasibility and effectiveness of *DivGen*.

[13] This is a rule of thumb.

We establish two baselines for comparisons to check whether *DivGen* is an effective approach. First, we compare it to $MonoGen(c, b = n*2)$ in order to assign *MonoGen* the same search budget to generate tests as *DivGen*. Second, we also compare *DivGen* to the classic, default EvoSuite settings of $MonoGen(c, b = 2)$. The goal is to evaluate whether *DivGen* achieves comparable or better results than *MonoGen* under the same budget constraints and execution conditions. The execution of $MonoGen_2$ (i.e., EvoSuite with a default time budget) on the target class c serves as a second way to assess whether alternative implementations actually contribute to the quality of the generated test sets.

5.2.1. EvoSuite's parameter setting

The default parameters of EvoSuite have been shown to be a reasonable choice (Arcuri and Fraser, 2011). Our implementation of *DivGen* adheres to this practice when using EvoSuite (i.e., all executions of EvoSuite within *DivGen* use the default parameters), but our experiment also includes invocations of EvoSuite with larger time budgets than the default in order to achieve a fair comparison of the diversity-driven approach and the mono-implementation driven approach. Similar to Panichella et al. (2017), we mention only a few of the many parameters (Fraser and Arcuri, 2012b) which may influence the performance of the DynaMOSA algorithm and which are set to their defaults —

- *Population size* is set to 50 individuals,
- *Crossover* is set to single-point crossover with crossover probability set to 0.75,
- *Mutation* is set to uniform mutation with mutation probability equal to 1/size where size denotes the number of statements in a test case which are subject to mutation operations (i.e., adding, deleting and modifying statements),
- *Selection* is set to tournament selection of tournament size equal to 10,
- *Search Budget* is set by the particular function as presented in Table 1,
- *Criteria* to be optimized is set to "LINE, BRANCH, EXCEPTION, WEAKMUTATION, OUTPUT, METHOD, METHODNOEXCEPTION, CBRANCH".

5.2.2. Random sampling classes

Classes were randomly sampled based on the selection criteria presented above. Since *DivGen* generates tests for all classes, we re-arranged the order in which classes are passed to *MonoGen*. To increase efficiency, we put the CUT c, (i.e., the randomly sampled class) first in the result set. If *MonoGen* was not able to generate tests for the CUT, the sampled class is dropped and the random sampling continued. The idea behind this approach is to avoid spending time on classes that are probably not executable since EvoSuite could not generate tests for them. Any randomly sampled class for which EvoSuite was able to generate tests was included in the pipeline to be further processed by *DivGen* and $MonoGen(c, b = n * 2)$.

5.2.3. Harvesting alternative implementations

Similar to the sampling of random classes and the generation of tests for them, we dropped any alternative classes from the pipeline in which the first attempt to generate tests failed. To establish a fair foundation for the comparison of approaches by consuming the same overall budget, when we were able to discover that an alternative implementation was not executable by EvoSuite we retrieved another one. For various reasons, it was not always possible to harvest 15 alternative implementations. One of the reasons is that the interface of the randomly sampled class may be too specific, so the interface-driven code search was not able to retrieve many similar classes. Apart from mismatches, there are several other technical issues that can cause

the retrieval of an alternative to fail. Amongst others, sometimes EvoSuite was not able to generate tests, since the classes required at run-time were not resolvable.

5.3. Evaluating test set effectiveness

In order to compare test set quality, we applied two coverage criteria frequently used in software testing studies to the test sets generated by the approaches under comparison (Rojas et al., 2015; Andrews et al., 2006; Fraser and Arcuri, 2014; Panichella et al., 2017) –

- *Branch Coverage* measured by JaCoCo (2021), which is the percentage of the branches exercised by a test set relative to the total number of branches,[14]
- *Strong Mutation Coverage* measured by the mutation testing tool PIT (Coles et al., 2016) using its default group of mutation operators,[15] which is the percentage of the mutants "killed" by a test set relative to the total number of mutants generated (i.e., strong mutation score).

As well as facilitating coverage measurement, running the generated JUnit4 classes also served as an additional check to ensure that the tests run deterministically without any failures. In case any failures occurred, we disregarded the test class. If we were not able to obtain the measurements for all approaches and the coverage criteria, we dropped the corresponding randomly sampled class.

5.4. Statistical significance

In order to increase the trustworthiness and generalizability of our results, we follow the well known guidelines of Arcuri and Briand (Arcuri and Briand, 2011, 2014) for evaluating inherently randomized algorithms such as the meta-heuristics search approach used by EvoSuite (i.e., genetic algorithms). Special care is required when evaluating such tools, since different runs, under the same conditions and inputs may yield different results. In the case of EvoSuite, the randomness means that there can be fluctuations in the quality of the returned test sets. When measuring test coverage criteria, therefore, it is important to accommodate this kind of non-deterministicness, since EvoSuite underpins both of the compared algorithms.

Since our experiment is essentially a feasibility study, we decided to prioritize maximization of the number of cases analyzed rather than the precision of each individual analysis, and therefore selected the minimum number of repetitions recommended by Arcuri and Briand (i.e., 10) which is also the number used by Panichella et al. (2017). In other words, within the overall time budget available for the experiment, we ran the AUTG approach under study 10 times for each randomly sampled class (i.e., CUT). We therefore obtained 10 coverage measures for each randomly sampled class, approach and coverage criterion (i.e., mutation score and branch coverage).

5.5. Practical challenges

The experiment was executed on six worker nodes (Intel(R) Core(TM) i9-10900T CPU @ 1.90 GHz with 32 GB RAM). To accelerate resource intensive actions like test generation and coverage measurement, we exploited the multi-threading capabilities of the machines by processing multiple implementations in parallel. The use of threads and other resources in EvoSuite was left to the default settings.

We used LASSO's sandbox mechanism (i.e., docker containers) to control the execution environments of each implementation. All implementations were therefore tested under the same environmental conditions based on OpenJDK's Java 1.8.[16] As a precaution, the sandbox was configured to ensure a reasonable level of safety. Our experience with foreign implementations retrieved from large repositories is similar to the one described by the EvoSuite authors in Fraser and Arcuri (2014).

We also faced technical and practical challenges similar to those reported by Fraser and Arcuri (2013) in the real-world generation of tests for Java software components from SourceForge (i.e., SF110). Such challenges include timeout issues when invoking methods etc. To tackle these challenges, we set reasonable timeouts based on our past experience with the LASSO platform and its corpus. The processes of EvoSuite and test coverage tools were therefore forced to terminate after a reasonable amount of time.

LASSO attempts to maximize the testability of candidate implementations fetched from Maven Central. Technically, however, making candidate implementations executable is still a practical challenge, even though Maven Central contains a lot of information about candidate builds and dependencies. As a result, this limits the success rate of our analysis, since the success of EvoSuite at generating test sets for arbitrary alternative implementations is limited. In the experiment, therefore, we removed instances of classes which could not be executed and hence were not testable. Note that in contrast to the issues faced in leveraging public code repositories, in practice we expect that commercial code repositories, such as those used in industry, will achieve a higher degree of executability.

6. Results and analysis

In this section we present the results of our study. We first characterize the randomly sampled classes, provide an overview of the tests generated for the approaches under analysis and present statistics about their performance. The remainder of this section explains what the obtained data tells us about the research questions introduced in Section 5. The data set including the results is available online (Kessel and Atkinson, 2022).

6.1. Overview

We first present an overview of, and characterize, the randomly sampled classes for the experiment. Recall that each sampled class actually provides two things of interest to the experiment — an implementation which represents the CUT for each sample and a functional abstraction that represents the behavior implemented by that CUT. Since *DivGen* retrieves and analyzes different implementations of that functional abstraction during its test generation process, it is the functional abstraction that characterizes each sample in the experiment, not the CUT per se.

To identify the different test generation functions, we use the labels presented in the bottom part of Table 1: $MonoGen_2$ denotes the invocation of *MonoGen* using the default time budget of 2 min, $MonoGen_{2n}$ denotes the invocation of *MonoGen* using a time budget of $n * 2$ minutes which is equivalent to the time budget used by the corresponding invocation of *DivGen*, on the same CUT, namely the number of available implementations (i.e., the CUT plus alternative harvested implementations) times the default time budget of 2 min. We use this time budget for *MonoGen* in order to establish a fair baseline for comparison with respect to

[14] https://www.eclemma.org/jacoco/trunk/doc/counters.html (retrieved 2022-07-01).

[15] https://pitest.org/quickstart/mutators/ (retrieved 2022-07-01).

[16] Using docker image tagged *maven:3.5.4-jdk-8*.

Table 3
Descriptive statistics: Overview (120 Randomly Sampled Classes from Maven Central).

	mean	sd	min	max	se	q.25	median	q.75
Randomly sampled classes								
# Methods	9.53	6.91	2.00	36.00	0.63	4.00	7.50	13.00
Cyclomatic complexity	23.91	15.50	7.00	89.00	1.42	13.00	19.00	28.00
# Branches	28.39	22.26	10.00	146.00	2.03	15.50	22.00	32.00
# Lines	48.49	38.39	6.00	204.00	3.50	25.00	37.00	56.25
# Mutants generated	34.92	30.24	1.00	200.00	2.76	18.00	26.00	40.00
DivGen								
# Clones detected	7.94	8.29	2.00	46.00	0.93	3.00	4.00	9.00
# Non-clone implementations n	12.96	5.91	0.00	16.00	0.17	14.00	16.00	16.00
# Non-redundant Tests harvested	67.93	113.17	0.00	813.00	3.27	7.00	21.00	79.25
Tests generated								
# Tests $MonoGen_2$	18.41	19.05	0.00	148.00	0.55	6.00	14.00	24.25
# Tests $DivGen_{2n}$	78.35	121.78	0.00	843.00	3.52	10.00	27.50	96.00
# Tests $MonoGen_{2n}$	19.02	21.80	0.00	207.00	0.63	6.00	14.00	25.00

the assigned time budgets. The actual number of classes used by *DivGen* in its test generation process is determined by the results of the *Harvest* function and is returned as one of the outputs of the function along with the generated tests. Although *DivGen* does not have a time budget parameter like *MonoGen* we nevertheless use the subscript $DivGen_{2n}$ when referring to the results for *DivGen* as a reminder of the time budget used in its execution.

We have observed two main reasons why n is sometimes smaller than the maximum of 15. The first is that the interface-driven code search service used by *Harvest* may simply not be able to retrieve 15 alternative implementations as there are too few matching classes available in the underlying repository. The second is that the tools used to generate tests (EvoSuite) or to measure test coverage criteria (PIT and JaCoCo) sometimes have external[17] or internal execution issues that make certain candidate implementations unprocessable. In the runs where EvoSuite did not return a test set, we simply recorded the coverage score of 0 to reflect the missing measurements. The recorded results have therefore not been sanitized by attempting to redo failed runs. If all approaches failed to return a test set for all runs, we removed that particular class from the data set.

We randomly sampled classes from our corpus until we ran out of time. This resulted in 120 randomly sampled classes that satisfied the selection criteria presented in Section 5 and could be fully processed in the analysis pipeline. For each of the three approaches under comparison, we attempted 10 repetitions (runs) for each randomly sampled class. So in total we executed each approach 1200 times given the following overall executions times for the different approaches ($MonoGen_2$: $1200 * 2$ minutes, $DivGen_{2n}$: $1200 * n * 2$ minutes, and, $MonoGen_{2n}$: $1200 * n * 2$ minutes where n is up to 16 (15 alternative classes plus CUT)).

The top part of Table 3 provides an overview of the basic statistics for the 120 sampled classes based on their size-based metrics measured using JaCoCo as well as the mutants generated using PIT. The second part of Table 3 presents the statistics about the number of implementations processed by $DivGen_{2n}$ as well as the number of non-redundant (no duplicates) tests harvested from them. Finally, the bottom part of Table 3 gives an overview of the number of tests generated by each approach. As shown in the table, the complexity of the sampled classed was, by design, non-trivial. The size-based measurements show that the classes have an average of 9.53 methods (including Java constructors), an average of 48.49 lines of code and 28.39 branches in their bodies, and an average cyclomatic complexity of 23.91. Next to size-based measurements, the average number of mutants generated by PIT based on its default set of mutation operators for the sampled classes is 34.92. The quartiles (.25, .5) also show that the number of mutants generated for the majority of classes is non-trivial (i.e., greater than 18 mutants for 3/4 of the classes).

The bottom part of the table shows that $DivGen_{2n}$ managed to obtain at least one alternative, non-redundant class in 1006 out of 1200 runs (i.e., a success rate of 83.8%). Moreover, $DivGen_{2n}$ managed to obtain an average of 12.96 non-clone classes to serve as alternative implementations for test generation. Such a high average number of classes suggest that the underlying corpus exhibits a considerable level of diversity with respect to the functional abstractions appearing and the structural properties of their implementations. The Nicad clone detection tool identified an average of 7.94 clones of type-2 for each alternative class retrieved by the *Harvest* function. All clone classes were rejected and took no further part in the test generation process of $DivGen_{2n}$. The large average number of code clones suggest that the underlying source for the corpus (i.e., Maven Central) exhibits a high level of trivial redundancy.

From the non-clone implementations available to $DivGen_{2n}$, the approach managed to mine an average of 67.93 non-redundant tests in addition to the tests generated for the CUT. The number of harvested tests suggest that $DivGen_{2n}$ is actually able to retrieve useful numbers of alternative class implementations and that its prototype adaptation algorithm for adapting them to the CUT is effective.

Overall, for the tests generated for all the approaches under analysis, we observe that $MonoGen_2$ generates fewer tests on average (i.e., 18.41) for the randomly sampled classes than $DivGen_{2n}$ and $MonoGen_{2n}$. $DivGen_{2n}$, however, generates significantly more tests on average than $MonoGen_{2n}$ (i.e., 78.35 vs 19.02). Since we do not attempt to apply test suite minimization techniques as part of $DivGen_{2n}$'s test generation, the results are more or less expected (i.e., the more alternative implementations we analyze, the more tests we likely obtain). Since the aim of this study was to establish the feasibility of diversity-driven AUTG, the minimization of tests was not a goal and is a topic for future work as explained in Section 7.

6.2. Characterizing diversity

The diversity of the harvested alternative implementations for a particular CUT obviously has a key influence on the effectiveness of our approach, and at the very least it is important that the alternative implementations are not identical. $DivGen_{2n}$ fulfills this basic criteria by detecting and removing type-2 clones and thus ensures that the harvested implementations are at least different. However, this gives no indication of "how different" (i.e., diverse) the harvested sets of implementations typically are which is obviously important in order to understand the nature

[17] e.g., unresolved classes.

Table 4
Diversity indicators: Spread of key properties of harvested implementations.

	mean	sd	min	max	se	q.25	median	q.75
# Distinct class name	11.17	4.07	1.00	16.00	0.38	10.00	12.00	14.00
# Distinct package name	12.38	3.77	1.00	16.00	0.35	11.00	14.00	15.00
# Distinct projects (URIs)	12.54	4.10	1.00	16.00	0.38	11.00	14.00	16.00
# Distinct branches	9.29	3.61	1.00	16.00	0.34	7.00	10.00	12.00
# Distinct lines	11.56	3.71	2.00	16.00	0.35	11.00	13.00	14.00
# Distinct Cycl. Complexity	8.10	3.54	1.00	15.00	0.33	6.00	9.00	11.00

of the subjects the approach is working with. Unfortunately there is no accepted definition of what "diversity" means in the context of software and how it should be measured. So to provide an indication of the kinds of implementation diversity $DivGen_{2n}$ is able harvest and work with, in this section we provide some statistics on the spread of key characteristics of software and the spread of their distances.

We picked two groups of properties: (1) those that are related to the purpose and origin of the classes, and, (2) those related to their size and complexity. In each case we selected three properties. For the first group we selected: (1) the distinct name of the class, (2) the distinct name of its package, and (3) the distinct origin of the class (i.e., its project in terms of the unique Maven artifact URI which consists of a *groupId*, *artifactId* and a *version*). For the second group we selected three size-based complexity metrics: (1) number of branches, (2) number of lines, and (3) cyclomatic complexity (McCabe).

The first group is interesting because these properties include indicators of the origin of the classes and thus of the likelihood that the classes were written by different software engineers. This is important, since different software developers are known to make different mistakes and apply different realization choices which is a rich source of diversity (cf., N-version programming (Chen and Avizienis, 1978; Avizienis, 1985)). The second group of metrics are of course important as key indicators of structural and complexity differences in class implementations. The size-based measurements were obtained using JaCoCo.

Table 4 summarizes the descriptive statistics about the aforementioned properties over all randomly sampled classes and the alternative implementations harvested by $DivGen_{2n}$. Note again that we retrieved up to 16 different class implementations (i.e., randomly sampled class plus the harvested class implementations).

Overall, the averages and quartiles (.25, median and .75) for basically all distinctness measures suggest significant diversity among the sets of class implementations (i.e., with respect to the number of unique classes per class-related property). For instance, the randomly sampled implementations came, on average, from 12.54 different projects (median 14). Alternatively, if we look at the distinct number of branches over all implementation sets, we get an average of 9.29 distinct implementations for a median of 10 implementations.

Fig. 2 provides a different visualization of the spread of property values for the 120 functional abstractions studied in the experiment by showing the frequency distributions of the numbers of harvested implementations with distinct properties for each of the 120 randomly sampled classes. The top three histograms show the frequencies of the number of distinct values for the purpose/origin properties, while the bottom ones show the frequencies of the number of distinct values for the size-based complexity metrics. Overall, for all six properties, we can see that the frequencies mainly peak in the second half of the plots, between 8 and 16 distinct implementations. This indicates that for the majority of our randomly sampled classes and their harvested implementations, there is a significant level of diversity at least in terms of these six properties.

Finally, to provide some deeper insights into the diversity of harvested implementations, Table 5 provides greater detail about these metrics for the alternative implementations harvested for two randomly sampled classes from the 120 study subjects. The first three columns show the class name, the package name and their project (Maven) URI, whereas the remaining three columns depict the indicator measures for diversity based on the number of lines, the number of branches and the cyclomatic complexity per class. Here we can see that for both randomly sampled classes (i.e., *ByteHashMap* and *ExtensionFileFilter*), sets of alternative classes were harvested which exhibit significant differences with respect to the six properties analyzed.

6.3. RQ1 (a): How does $DivGen_{2n}$ perform compared to $MonoGen_{2n}$ on mutation score?

In RQ1 (a), we assess how $DivGen_{2n}$ performs compared to $MonoGen_2$ and $MonoGen_{2n}$ for strong mutation score as measured by PIT on the randomly sampled classes. In order to determine statistically whether the comparison results obtained as part of our study are significant, we conduct statistical analyses similar to the experimental analysis of Panichella et al. and their DynaMOSA algorithm (Panichella et al., 2017).

Accordingly, we use the non-parametric Wilcoxon test (Conover, 1999) as proposed by Arcuri and Briand (2014) (also referred to as the "Mann–Whitney U test") with a p-value threshold of 0.05. p-values below 0.05 indicate a significant difference between two approaches based on the coverage criterion under consideration (i.e., mutation score or branch coverage). In other words, significant p-values reject the null hypothesis that there is no significant difference between the results from the two approaches. As proposed by Arcuri and Briand (2014) and used in related studies in the field of search-based software testing, we complement hypothesis testing with the Vargha–Delaney $\hat{A}_{12}$ estimate (Vargha and Delaney, 2000). $\hat{A}_{12}$ is an effect size measure that indicates the magnitude of significant differences between two approaches and their observations (e.g., $DivGen_{2n}$ vs $MonoGen_{2n}$). Next, the interpretation of the actual effect size measure $\hat{A}_{12}$ is used to indicate three situations which may occur when two approaches, X and Y, are compared based on their significant differences: (1) if $\hat{A}_{12} = 0.5$ then $\hat{A}_{12}$ indicates that there is no noticeable difference between X and Y, (2) if $\hat{A}_{12} > 0.5$, then $\hat{A}_{12}$ indicates that X is significantly better than Y, and finally, (3) if $\hat{A}_{12} < 0.5$, then $\hat{A}_{12}$ indicates that Y is significantly better than X (or X is significantly worse than Y).

In the following, we present the results of the comparisons between the two pairs of approaches ($DivGen_{2n}$ vs $MonoGen_{2n}$ and $DivGen_{2n}$ vs $MonoGen_2$) for 10 runs on the sampled classes for which we determine a significant difference for the coverage measure under consideration based on the p-value. The inclusion criteria for the set of randomly sampled classes under investigation for a pair of approaches and the coverage criterion measures, therefore, is a p-value threshold of 0.05 as well as $\hat{A}_{12}$ unequal to 0.5.

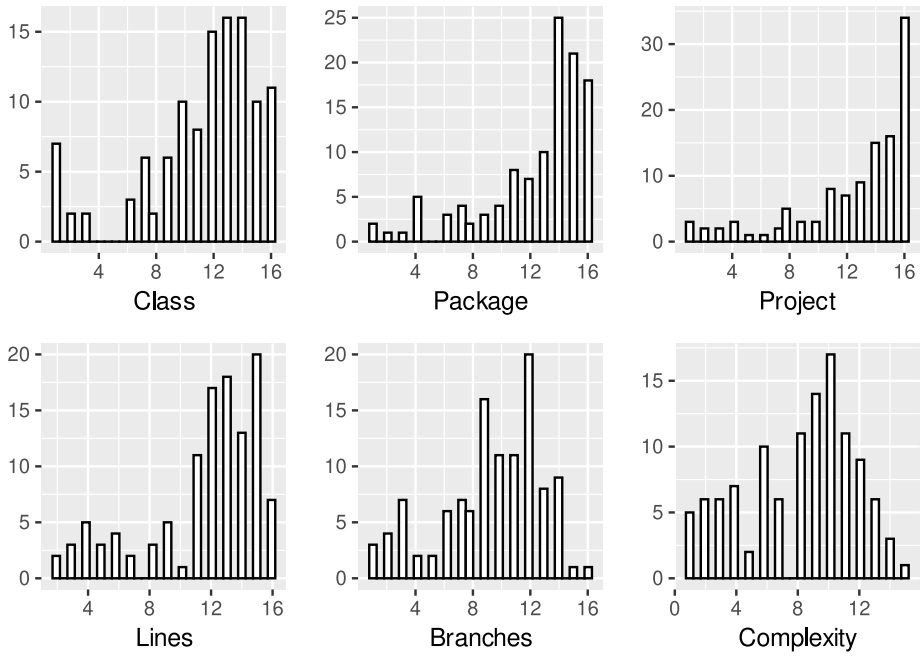


Fig. 2. Diversity indicators: Histogram of frequency distributions of key properties.

Table 5
Diversity indicators: Two samples of randomly sampled classes (bold) and their harvested classes.

Class	Package	Project (Maven URI)	#Lines	#Branches	#Complexity
ByteHashMap	org.kafkacrypto.types	org.kafkacrypto:kafkacrypto-java:0.9.9.7	36	12	19
ByteArrayMap	com.github.sviperll.collection	com.github.sviperll:chicory-core:0.36	24	6	17
DynamicMap	org.elasticsearch.script	org.elasticsearch:elasticsearch:7.9.3	22	2	14
JsonMap	com.haulmont.yarg.loaders.impl.json	com.haulmont.yarg:yarg:2.0.12	26	8	18
V8PropertyMap	com.eclipsesource.v8.utils	com.eclipsesource.j2v8:j2v8_win32_x86:4.6.0	45	30	28
BeanMap	jetbrick.bean	com.github.subchen:jetbrick-commons:2.1.9	32	12	19
JedisByteHashMap	redis.clients.jedis.util	com.redislabs:jedis:3.0.0-m1	35	12	19
RemoteMapProxy	org.sapia.regis.cache	org.sapia:sapia_regis:3.1	32	8	17
MergedMap	services.moleculer.web.router	com.github.berkesa:moleculer-java-web:1.2.12	30	14	20
MonitoredMap	pl.aislib.util	net.sf.aislib:aislib:0.6.1	60	44	35
RegexHashMap	de.unihd.dbs.uima.annotator.heideltime.resources	com.github.heideltime:heideltime:2.2.1	53	26	27
AvroMapWrapper	org.apache.pig.impl.util.avro	org.apache.pig:pig:0.17.0	41	20	23
OrderedMap	httl.util	com.github.httl:httl:1.0.11	43	30	28
ByteHashMap	com.navercorp.redis.cluster.util	com.navercorp:nbase-arc-java-client:1.5.4	37	12	19
AttributesMap	com.knoema.series	com.knoema:knoema-java-driver:1.0.7	32	26	26
ReadWriteMap	com.talent.aio.common.syn	com.talent-aio:talent-aio-common:1.6.7.v20170328-RELEASE	88	2	14
ExtensionFileFilter	org.freehep.graphicsbase.swing	org.freehep:freehep-graphicsbase:2.4	54	36	31
ExtensionFileFilter	jptools.swing	net.sf.jptools:jptools:1.7.2	64	40	32
ExportFileFilter	org.tn5250j.spoolfile	com.github.vebqa:tn5250j:0.7.6.4	70	46	38
FuzzerFileFilter	org.owasp.jbrofuzz.util	org.owasp.jbrofuzz:jbrofuzz-encoder:2.5.1	42	28	21
SimpleFileFilter	prefuse.util.io	org.timebench:prefuse-vienna:1.0.0	28	10	12
ExtensionFileFilter	de.uni_leipzig.asv.util.commonFileChooser	de.uni_leipzig.asv.toolbox:toolbox-utils:1.0	20	8	10
CustomFileFilter	org.ow2.jonas.autostart.utility	org.ow2.jonas.autostart:utility:1.0.0-M2	17	14	11
ExtensionFileFilter	com.antlersoft.util	com.antlersoft.browsebyquery:browsebyquery-core:0.9.3	53	10	13
UniversalFileFilter	emulib.runtime	net.sf.emustudio:emuLib:9.0.1	31	20	20
DefaultFileFilter	org.bounce	nz.ac.waikato.cms.weka.thirdparty:bounce:0.18	42	28	21
ExtensionFileFilter	org.microemu.app.ui.swing	org.floggy.3rd.org.microemu:microemulator:3.0.0-r2474	20	14	12
JFileFilter	bdsup2sub.tools	com.github.riccardove.easyjasub:bdsup2sub-lib:5.2.0	57	36	30
CompoundFileFilter	com.actelion.research.chem.io	com.actelion.research:openchemlib:2020.11.1	63	38	33
ExtensionFileFilter	org.jflac.apps	org.jflac:jflac-examples:1.5.2	57	36	30
OpenDialogFileFilter	com.freedomotic.jfrontend.utils	com.freedomotic:frontend-java:3.0	62	36	31
ExtensionFilter	nu.zoom.swing	nu.zoom:base:1.0.1	24	14	13

Table 6 combines the results obtained for the three approaches for strong mutation score for each randomly sampled class with $\hat{A}_{12} \neq 0.5$ where p-value ≤ 0.05. For space reasons,[18] we report $\hat{A}_{12}$ measures only for which the p-value is below the threshold. The first column denotes the name of the randomly sampled class, while the second column denotes the project from which it was sampled. The middle part of the table shows the mutation scores obtained for each class for each approach averaged over

[18] A complete table including all p-values is available in the data set provided.

Table 6
Mean mutation score achieved for each class with $\hat{A}_{12} \neq 0.5$ where p-value ≤ 0.05.

	Class	Project (Maven URI)	Mutation score			$\hat{A}_{12}$ *DivGen*$_{2n}$			
			MonoGen$_2$	*DivGen*$_{2n}$	*MonoGen*$_{2n}$	vs. *MonoGen*$_2$		vs. *MonoGen*$_{2n}$	
						>0.5	<0.5	>0.5	<0.5
1	H265PerTitleConfiguration	com.bitmovin.api.sdk:bitmovin-api-sdk:1.52.0	0.79	1.00	0.79	1.00		1.00	
2	ProjectedVolumeSource	io.alauda:kubernetes-model:0.2.12	0.58	0.72	0.57	0.79		0.81	
3	ScannerCallable	com.google.code.maven-play-plugin.org.apache.hbase:hbase:0.21.0–20100622	0.14	0.18	0.14	0.91		0.91	
4	MemoryBlob	org.apache.cayenne:cayenne-server:4.2.M2	0.77	0.78	0.77	0.76			
5	CharTypes	com.fasterxml.jackson.jr:jackson-jr-all:2.12.0-rc1	0.75	0.81	0.76	0.75		0.72	
6	MaximumFilter	org.gjgr.pig:pig-chivalrous-image:0.0.1.20200202G	0.60	0.80	0.47	0.89		0.89	
7	FuVectordouble	org.fudaa.framework.ctulu:ctulu-fu:2.5	0.82	0.87	0.82	0.82		0.81	
8	AlertsExternalServiceCondition	com.ocadotechnology...:newrelic-api-client:3.4.1	0.72	0.96	0.75	1.00		1.00	
9	NodeBuilder	io.fabric8:kubernetes-model-core:4.12.0	0.01	0.29	0.36	0.83			
10	RegionStreamConfig	com.homeaway.streamplatform:stream-registry:0.5.2	0.74	0.91	0.74	0.91		0.92	
11	PaginationContext	com.amazon.alexa:ask-smapi-model:1.9.0	0.79	0.90	0.79	1.00		1.00	
12	SSLTermination	org.apache.jclouds.api:rackspace-cloudloadbalancers:2.2.1	0.95	0.99	0.95	0.93		0.93	
13	AliPayConfig	cn.springboot:best-pay-sdk:1.3.4-BETA	0.75	0.92	0.75	0.95		0.95	
14	HTMLEntity	net.homeip.yusuke:twitter4j:2.0.10	0.75	0.80	0.73	0.86		0.83	
15	Result	cn.detachment:detachment-core:1.0.6-RELEASE	0.59	0.94	0.60	1.00		1.00	
16	ForestNode	club.lemos:jtool:1.0.0-RC1	0.69	0.87	0.72	1.00		1.00	
17	SearchDetailsType	com.github.lespaul361:eBaySDK1027:1027.0.0	0.82	0.95	0.80	0.97		0.94	
18	ImHistory	com.clivern:fred:1.0.2	0.46	0.78	0.46	0.98		0.98	
19	StringUtil	com.telnyx.sdk:telnyx:2.1.1	0.87	0.99	0.57	0.89		0.94	
20	BinaryCodec	in.clouthink.daas:daas-token:1.8.1	0.91	0.99	0.85	0.97		1.00	
21	ExtensionFileFilter	org.freehep:freehep-graphicsbase:2.4	0.88	0.91	0.86	0.74		0.85	
22	ByteUtils	com.github.1991wangliang:sds-socket:1.2.5	0.76	0.87	0.76	1.00		1.00	
23	PostgresWriter	org.revenj:revenj-core:1.2.0	0.67	0.71	0.70	0.77			
24	JSON	io.fabric8:kubernetes-model:4.9.2	0.62	0.80	0.58	1.00		1.00	
25	MaskUtil	com.github.shipengyan:spy-framework:1.6.0	0.82	0.91	0.84	0.91		0.89	
26	CompositeFormat	com.elastisys:autoscaler.distro.standard:5.2.2	0.80	0.83	0.83	0.76			
27	Ascii	io.bitsensor.plugins:apiconnector-bitsensor:3.1.0	0.77	0.84	0.51	0.91		0.92	
28	UTF8UrlDecoder	org.kill-bill.billing.plugin.java:bridge-plugin:0.2.5	0.78	0.81	0.76	0.79		0.81	
29	HtmlTokenizer	com.tr8nhub:core:0.1.0	0.62	0.62	0.00			1.00	
30	QueueInfoType	com.voximplant:apiclient:1.0.1	0.42	0.42	0.41			0.90	
31	Settings_impl	org.apache.uima:uimaj-core:3.1.1	0.30	0.31	0.34				0.23
32	SipHashFunction	org.voltdb:voltdbclient:9.2.2	0.70	0.71	0.68			0.74	
33	SignatureSpi	org.bouncycastle:bcprov-debug-jdk15to18:1.64	0.42	0.42	0.57				0.06
34	ArrayTransporter	com.aceql:aceql-http:6.1	0.83	0.87	0.83			0.83	
35	NtlmAuthenticationScheme	com.refinitiv.eta:eta:3.6.0.0	0.24	0.24	0.38				0.14
36	TokenMgrError	com.wenyu7980:el:1.0.0	0.97	1.00	0.96			0.70	
37	WebResource	org.httpunit:httpunit:1.7.3	0.50	0.50	0.00			0.80	
38	TimePatternConverter	org.apache.camel:camel-core-catalog:3.6.0	0.55	0.55	0.61				0.20
39	DArray	de.openea:eva2:2.2.0	0.68	0.68	0.74				0.06
40	TokenMgrError	org.eclipse.rdf4j:rdf4j-client:3.0.0	1.00	1.00	0.85			0.80	
41	ModbusUDPListener	com.ghgande:j2mod:2.7.0	0.16	0.20	0.34				0.07
42	ByteHashMap	org.kafkacrypto:kafkacrypto-java:0.9.9.7	0.69	0.75	0.67			0.80	
43	FileExtFilter	com.github.becauseQA:becauseQA-utils-windows:1.0.3	0.31	0.32	0.00			0.70	
44	ClientDebugDisplayCallable	com.alibaba.pelican:PelicanDT:1.0.10	0.02	0.02	0.04				0.14
45	CompressedInputStreamStorage	org.jpmml:jpmml-python:1.0.8	0.16	0.16	0.25				0.22

the 10 runs for each class. Finally, the right-hand side of the table presents the $\hat{A}_{12}$ estimates for the comparisons of *DivGen* versus *MonoGen*$_D$ as well as *DivGen* versus *MonoGen*$_N$.

From the $\hat{A}_{12}$ estimates in Table 6 we can observe that *DivGen*$_{2n}$ is significantly better than *MonoGen*$_{2n}$ in 33 out of 120 classes (i.e., 27.5% of the classes). *MonoGen*$_{2n}$, however, is only significantly better than *DivGen*$_2n$ in 8 out of 120 classes (i.e., 6.6%). We observe no significant difference between *DivGen*$_{2n}$ and *MonoGen*$_{2n}$ for the remaining 79 classes. For those 33 classes for which *DivGen*$_{2n}$ is significantly better than *MonoGen*$_{2n}$, we observe an average improvement of 16% in mutation score (median 13%). *MonoGen*$_{2n}$, on the other hand, only achieved an average improvement of 8% for the 8 classes it significantly improved (median 7%).

6.4. RQ1 (b): How does *DivGen*$_{2n}$ perform compared to *MonoGen*$_2$ on mutation score?

Having compared *DivGen*$_{2n}$ to *MonoGen*$_{2n}$ using the same time budget, we now compare *DivGen*$_{2n}$ to *MonoGen*$_2$ which uses the default time budget of EvoSuite. From the $\hat{A}_{12}$ estimates in Table 6 we can observe that *DivGen*$_{2n}$ is significantly better than *MonoGen*$_2$ in 28 out of 120 classes (i.e., 23.3% of the classes).

6.5. RQ2 (a): How does *DivGen*$_{2n}$ perform compared to *MonoGen*$_{2n}$ on branch coverage?

As with RQ1, in RQ2 we assess whether *DivGen*$_{2n}$ performs better, equally or worse than *MonoGen*$_2$ and *MonoGen*$_{2n}$ with respect to branch coverage as measured by JaCoCo on the set of sampled classes. For this analysis, we identified significant differences between the two approaches based on the same criteria for statistical significance determined by the p-values and the $\hat{A}_{12}$ estimate for effect size. Accordingly, similarly to the layout of Table 6, Table 7 combines the results obtained for the three approaches for branch coverage for each sampled class with $\hat{A}_{12} \neq 0.5$ where p-value ≤ 0.05.

At first glance, the magnitude of the significant differences between the pair-wise comparisons of the approaches appear to be less than for mutation score (45 differences versus 22 differences in branch coverage). Moreover, if we compare the significant differences of mutation coverage to branch coverage, we can see

Table 7
Mean branch coverage achieved for each class with $\hat{A}_{12} \neq 0.5$ where p-value ≤ 0.05.

	Class	Project	Branch coverage			$\hat{A}_{12}$ *DivGen*$_{2n}$			
			MonoGen$_2$	*DivGen*$_{2n}$	*MonoGen*$_{2n}$	vs. *MonoGen*$_2$		vs. *MonoGen*$_{2n}$	
						>0.5	<0.5	>0.5	<0.5
1	CharTypes	com.fasterxml.jackson.jr:jackson-jr-all:2.12.0-rc1	0.94	0.95	0.93	0.75		0.72	
2	PoolingConnectionProvider	io.logspace:logspace-jvm-agent:0.3.1	0.69	0.80	0.81	0.78			
3	NodeBuilder	io.fabric8:kubernetes-model-core:4.12.0	0.01	0.46	0.48	0.83			
4	TokenMgrError	com.wenyu7980:el:1.0.0	0.98	1.00	0.98	0.70			
5	Result	cn.detachment:detachment-core:1.0.6-RELEASE	0.87	0.96	0.85	0.90		0.90	
6	TokenMgrError	org.eclipse.rdf4j:rdf4j-client:3.0.0	0.93	0.97	0.86	0.82			
7	StringUtil	com.telnyx.sdk:telnyx:2.1.1	0.96	1.00	0.68	0.70		0.75	
8	HtmlTokenizer	com.tr8nhub:core:0.1.0	0.87	0.87	0.00			1.00	
9	Settings_impl	org.apache.uima:uimaj-core:3.1.1	0.38	0.39	0.46				0.04
10	AllowedFlexVolumeBuilder	io.fabric8:kubernetes-model:4.9.2	0.94	0.94	1.00				0.10
11	AlertsExternalServiceCondition	com.ocadotechnology...:newrelic-api-client:3.4.1	0.95	0.95	0.99				0.14
12	SignatureSpi	org.bouncycastle:bcprov-debug-jdk15to18:1.64	0.78	0.78	0.85				0.14
13	NtlmAuthenticationScheme	com.refinitiv.eta:eta:3.6.0.0	0.36	0.36	0.73				0.10
14	TrueMovingImage	org.tros:docking-frames-core:1.1.2-P19c	0.85	0.86	0.94				0.25
15	TimePatternConverter	org.apache.camel:camel-core-catalog:3.6.0	0.97	0.97	1.00				0.20
16	VolumeDeviceBuilder	io.fabric8:kubernetes-model:4.9.2	0.94	0.94	1.00				0.20
17	DArray	de.openea:eva2:2.2.0	0.96	0.96	0.99				0.00
18	TemporalComparator	org.apache.geode:geode-core:1.13.0	0.82	0.82	1.00				0.15
19	ModbusUDPListener	com.ghgande:j2mod:2.7.0	0.25	0.34	0.65				0.03
20	IndentationFilter	de.sstoehr:css-prettifier:1.0.1	0.55	0.55	0.63				0.09
21	JSON	io.fabric8:kubernetes-model:4.9.2	0.77	0.77	0.74			0.70	
22	FileExtFilter	com.github.becauseQA:becauseQA-utils-windows:1.0.3	0.40	0.40	0.00			0.70	
23	ClientDebugDisplayCallable	com.alibaba.pelican:PelicanDT:1.0.10	0.04	0.04	0.27				0.07
24	NamedParametersParser	io.github.mfvanek:pg-index-health:0.3.0	0.97	0.97	0.98				0.30
25	DefaultOption	org.springframework.data:spring-data-cassandra:3.1.0	0.74	0.74	0.77				0.23
26	CompressedInputStreamStorage	org.jpmml:jpmml-python:1.0.8	0.37	0.37	0.57				0.04

that *DivGen*$_{2n}$ only performs significantly better than *MonoGen*$_{2n}$ in 6 out of 120 classes (i.e., 5% of the classes). *MonoGen*$_{2n}$, however, outperforms *DivGen*$_{2n}$ in 16 out of 120 classes (i.e., 12.4% of the classes), while the remaining 98 classes have no significant differences.

Since the number of significant differences achieved is much smaller for branch coverage than for mutation score, we analyzed branch coverage in greater detail for each of the approaches. We discovered that in contrast to mutation score, high branch coverage measures are already achieved by the default variant of EvoSuite (i.e., *MonoGen*$_2$) using a default time budget of 2 min. The measurements for our sampled classes suggest high branch coverage is easier to achieve than mutation score, even with the default time budget of 2 min. *MonoGen*$_2$ achieves a median of 82% which implies that half of the classes have a branch coverage equal to or greater than 82%. The third quartile (i.e., .75) is even at the ideal branch coverage level, meaning that a quarter of the classes already have ideal branch coverage which neither of the alternative approaches can further improve. From this, it follows that since the branch coverage achieved by *MonoGen*$_2$ is already high, there is smaller room for improvements for the alternative approaches which use higher time budgets (which is what we observe).

For those 6 classes for which *DivGen*$_{2n}$ is significantly better than *MonoGen*$_{2n}$, we observe an average improvement of 24% in branch coverage (median 11%). *MonoGen*$_{2n}$, on the other hand, achieved an average improvement of 10% for the 16 classes it significantly improved (median 6%).

6.6. RQ2 (b): How does *DivGen*$_{2n}$ perform compared to *MonoGen*$_2$ on branch coverage?

Finally, we compared *DivGen*$_{2n}$ to *MonoGen*$_2$, based on significant differences in branch coverage. We observe that *DivGen*$_{2n}$ performs significantly better than *MonoGen*$_2$ in 7 out of 120 classes (i.e., 5.8%) with an average improvement of 9% (median 4%) for branch coverage. Again, as for the magnitude of significant differences obtained for the comparison of *DivGen*$_{2n}$ and *MonoGen*$_{2n}$, *MonoGen*$_2$ already achieved high branch coverage levels for a majority of the randomly sampled classes, so *DivGen*$_{2n}$ was not able to improve upon those for a significant number of classes.

7. Discussion

In this section we discuss the results presented in the previous section and draw conclusions about the viability of diversity-driven test generation as a strategy for enhancing the quality of the tests obtainable using EvoSuite. First we discuss the significance of the results and what insights they provide. Then we discuss the generalizability of the results in terms of the threats to their validity. Finally, we discuss potential limitations of the work and identify various ways they can be addressed in the future.

7.1. Significance of the results

The most important results of the experiment are of course the comparison between *DivGen*$_{2n}$ and *MonoGen*$_{2n}$ since the compared scores were created using the same time budgets. In the case of mutation score, the results appear to validate our hypothesis, since *DivGen*$_{2n}$ outperforms *MonoGen*$_{2n}$ for a significantly higher number of cases than when *MonoGen*$_{2n}$ outperforms *DivGen*$_{2n}$, with an average improvement of 16% in mutation score. However, in the case of branch coverage the situation is reversed, with *MonoGen*$_{2n}$ outperforming *DivGen*$_{2n}$ more times than the other way round. Our hypothesis would therefore appear to be only valid for mutation score.

At first sight this may be a surprising result, but in fact it reflects a deeper truth about the effectiveness of EvoSuite's algorithms and fitness functions for different tasks relative to the information they have to work with. In the case of branch coverage, the challenge of the AUTG tool is to generate tests to achieve the highest possible coverage of the actual CUT in hand. In other words, branch coverage is an AUTG generation challenge that focuses exclusively on one, and only one specific **implementation**, namely, the CUT in hand. It should not be surprising therefore

that $MonoGen_{2n}$ delivers the best results for branch coverage as it devotes its total, extended time budget (compared to $MonoGen_2$) to just that task.

In the case of mutation coverage, the challenge of the AUTG is rather to generate tests that achieve the highest possible coverage of the functionality delivered by the CUT rather than the one specific algorithm used in the CUT in hand. In other words, mutation coverage is an AUTG generation challenge that focuses more on the **functional abstraction** realized by the CUT in hand, rather on the idiosyncrasies of its particular implementation. Mutation score therefore tends to be favored by techniques that take a more interface-based, input-oriented approach whereas branch coverage tends to be enhanced more by techniques that take more implementation-based, structure oriented approach. The fact that $DivGen_{2n}$ outperforms $MonoGen_{2n}$ for mutation score is therefore to be expected and reinforces the assumption that it is the extra domain knowledge wrapped up in the harvested implementations that enables the enhancement. Whereas $MonoGen_{2n}$ focuses all of its extended time budget on analyzing the specific implementation of the CUT, $DivGen_{2n}$ focuses most of its extended time budget on analyzing alternative implementations of the functional abstraction realized by the CUT.

A more detailed explanation of why $DivGen_{2n}$ outperforms $MonoGen_{2n}$ for mutation score may be attributable to the additional domain information embedded in the code of the alternative implementations. From a larger set of implementations, EvoSuite has a higher chance to find "good inputs" which not only cover and execute a certain statement, but which also cause that statement to exhibit and propagate different behavior. For example, if the statement $X = Y + Z$ is mutated to $X = Y - Z$ in a mutant, a test that executes this statement with $Z = 0$ covers the mutated statement but will not kill the mutant. A test which executes the statement with $Z = 5$, for instance, will also kill the mutant. So apart from satisfying the pre-condition of covering the control-flow of a mutant (sometimes referred to as weak mutation (Rojas et al., 2015)), we still need to provide "good inputs" which achieve an infection of state which results in different observable behavior (strong mutation).

It is also worth noting that the enhanced mutation scores achieved by $DivGen_{2n}$ are noticeably larger than the enhanced branch coverage scores achieved by $MonoGen_{2n}$. More specifically, the number and average size of $DivGen_{2n}$'s outperformances of $MonoGen_{2n}$ in mutation score is much larger than the number and average size of $MonoGen_{2n}$'s outperformances of $DivGen_{2n}$ in branch coverage (33 with an average of 16% for the former versus 16 with an average of 10% for the latter). Of the two metrics, mutation score is arguably the more important one since it measures the ability of tests to reveal deviant behavior rather than to just execute the fault causing it.

A secondary important result of the experiment is that mainstream software repositories such as Maven Central clearly contain a sufficient number of harvestable, diverse implementations of a significant range of functional abstractions to be able to drive a diversity-based test generation approach such as *DivGen*'s. The proportion of originally sampled classes for which tests could be generated using *DivGen* was 83.8%, which is enough to support a reasonable likelihood of success, especially when *DivGen* is used in the background as a test recommendation engine.

7.2. Threats to validity

Threats to the *construct validity* of experiments relate to the extent to which the measurements made in the experiment are accurate and reflect accepted theory. As in most other studies of search-based AUTGs (often referred to as genetic algorithm based), we compare different algorithms (*DivGen* and *MonoGen*) based on widely used coverage criteria, in this case branch coverage and mutation coverage. These coverage criteria and metrics are mature and well established indicators of test set quality.

Threats related to the *internal validity* of experiments includes any bias or factors which affect the results. We carefully compared the significance of the results achieved by the approaches by applying rigorous statistical analysis, which involved executing each approach on each randomly sampled class 10 times. This approach is used in other related studies of AUTG tool performance such as Panichella et al. (2017) in order to take into account the non-deterministic nature of genetic algorithms (Arcuri and Briand, 2014). Furthermore, despite the enhanced search budget available to generate tests using EvoSuite, we rely on the default settings of the parameters available in EvoSuite as suggested by Arcuri and Fraser (2011) (cf. Section 5.2.1), except where we have to extend the time budget of EvoSuite to achieve a fair comparison to *DivGen*. The measurements of the coverage for each approach are obtained using the execution environment provided by LASSO. Since this can be carefully controlled, we can ensure that any class is executed under the same environmental conditions and measured by the same tools for every approach. To actually measure the coverage criteria we use the widely used JaCoCo tool for obtaining branch coverage and PIT tool to measure mutation score.

In order to establish a fair baseline for the comparisons of $DivGen_{2n}$ and $MonoGen_{2n}$, we made sure that both approaches had the same amount of time available (i.e., search budget) to generate tests. To mitigate the risk of unfair comparisons, in the event that $DivGen_{2n}$ was unable to harvest the full 15 alternative implementations allowed, we adapted the time budget assigned to $MonoGen_{2n}$ accordingly. Since we randomly sampled classes from a non-curated corpus, we faced a variety of practical (technical) challenges and "unknowns" (cf. Section 5.5). We addressed these threats by using sensible timeouts as recommended by EvoSuite as well as the coverage tools and by applying additional sanity checks to increase the confidence of the implementations' deterministic behavior.

An additional risk related to unfair comparisons arises from the fact that $MonoGen_{2n}$ can be realized in at least one alternative way. Instead of increasing the time budget in one run as we did, we may choose to re-run $MonoGen_2$ several times (up to the maximum time budget). We do not believe, however, that this risk is a severe threat to our study. Since we assume a practical setting, the option of simply increasing the time budget is a natural choice (and is even an explicit parameter choice in EvoSuite), instead of repeating *MonoGen* several times and merging test sets manually.

Another potential threat to the internal validity of our experiment is related to our observation that $MonoGen_{2n}$ sometimes performs worse than $MonoGen_2$. We believe that this observation has two main causes. The first is that the inherent non-determinism of search-based algorithms, and their initial seed tests have a significant influence on their "optimized" results (Fraser and Arcuri, 2012a). Accordingly, a longer time budget does not always lead to a better set of generated tests than a shorter time budget (i.e., the former may be unlucky, whereas the latter may be lucky). Note that this effect is the core reason why the standard practice in (search-based testing) experiments is to perform multiple executions of the tool under investigation on each sample (Arcuri and Briand, 2014). The second reason is that we observed that EvoSuite has a non-trivial probability of failing (i.e., it fails to generate a non-empty set of tests for some reason). Other experiments on EvoSuite do not usually include such test generation runs in the final statistics, but we include them by recording the score of zero for an empty test set. Since we want to measure the practical performance users are likely to experience in real projects, we do this because this is obviously

the true mutation score and branch coverage of what is delivered in a failed test generation run (i.e., nothing is delivered). Unfortunately, EvoSuite's execution log only provides short, cryptic error messages about the reason for the failure, which are very much related to its internal implementation (e.g., inconsistent goals detected, incomplete statistics like missing data points or no tests were generated).

Due to the characteristics of *DivGen*, there is a risk that the *Harvest* function may retrieve class duplicates which are not sufficiently "diverse", leading to the situation that clones of the target class or some of its alternative classes are executed several times using $MonoGen_2$. Apart from Carzaniga et al.'s work on measuring the redundancy (i.e., realizational distinctiveness) of single software systems automatically (Carzaniga et al., 2015), no automated diversity measurement approaches have been published to date. To mitigate the risk of retrieving alternative implementations which may not be "realizationally" distinct, we used Nicad to reject syntactical code clones (type-1 and type-2 clones). By manual inspection of the harvested class implementations we noticed that a non-trivial proportion of class implementations are likely type-3 clones, and thus highly similar (or simply older or newer versions of the randomly sampled class). But since type-3 clones are arguably realizationally distinct (i.e., pair-wise diverse with potential modifications and may exhibit different observable behavior; cf. Roy and Cordy (2007)), we do not attempt to detect and reject them. As discussed in Section 6.2, to gain some insights into the typical diversity of the harvested alternative implementations we analyzed the spreads of six key properties related to the implementations' structure/size and origin/purpose. Intuitively, the results suggest fairly high levels of diversity, since the key property measurements varied significantly among the sets of implementations.

We used rigorous statistical testing in order to mitigate the risk of threats to *conclusion validity* (i.e., treatment versus outcome). As in more recent studies like (Panichella et al., 2017), we complement Wilcoxon hypothesis testing using p-values (in our case Mann–Whitney-U) with effect size estimates by Vargha–Delaney which indicate the magnitude of the differences observed. Both are non-parameterized statistical analyses which make no assumptions about the distribution of the underlying measures (as advised by Arcuri and Briand (2014)). This ensures that we focus only on classes for which we have obtained significant differences between two approaches, and for which we could identify a significant difference using the effect size estimates by Vargha–Delaney. If differences were not identified as significant by our two statistical analyses, they were not used in the derivation of our conclusions.

Threats to the generalizability of our results (i.e., *external validity*) are related to the subjects (here classes) that participated in the study. By using the strategy of fully randomly sampling, we sampled 120 Java classes from a large, non-curated, real-world repository, Maven Central, which contains classes from popular Java projects and libraries. Our selection criteria achieved a high diversity with respect to the projects from which they were obtained, and made sure that the classes sampled have non-trivial functionality based on their structural complexity (i.e., at least 10 branches), so that test generation is sufficiently challenging for $MonoGen_2$ (i.e., in order to be able to observe differences). To further strengthen the generalizability of our results, we compared $DivGen_{2n}$ to both $MonoGen_2$ (using the default EvoSuite time budget) as well $MonoGen_{2n}$ (using the same time budget as $DivGen_{2n}$ for each sampled class).

Since the adaptation of custom "types" defined by projects is still a major research challenge, a threat to the generalizability of our study results is that we deliberately limited the samples to classes which have method signatures that only use Java-compatible types (classes and types that are part of the JDK). Furthermore, since the realization of *DivGen* was rudimentary, advanced features like mocking support in testing Java classes were not supported.

7.3. Future research directions

The conclusions we draw from the analysis of the results obtained for diversity-driven test generation open multiple future research paths. In this work, we realized *DivGen* in a rudimentary way as a "proactive" approach which takes a given time budget and attempts to generate effective tests. There are, however, many other potential ways in which such a diversity-driven test generation approach may be realized. First, instead of mining code and then generating tests from them using $MonoGen_2$, one could simply mine existing tests directly (i.e., as a reactive approach). The costs in this case would dramatically decrease, since it would obviate the generation of new tests. The main effort would be the mining of test sequences and their adaption to the target interface. Alternatively, if AUTG tools were applied on a larger basis as part of continuous integration, one could collect and exploit previously generated tests in order to feed them into *DivGen* (cf. Campos et al., 2014).

As explained previously, since the main goal of the study was to establish the practical feasibility of the approach the current prototype implementation is rather rudimentary and the recognized tuning parameters were set to basic values. These include (1) the number of alternative implementations to retrieve, (2) the number of adapter mappings to use for diversity-driven test generation, and, (3) the time budget available for *MonoGen* for each of the implementations. For the first parameter, we selected 15 as a reasonable number of alternative implementations to work with based on our past experience with code search engines (Merobase and LASSO). For the second parameter which defines how many adapter mappings the algorithm should attempt to apply, we simply selected one to test the feasibility of the approach. However, higher values could significantly increase the number of tests generated without requiring more alternative implementations to be harvested. And for the third parameter, we selected the default value wherever possible since this has been shown to be a reasonable approach. There is obviously significant potential to fine-tune these parameters of *DivGen* and potentially new ones, and to identify which combinations provide the best results. This is ultimately a multi-objective optimization problem of the kind addressed by AUTG tools, so the techniques from search-based algorithms are likely to be useful.

In this study our prototype implementation used interface-driven code search technology to harvest alternative classes based on similar interface signatures. However, NLP-like searches have fairly low precision since it is impossible, in general, to determine the functional equivalence of software algorithmically due to Rice's theorem (Rice, 1953). The best way to estimate the functionality of a class in practice, therefore, is to execute it using behavior sampling techniques (Podgurski and Pierce, 1993). An interesting research direction is therefore to replace the interface-driven code search of *Harvest* with more precise approaches such as test-driven code search where alternative implementations are filtered based on their observable behavior at run-time (e.g., by the number of tests which pass on both the target class and the alternative ones). Arguably, adding additional tests from any of the harvesting strategies will not hurt, but nevertheless they may not increase the quality of the returned tests, and thus will not improve defect detectability. Using behavior sampling, one can ensure that alternative implementations share the same domain knowledge. This makes it possible to retrieve alternative implementations encoding additional domain information that can be exploited as part of the test generation process.

Another interesting research direction is to collect and analyze the "expected" behavior of alternative implementations. EvoSuite records the observed behavior of SUTs in response to the stimuli it generates at run-time and encodes the behavior in the assertion statements of JUnit classes to enable regression testing. One may exploit such observable behavior in order to mine pseudo-oracles (cf. automatic test oracles (Langdon et al., 2017)) which may also be used and collected as part of the diversity-driven test generation process.

Finally, our current prototype does not attempt to minimize the set of tests returned by $DivGen_{2n}$ in the same way that sophisticated AUTG tools like EvoSuite do. This was not felt to be necessary in this feasibility study, but would be important functionality if the approach should ever be used in practice. The current implementation simply identifies duplicate tests based on the underlying test sequence abstraction and keeps only one. An important future step in the evolution of *DivGen* is to include test set reduction techniques in order to assess whether *DivGen* can also match the test set effectiveness of existing approaches like EvoSuite. An interesting idea here is to investigate whether diversity opens up new possibilities for test suite minimization criteria.

8. Conclusion

The impulse for the study presented in this paper is the convergence of three ongoing trends in mainstream software engineering - (1) the growth of large scale, online software repositories containing vast swathes of rich software functionality, (2) the maturing of sophisticated search and mining technologies which can retrieve software with desired properties and (3) the adoption of agile and continuous development practices and powerful integration platforms to support them. Together, the first two have made the reuse of class-scale software components feasible in practical software engineering projects, while the third has opened up the prospect of running resource intensive search-based technologies in the background (e.g., at night) in the style of a recommendation service rather than proactively executed functionality. AUTG is a prime target for exploiting this technology because it is (1) an extremely important capability for the engineering of high quality software, (2) it is a resource intensive search process whose results are strongly effected by time budgets and (3) current AUTG tools like EvoSuite face a very challenging fitness landscape which makes it increasingly difficult to tease out extra performance without additional information and parameters to optimize.

As reported in this work we have been able to build a prototype implementation of a diversity-driven AUTG tool, *DivGen*, for class-scale test set generation, which was able to harvest sufficient diversity to generate higher quality test sets than EvoSuite, the leading AUTG tool for Java, as measured by mutation score. The fact that, *DivGen* uses EvoSuite to achieve these performance enhancements demonstrates the synergy between the two technologies and how the form of "search technology" that drives EvoSuite can be complemented by the form of "search technology" that drives the harvesting of component in *DivGen* (Atkinson et al., 2013). The main challenge for the future is to reverse the usage relationship of the two technologies. In our current prototype implementation, EvoSuite is invoked as a black box component of *DivGen*. However, even better performance enhancements are likely to be gained in the future by making *DivGen*, or at least the *Harvest* component that drives it, an internal component of EvoSuite. Only then will the full power of search based algorithms and heuristics be applicable to all the extra information and performance parameters made feasible by exploiting software diversity.

Finally, although the current implementation of the *DivGen* approach proposed in this work provides the traditional AUTG service, without the inclusion of any trustworthy oracle values, the availability of multiple alternative implementations also opens up the possibility of automatically generating more trustworthy oracle information using the cross-checking oracle approach (Carzaniga et al., 2014). However, this will require higher confidence that harvested implementations are true implementations of the functional abstraction in hand. Until then, the main use cases for *DivGen* generated test sets are likely to be in regression testing, as with current AUTG tools like EvoSuite.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

Abdalkareem, R., Shihab, E., Rilling, J., 2017. On code reuse from StackOverflow: An exploratory study on Android apps. Inf. Softw. Technol. 88, 148–158. http://dx.doi.org/10.1016/j.infsof.2017.04.005, URL: http://www.sciencedirect.com/science/article/pii/S0950584917303610.

Agitar, 2021. AgitarOne. URL: http://www.agitar.com/solutions/products/agitarone.html. (accessed 2021-04-01).

Albunian, N.M., 2017. Diversity in search-based unit test suite generation. In: Menzies, T., Petke, J. (Eds.), Search Based Software Engineering. Springer International Publishing, Cham, pp. 183–189.

Anand, S., Burke, E.K., Chen, T.Y., Clark, J., Cohen, M.B., Grieskamp, W., Harman, M., Harrold, M.J., McMinn, P., Bertolino, A., et al., 2013. An orchestrated survey of methodologies for automated software test case generation. J. Syst. Softw. 86 (8), 1978–2001.

Andrews, J.H., Briand, L.C., Labiche, Y., Namin, A.S., 2006. Using mutation analysis for assessing and comparing testing coverage criteria. IEEE Trans. Softw. Eng. 32 (8), 608–624.

Arcuri, A., Briand, L., 2011. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In: Proceedings of the 33rd International Conference on Software Engineering. In: ICSE '11, Association for Computing Machinery, New York, NY, USA, pp. 1–10. http://dx.doi.org/10.1145/1985793.1985795.

Arcuri, A., Briand, L., 2014. A Hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. Softw. Test. Verif. Reliab. 24 (3), 219–250. http://dx.doi.org/10.1002/stvr.1486, URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/stvr.1486.

Arcuri, A., Fraser, G., 2011. On parameter tuning in search based software engineering. In: Cohen, M.B., Ó Cinnéide, M. (Eds.), Search Based Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 33–47.

Atkinson, C., Kessel, M., Schumacher, M., 2013. On the synergy between search-based and search-driven software engineering. In: Ruhe, G., Zhang, Y. (Eds.), Search Based Software Engineering. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 239–244.

Avizienis, A., 1985. The N-version approach to fault-tolerant software. IEEE Trans. Softw. Eng. (12), 1491–1501.

Bajracharya, S., Ossher, J., Lopes, C., 2014. Sourcerer: An infrastructure for large-scale collection and analysis of open-source code. Sci. Comput. Program. 79, 241–259. http://dx.doi.org/10.1016/j.scico.2012.04.008, URL: http://www.sciencedirect.com/science/article/pii/S016764231200072X. Experimental Software and Toolkits (EST 4): A special issue of the Workshop on Academic Software Development Tools and Techniques (WASDeTT-3 2010).

Baldoni, R., Coppa, E., D'elia, D.C., Demetrescu, C., Finocchi, I., 2018. A survey of symbolic execution techniques. ACM Comput. Surv. 51 (3), http://dx.doi.org/10.1145/3182657.

Boettiger, C., 2015. An introduction to docker for reproducible research. SIGOPS Oper. Syst. Rev. 49 (1), 71–79. http://dx.doi.org/10.1145/2723872.2723882, URL: http://doi.acm.org/10.1145/2723872.2723882.

Campos, J., Arcuri, A., Fraser, G., Abreu, R., 2014. Continuous test generation: Enhancing continuous integration with automated test generation. In: Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering. In: ASE '14, Association for Computing Machinery, New York, NY, USA, pp. 55–66. http://dx.doi.org/10.1145/2642937.2643002.

Carzaniga, A., Goffi, A., Gorla, A., Mattavelli, A., Pezzè, M., 2014. Cross-checking oracles from intrinsic software redundancy. In: Proceedings of the 36th International Conference on Software Engineering. In: ICSE 2014, ACM, New York, NY, USA, pp. 931–942. http://dx.doi.org/10.1145/2568225.2568287, URL: http://doi.acm.org/10.1145/2568225.2568287.

Carzaniga, A., Mattavelli, A., Pezzè, M., 2015. Measuring software redundancy. In: Proceedings of the 37th International Conference on Software Engineering - Volume 1. In: ICSE '15, IEEE Press, Piscataway, NJ, USA, pp. 156–166, URL: http://dl.acm.org/citation.cfm?id=2818754.2818776.

Chen, L., Avizienis, A., 1978. N-version programming: A fault-tolerance approach to reliability of software operation. In: Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8), Vol. 1. pp. 3–9.

Coles, H., Laurent, T., Henard, C., Papadakis, M., Ventresque, A., 2016. PIT: A practical mutation testing tool for Java (demo). In: Proceedings of the 25th International Symposium on Software Testing and Analysis. In: ISSTA 2016, Association for Computing Machinery, New York, NY, USA, pp. 449–452. http://dx.doi.org/10.1145/2931037.2948707.

Conover, W.J., 1999. Practical Nonparametric Statistics, Vol. 350. john wiley & sons.

Cordy, J.R., Roy, C.K., 2011. The NiCad clone detector. In: 2011 IEEE 19th International Conference on Program Comprehension. pp. 219–220.

De Paula, A.C., Guerra, E., Lopes, C.V., Sajnani, H., Lazzarini Lemos, O.A., 2016. An exploratory study of interface redundancy in code repositories. In: 2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 107–116. http://dx.doi.org/10.1109/SCAM.2016.31.

Deb, K., 2011. Multi-objective optimisation using evolutionary algorithms: an introduction. In: Multi-Objective Evolutionary Optimisation for Product Design and Manufacturing. Springer, pp. 3–34.

Dietrich, J., Schole, H., Sui, L., Tempero, E., 2017. XCorpus - An executable corpus of Java programs. J. Object Technol. 16 (4), 1:1–24. http://dx.doi.org/10.5381/jot.2017.16.4.a1.

Dyer, R., Nguyen, H.A., Rajan, H., Nguyen, T.N., 2015. Boa: Ultra-large-scale software repository and source-code mining. ACM Trans. Softw. Eng. Methodol. 25 (1), 7:1–7:34. http://dx.doi.org/10.1145/2803171, URL: http://doi.acm.org/10.1145/2803171.

Frakes, W.B., Kang, K., 2005. Software reuse research: status and future. IEEE Trans. Softw. Eng. 31 (7), 529–536. http://dx.doi.org/10.1109/TSE.2005.85.

Fraser, G., 2018. A tutorial on using and extending the EvoSuite search-based test generator. In: Search-Based Software Engineering. Springer, pp. 106–130.

Fraser, G., Arcuri, A., 2011. 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. In: ESEC/FSE '11, ACM, New York, NY, USA, pp. 416–419. http://dx.doi.org/10.1145/2025113.2025179, URL: http://doi.acm.org/10.1145/2025113.2025179.

Fraser, G., Arcuri, A., 2012a. The seed is strong: Seeding strategies in search-based software testing. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation. IEEE, pp. 121–130.

Fraser, G., Arcuri, A., 2012b. Whole test suite generation. IEEE Trans. Softw. Eng. 39 (2), 276–291.

Fraser, G., Arcuri, A., 2013. EvoSuite: On the challenges of test case generation in the real world. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation. pp. 362–369. http://dx.doi.org/10.1109/ICST.2013.51.

Fraser, G., Arcuri, A., 2014. A large-scale evaluation of automated unit test generation using EvoSuite. ACM Trans. Softw. Eng. Methodol. 24 (2), 8:1–8:42. http://dx.doi.org/10.1145/2685612, URL: http://doi.acm.org/10.1145/2685612.

Fraser, G., Rojas, J.M., Campos, J., Arcuri, A., 2017. EvoSuite at the SBST 2017 tool competition. In: 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST). pp. 39–42.

Galeotti, J.P., Fraser, G., Arcuri, A., 2013. Improving search-based test suite generation with dynamic symbolic execution. In: 2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE). pp. 360–369.

Harman, M., Jones, B.F., 2001. Search-based software engineering. Inf. Softw. Technol. 43 (14), 833–839. http://dx.doi.org/10.1016/S0950-5849(01)00189-6, URL: https://www.sciencedirect.com/science/article/pii/S0950584901001896.

Hummel, O., 2008. Semantic Component Retrieval in Software Engineering (Ph.D. thesis). Universität Mannheim.

Hummel, O., Janjic, W., Atkinson, C., 2008. Code conjurer: Pulling reusable software out of thin air. IEEE Softw. 25 (5), 45–52. http://dx.doi.org/10.1109/MS.2008.110.

JaCoCo, 2021. JaCoCo. URL: https://www.jacoco.org/jacoco/. (accessed 2021-04-01).

Kessel, M., Atkinson, C., 2019a. Automatically curated data sets. In: 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 56–61. http://dx.doi.org/10.1109/SCAM.2019.00015.

Kessel, M., Atkinson, C., 2019b. On the efficacy of dynamic behavior comparison for judging functional equivalence. In: 2019 19th International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 193–203. http://dx.doi.org/10.1109/SCAM.2019.00030.

Kessel, M., Atkinson, C., 2019c. A platform for diversity-driven test amplification. In: Proceedings of the 10th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation. In: A-TEST 2019, ACM, New York, NY, USA, pp. 35–41. http://dx.doi.org/10.1145/3340433.3342825, URL: http://doi.acm.org/10.1145/3340433.3342825.

Kessel, M., Atkinson, C., 2022. Diversity-driven unit test generation (data set). http://dx.doi.org/10.5281/zenodo.6962676.

Krueger, C.W., 1992. Software reuse. ACM Comput. Surv. 24 (2), 131–183. http://dx.doi.org/10.1145/130844.130856, URL: http://doi.acm.org/10.1145/130844.130856.

Langdon, W.B., Yoo, S., Harman, M., 2017. Inferring automatic test oracles. In: 2017 IEEE/ACM 10th International Workshop on Search-Based Software Testing (SBST). pp. 5–6. http://dx.doi.org/10.1109/SBST.2017.1.

Lemos, O.A.L., Bajracharya, S.K., Ossher, J., Morla, R.S., Masiero, P.C., Baldi, P., Lopes, C.V., 2007. CodeGenie: Using test-cases to search and reuse source code. In: Proceedings of the Twenty-Second IEEE/ACM International Conference on Automated Software Engineering. In: ASE '07, ACM, New York, NY, USA, pp. 525–526. http://dx.doi.org/10.1145/1321631.1321726, URL: http://doi.acm.org/10.1145/1321631.1321726.

Lemos, O.A.L., de Paula, A.C., Zanichelli, F.C., Lopes, C.V., 2014. Thesaurus-based automatic query expansion for interface-driven code search. In: Proceedings of the 11th Working Conference on Mining Software Repositories. In: MSR 2014, ACM, New York, NY, USA, pp. 212–221. http://dx.doi.org/10.1145/2597073.2597087, URL: http://doi.acm.org/10.1145/2597073.2597087.

Maven, 2021. Maven. URL: http://maven.apache.org/. (accessed 2021-04-01).

McCabe, T.J., 1976. A complexity measure. IEEE Trans. Softw. Eng. SE-2 (4), 308–320. http://dx.doi.org/10.1109/TSE.1976.233837.

McMinn, P., 2011. Search-based software testing: Past, present and future. In: 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops. pp. 153–163.

Mili, H., Mili, F., Mili, A., 1995. Reusing software: issues and research directions. IEEE Trans. Softw. Eng. 21 (6), 528–562. http://dx.doi.org/10.1109/32.391379.

Mili, A., Mili, R., Mittermeir, R., 1998. A survey of software reuse libraries. Ann. Softw. Eng. 5 (1), 349–414. http://dx.doi.org/10.1023/A:1018964121953.

Miller, G.A., 1995. WordNet: A lexical database for english. Commun. ACM 38 (11), 39–41. http://dx.doi.org/10.1145/219717.219748.

Morrison, R.W., De Jong, K.A., 2002. Measurement of population diversity. In: Collet, P., Fonlupt, C., Hao, J.-K., Lutton, E., Schoenauer, M. (Eds.), Artificial Evolution. Springer Berlin Heidelberg, Berlin, Heidelberg, pp. 31–41.

Pacheco, C., Lahiri, S.K., Ernst, M.D., Ball, T., 2007. Feedback-directed random test generation. In: Proceedings of the 29th International Conference on Software Engineering. In: ICSE '07, IEEE Computer Society, USA, pp. 75–84. http://dx.doi.org/10.1109/ICSE.2007.37.

Palsberg, J., Lopes, C.V., 2018. NJR: A normalized java resource. In: Companion Proceedings for the ISSTA/ECOOP 2018 Workshops. In: ISSTA '18, ACM, New York, NY, USA, pp. 100–106. http://dx.doi.org/10.1145/3236454.3236501, URL: http://doi.acm.org/10.1145/3236454.3236501.

Panichella, A., Kifetew, F.M., Tonella, P., 2017. Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Trans. Softw. Eng. 44 (2), 122–158.

Podgurski, A., Pierce, L., 1992. Behavior sampling: A technique for automated retrieval of reusable components. In: Proceedings of the 14th International Conference on Software Engineering. In: ICSE '92, ACM, New York, NY, USA, pp. 349–361. http://dx.doi.org/10.1145/143062.143152, URL: http://doi.acm.org/10.1145/143062.143152.

Podgurski, A., Pierce, L., 1993. Retrieving reusable software by sampling behavior. ACM Trans. Softw. Eng. Methodol. 2 (3), 286–303. http://dx.doi.org/10.1145/152388.152392, URL: http://doi.acm.org/10.1145/152388.152392.

Raemaekers, S., van Deursen, A., Visser, J., 2013. The Maven repository dataset of metrics, changes, and dependencies. In: 2013 10th Working Conference on Mining Software Repositories (MSR). pp. 221–224. http://dx.doi.org/10.1109/MSR.2013.6624031.

Raemaekers, S., van Deursen, A., Visser, J., 2017. Semantic versioning and impact of breaking changes in the Maven repository. J. Syst. Softw. 129, 140–158. http://dx.doi.org/10.1016/j.jss.2016.04.008, URL: http://www.sciencedirect.com/science/article/pii/S0164121216300243.

Reiss, S.P., 2009. Semantics-based code search. In: Proceedings of the 31st International Conference on Software Engineering. In: ICSE '09, IEEE Computer Society, Washington, DC, USA, pp. 243–253. http://dx.doi.org/10.1109/ICSE.2009.5070525.

Rice, H.G., 1953. Classes of recursively enumerable sets and their decision problems. Trans. Amer. Math. Soc. 74 (2), 358–366, URL: http://www.jstor.org/stable/1990888.

Robillard, M., Walker, R., Zimmermann, T., 2010. Recommendation systems for software engineering. IEEE Softw. 27 (4), 80–86. http://dx.doi.org/10.1109/MS.2009.161.

Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A., 2015. Combining multiple coverage criteria in search-based unit test generation. In: Barros, M., Labiche, Y. (Eds.), Search-Based Software Engineering. Springer. Springer International Publishing, Cham, pp. 93–108.

Roy, C.K., Cordy, J.R., 2007. A survey on software clone detection research. In: Queen's School of Computing TR 541. pp. 64–68.

Sim, S.E., Gallardo-Valencia, R.E., 2013. Finding Source Code on the Web for Remix and Reuse. Springer.

Sonatype, 2021. Maven central. URL: http://search.maven.org. (accessed 2021-04-01).

Stolee, K.T., Elbaum, S., Dwyer, M.B., 2016. Code search with input/output queries: Generalizing, ranking, and assessment. J. Syst. Softw. 116, 35–48. http://dx.doi.org/10.1016/j.jss.2015.04.081, URL: http://www.sciencedirect.com/science/article/pii/S0164121215000874.

Vargha, A., Delaney, H.D., 2000. A critique and improvement of the CL common language effect size statistics of McGraw and wong. J. Educ. Behav. Stat. 25 (2), 101–132. http://dx.doi.org/10.3102/10769986025002101.

Wang, Y., Feng, Y., Martins, R., Kaushik, A., Dillig, I., Reiss, S.P., 2016. Hunter: Next-generation code reuse for Java. In: Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering. In: FSE 2016, ACM, New York, NY, USA, pp. 1028–1032. http://dx.doi.org/10.1145/2950290.2983934, URL: http://doi.acm.org/10.1145/2950290.2983934.

Wolpert, D., Macready, W., 1997. No free lunch theorems for optimization. IEEE Trans. Evol. Comput. 1 (1), 67–82. http://dx.doi.org/10.1109/4235.585893.