SOFTWARE TESTING, VERIFICATION AND RELIABILITY
Softw. Test. Verif. Reliab. 2016; **00**:1–34
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/stvr

Evaluating code-based test input generator tools

Lajos Cseppentő and Zoltán Micskei*

Budapest University of Technology and Economics, Department of Measurement and Information Systems, Hungary

SUMMARY

In recent years several tools have been developed to automatically select test inputs from the code of the system under test. However, each of these tools has different advantages, and there is little detailed feedback available on the actual capabilities of the various tools. In order to evaluate test input generators this paper collects a set of programming language concepts that should be handled by the tools, and maps these core concepts and challenging features like handling the environment or multi-threading to 363 code snippets respectively. These snippets would serve as inputs for the tools. Next, the paper presents SETTE, an automated framework to execute and evaluate these snippets. Using SETTE multiple experiments were performed on five Java and one .NET-based tools using symbolic execution, search-based and random techniques. The test suites' coverage, size, generation time and mutation score were compared. The results highlight the strengths and weaknesses of each tool and approach, and identify hard code parts that are difficult to tackle for most of the tools. We hope that this research could serve as actionable feedback to tool developers and help practitioners assess the readiness of test input generation. Copyright © 2016 John Wiley & Sons, Ltd.

Received . . .

KEY WORDS: software testing; test generation; white-box testing; test data

1. INTRODUCTION

Testing is one of the most commonly used techniques to check and improve the quality of software systems, where the system is executed under specified conditions defined by test cases. A test case should include "test inputs, execution conditions, and expected results developed for a particular objective" [1]. However, creating efficient and effective tests is a challenging and resource consuming task. That is why extensive research has been performed in the last decades to automatically derive the various test artifacts. For example, *model-based testing* methods can generate test cases from behavioral models. *Code-based methods* start from the source or binary code of the system under test and select test inputs according to some criteria, e.g. maximizing achieved code coverage. Code-based methods primarily generate only *test inputs* without expected outputs, and rely on assertions or exceptions to detect issues.[†]

Several techniques have been proposed for test input generation [2], e.g. *symbolic execution* [3], search-based software testing [4] or variants of random methods [5]. Test input generation is an active research topic with more and more tools developed. There exists tools for several platforms, such as C, .NET, Java or x86 machine code. However, the majority of these tools are academic or

*Correspondence to: H-1117 Budapest, Magyar tudosok krt. 2., Hungary. E-mail: micskeiz@mit.bme.hu

Contract/grant sponsor: This work has been partially supported by the CECRIS ("CErtification of CRItical Systems") FP7-Marie Curie (IAPP) project and by the ARTEMIS JU and the Hungarian National Research, Development and Innovation Fund in the frame of the R5-COP project.

[†]Note on similar terminology: Some papers use the terms *test data generation* and *white-box testing* for these concepts.

research prototypes. Although these methods are quite successful for some objectives (e.g. white-box fuzzing [6]), they are still not used generally by engineers.

As the problem at the heart of test input generation is computationally hard in general, the development of such tools faces several theoretical challenges. Additionally, as modern programming languages and platforms offer a wide variety of complex language constructs and features, supporting all of them is a significant development task. Thus the available test generators have varying capabilities.

Typically the publication of a new tool includes also experimental results to assess the tool's capabilities. However, some of the developers use their own sample programs, while others conduct case studies on open source software. The used third-party software are usually different, thus *comparing* the abilities of the tools is not trivial. The tool papers always describe the benefits of the new tool and the innovation carried out during development. Nonetheless, only some of them mention all limits of the tool. Several experiments [7, 8] have been published in which tools were applied to complex software. These experiments communicate aggregated quantitative results (such as average code coverage) in the first place and point out how the tools perform and handle the challenges in real situations. However, we only found one general survey [9] containing *fine-grained feedback* on what pieces of code can or cannot be handled by a certain tool. Thus the initial question that motivated this work was:

How can the different test input generator tools be compared and evaluated?

Designing a common evaluation method and framework would help to identify general challenges, would provide tool developers with actionable, reproducible feedback, and would help practitioners assess the readiness of the tools and test input generation more precisely.

The following *method* was applied in this research. First, the programming language concepts were collected and organized that should be handled by a test input generator (e.g. recursion, complex structures). Then, minimal code snippets were defined that target these features and serve as inputs for the test input generators. Next, the tools were executed on the above snippets, and based on the results of the test generation, it was possible to conclude whether the given tool handles a certain feature properly or not. Moreover, several other metrics including test suite size or generation time were analyzed. Using all the snippets, detailed feedback could be obtained, which makes possible to compare the tools.

Our initial paper [10] concentrated on tools using symbolic execution. This paper extends the scope of the work by including search-based and random testing; extending the features with handling environment, reflection, multi-threading and native code; adding a new tool and a previously manually evaluated tool to the automatic evaluation framework; performing a large number of experiments; and extending the analysis from only code coverage to several other properties.

Thus the *contributions* of this paper include:

- collecting core and challenging features of imperative languages w.r.t. test generation;
- mapping part of these features to the Java and .NET languages and platforms and implementing them in 363 code snippets;
- creating an extendable framework called SETTE for automatically evaluating the snippets on selected tools;
- designing and conducting a large number of experiments on five Java and one .NET tools;
- analyzing the results by highlighting the significant differences and the 'hard code parts', i.e. the features which are difficult to tackle for most of the tools.

The SETTE framework, the code snippets and all the experimental results are publicly available from the tool's website:

http://sette-testing.github.io

DOI: 10.1002/stvr

Table I. List of evaluated test input generator tools

Name	Platform	OS	Access	Published	Updated	Technique
CATG [23]	Java	–	open source	2012	2015	SE
EVOSUITE [24]	Java	–	open source	2008	2016	SBST
jPET [25]	Java	Linux	closed source	2009	2011	SE
PEX / INTELLITEST [26]	.NET	Windows	closed source	2008	2016	SE
RANDOOP [27]	Java	–	open source	2007	2016	RT
SPF [28]	Java	–	open source	2012	2016	SE

2. OVERVIEW

Structural or white-box testing is a well-known method since the early days of software testing [11]. Even in the 1970s several algorithms and tools were proposed for automatic test input generation starting from the source code [12, 13]. Since then numerous techniques have been developed (e.g. using mutation analysis [14] or pre-/postconditions [15], etc.). This work focuses on tools using symbolic execution, search-based and random testing.

Symbolic execution (SE) is a program analysis technique where *symbolic variables* are used instead of concrete inputs and an execution path in the program is represented with an expression over the symbolic variables (called a path condition or path constraint) [16]. The possible execution paths are collected and the respective path constraints are solved by usually an SMT solver yielding a set of concrete input values activating the given path in the program. Although the idea of symbolic execution was born in the 1970s, it has been used for test generation in practice only recently because of its high computational need [3].

Search-based software testing (SBST) is an area of search-based software engineering, i.e. the application of metaheuristic search techniques to optimization problems found in software engineering. In SBST the test input generation is formulated as a search problem: possible inputs to the program form a search space and the test adequacy criterion is coded as a fitness function [17].

Variants of *random testing* (RT) have also been used for test input generation. Even in its simple forms, random testing could be useful in practice, e.g. to find robustness failures [18]. More recently, adaptive random testing and feedback-directed testing have been proposed as extensions.

Note that the distinction between these techniques are often blurred, e.g. in symbolic execution various search techniques could be used to select the next path, and these techniques could be combined in hybrid approaches [2].

Challenges However, like other hard problems, the practical application of test input generation faces also several *challenges* [2]. Common important ones to deal with include path explosion (exponentially increasing number of possible execution paths), complex and external arithmetic functions, floating-point calculations, pointer operations, interaction with the environment and multi-threading [19]; predicates with only a flag variable, nested conditions and enumerations [20]; complex strings and objects with internal state [4]; nun-functional requirements [21]; exception-dependent paths [22]; and calling native code or interacting with non-native libraries.

Tools In the last decade several test input generator tools have been published. A list of tools can be found in a recent orchestrated survey [2], or on the website of one the authors[‡]. Some tools are open source, some tools are still actively developed while others are not available any more. The tools also vary in their input language and platform (C, Java and .NET).

This paper concentrates on tools for Java, and evaluates the following tools: CATG [23], EVOSUITE [24], jPET [25], RANDOOP [27], and Symbolic PathFinder (SPF) [28] and includes additional results for INTELLITEST (formerly known as PEX [26]). Table I contains some details about the tools.

[‡]"Code-based test generation methods and tools", http://mit.bme.hu/~micskeiz/pages/cbtg.html

DOI: 10.1002/stvr

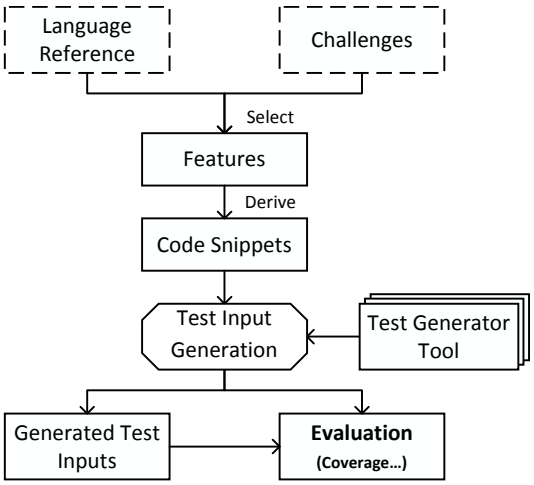


Figure 1. The approach for comparing test input generators

Our approach The work of code-based test generators usually consists of three phases:

- select inputs to reach different parts of the code,
- try to find possible errors in the reached part, e.g. dereference a null pointer,
- capture the (current) behavior in asserts, to be used as test oracles.

This work focused on the first phase, i.e. comparing the generated test inputs. The overview of the approach is illustrated on Fig. 1.

1. The common language elements and program organizational structures for C/C++, Java and .NET languages were collected ranging from basic data types and operators to complex concepts like handling string manipulations or class inheritance. These are collectively referred as *"features"*. The ones responsible for the challenges above were also included.
2. These features are later mapped to a specific programming language by creating *code snippets* targeting a feature. A code snippet is an executable program code, like a general main function. Several code snippets could be defined for a feature depending on its complexity. The majority of the code snippets contain 10–20 lines.
3. The tools under evaluation are ordered to generate inputs for these snippets separately. The generated inputs, the code coverage achieved by these inputs on the code snippets and several other metrics are collected.
4. Using these results detailed feedback can be given and several tools can be compared.

The next sections detail the features (Section 3), the code snippets (Section 4), the design of the experiments performed to evaluate the approach (Section 5), the obtained results (Section 6) and the discussion of the experiences with the tools (Section 7).

3. FEATURES TO COMPARE

The goal of one feature is to check whether a tool supports a certain language construct or program organizational structure (e.g. recursion). These features are grouped into categories and focus on imperative programming languages. The guidelines during the selection of the features were the following:

- *Coverage:* in order to get basic and detailed feedback on the tools, typical language elements shall be covered at least once. It must be noted that because of the large number of elements and combinations full coverage cannot be a reasonable objective.

DOI: 10.1002/stvr

- *Clarity:* the methodology should be clear for each programming language since sometimes the common concept in two different programming languages can have different meanings.
- *Well-organized structure:* it not only increases clarity and helps maintenance, but all the partial and final results will have the same structure, which makes evaluation easier.
- *Compactness:* the number of code snippets should not be unnecessarily large, otherwise the maintenance, the test execution and the evaluation would require more resource.
- *Minimizing the dependencies:* inevitably there will be dependencies between the features. For example, to use a conditional statement, support for the used type is essential. Care must be taken about that dependencies should be only present in one direction between two criteria and there should be no circular dependencies. In addition, the number of dependencies should be small.

Combination of features We generally tried not to combine the features to have small, isolated snippets for each feature. However, this was not possible i) for some of the basics (types or operators, which are used later in every feature), ii) where a feature builds on a previous one (objects usually have fields, like structures), or iii) a feature is needed to trigger different behaviors (conditionals are required to have multiple branches). Moreover, in order to have a reasonable number of snippets that can be manually examined and comprehended, these combinations are also limited, e.g. conditionals are used in the subsequent categories, but only `if` is employed and not `switch`.

Basic definitions Before discussing the concrete features, some notions must be clarified, as the differences between C/C++, Java and C# can be significant:

- *Function:* a program routine which can be called directly several times without object instantiation, i.e. functions in C/C++, static methods in Java and C#.
- *Structure:* a complex type which can contain other types (even another structure), but does not have methods and all parts of it are accessible, i.e. structs and classes without methods and with only public fields.

3.1. Core Features

Table II lists the core features selected in our initial work [10], whose details will be discussed in the next subsections.

3.1.1. Primitive Types and Operators (B) As our former experiences showed, it is not obvious that a tool is capable of handling all the primitive types and constructs of a programming language, thus the support for these features should be checked first. This category also includes operators, control flow statements and both simple and complex mathematical problems. Arrays[§] are checked with safe and unsafe snippets: safe snippets handle illegal indices and null references, while unsafe snippets do not. The ability of the tool to detect common exceptions is also checked in this category. The main questions were whether a tool is able to

B1 handle all the basic language elements,
B2 solve simple and complex arithmetic problems,
B3 generate inputs for simple loops, loops with inner state and nested loops,
B4 use arrays and generate arrays as input values,
B5 dispatch function calls, cover called functions and handle recursion and
B6 detect exceptions and handle exception-specific language constructs.

3.1.2. Structures (S) The following step is to check support for complex (data) types. The goal is to determine whether a certain tool is able to handle types containing other types, even in combination with conditional statements and loops. The main questions were whether a tool is able to

[§]In Java, arrays are also objects, however, they are discussed here because of their special function.

DOI: 10.1002/stvr

Table II. Selected core language features used in the evaluation

B	Basic language constructs, operations and control flow statements	
	B1	Primitive types, constants and operators
	B2	Conditional statements, linear and non-linear expressions
	B3	Looping statements
	B4	Arrays
	B5	Function calls and recursion
	B6	Exceptions
S	Structures	
	S1	Basic structure usage
	S2	Structure usage with conditional statements
	S3	Structure usage with looping statements
	S4	Structures containing other structures
O	Objects and their relations	
	O1	Basic object usage
	O2	Class delegation
	O3	Inheritance and interfaces
	O4	Method overriding
G	Generics	
	G1	Generic functions
	G2	Generic objects
L	Built-in class library	
	L1	Complex arithmetic functions
	L2	Strings
	L3	Wrapper classes
	L4	Collections
	LO	Other built-in library features
Others	Other features	

S1 use data fields of structures,
S2 use structures with conditional statements,
S3 use structures with looping statements and
S4 generate complex structures as input values.

3.1.3. Objects and Their Relations (O) A major part of modern programming languages support object-oriented programming and these language concepts are commonly used by software. Objects can have states and it is common that an object cannot have certain field values. In addition, other OO concepts should be supported by an ideal tool such as delegation, inheritance, interfaces, abstract classes and method overriding. Latter is not trivial since for example in C++ and C#.NET not all the methods are implicitly *virtual*.

An ideal tool should be able to i) handle objects, ii) create instances of classes, iii) guess the concrete type or create objects when using interfaces, and iv) it should never produce an object which cannot be created with the available methods. The last requirement describes that e.g. given a class whose integer field is ensured to be positive, a test input generator should never create an instance of the class, which has a negative field value. The main questions were whether a tool is able to

O1 use objects, generate objects for different criteria as input values,
O2 handle class delegation,
O3 handle inheritance and interfaces and
O4 handle method overriding.

3.1.4. Generics (G) Generics became widespread and commonly used in the last decade, therefore a test input generator should not fail when it encounters generic function or objects. When designing

DOI: 10.1002/stvr

Table III. Selected extra features used in the evaluation

Env	Working with the environment	
	Env1	Standard input
	Env2	Files and directories
	Env3	Sockets, ports
	Env4	System properties
T	Multi-threading	
	T1	Threads
	T2	Locks
	T3	Indeterminacy
R	Reflection	
	R1	Classes
	R2	Methods
	R3	Objects
N	Native code	
	N1	Native functions

concrete snippets for generics the features of the target platform should be seriously taken into account since the implementation and behavior of generics is different for all the three major platforms mentioned before. The main questions were whether a tool is able to

G1 handle and generate inputs for generic functions and
G2 use and create generic objects as input values.

3.1.5. The Built-in Class Library (L) Modern programming languages are shipped with a built-in class library, whose components receive calls quite frequently. Since today's class libraries are huge (the Java 7 SE platform API specification contains 4024 classes), these features target only a little part of it focusing on commonly used classes. The parts of the class library whose support was under investigation can be seen in Table II.

3.1.6. Others The majority of programming languages have several unique concepts and language constructs, like *anonymous classes* in Java or *delegates* and *events* in C#.NET. In addition, the support for other common practices should also be checked. One of them is the usage of a third party library for which only the binary is available. The goal is that a tool should be able to handle even this code to reach the maximal coverage. This case is not trivial for source code-based test input generators or for languages which compile to machine code. Code snippets have been implemented for the following subset:

- anonymous classes,
- enumerations,
- third party library (no source code, only binary),
- variable number of arguments.

3.2. Extra Features

The previous features contained the core parts of imperative languages, but were limited to isolated, single-threaded code (such as the code of a library). However, if complex applications are used as inputs for test generators, then the tools should handle additional features as these could be quite common. For example, Fraser and Arcuri [29] measured that in 110 randomly chosen open source projects 42% of the classes manipulate the file system and 30% try to open a socket. Pinto *et al.* [30] report that in 2 227 analyzed Java projects 77.5% contained at least one occurrence of a concurrent programming construct.

From the challenges listed in Section 2 the following ones were selected (Table III).

DOI: 10.1002/stvr

3.2.1. Working with the environment (Env) Applications can communicate with or depend on the environment in several ways. The most frequent methods are using the standard input or output, working with the file system or communicating through the network. Moreover, even simple queries, like getting the current date, a random number or an environment variable represent dependencies that have to be handled and manipulated somehow by the test generator. The main questions were whether a tool is able to handle code

Env1 reading or writing the standard input or output,
Env2 working with the file system,
Env3 opening network sockets and reading or writing to them[¶],
Env4 querying system properties, system clock and random generator.

3.2.2. Multi-threading (T) Multi-threading is a commonly used feature that poses a special challenge to test generators as the behavior of the tested code depends on the timing and ordering of the different threads. More complex interactions need to be managed if threads start to use locking. This could lead to many well-known problems (e.g. race conditions, deadlock or livelocks). The main questions were whether a tool is able to handle

T1 code starting and manipulating individual threads,
T2 code creating and waiting on locks,
T3 typical problems in multi-threading and discover errors which have low chance to occur.

3.2.3. Reflection (R) Reflection can be used to access and manipulate class, field and method information at runtime. A program can work with the already loaded types, or can use reflection to dynamically load new classes, both cases should be taken into account. The main questions were whether a tool is able to handle code

R1 querying class information with reflection,
R2 querying method information with reflection,
R3 querying information about objects at runtime.

3.2.4. Native code (N) An application can use external functions or libraries that are written in a native language (e.g. a Java application calling into a C library). Only the most simple cases were considered: calling a native function directly or using an object with a method implemented as a native function. The main question was whether a tool is able to handle code

N1 calling native functions.

3.3. Evaluating the Selection of Features

The selection of the features was a systematic approach based on language references and the challenges reported in related work.

Language The code snippets use 90% of the Java keywords, the omitted ones are the following:

- `assert`: assertions which should be never triggered in production code, not turned on by default,
- `const`, `goto`: not used reserved words,
- `transient`: used to prevent the serialization of certain fields,
- `strictfp`: ensures that floating-point precision is the same on any platform,

[¶]For these snippets the server is running on a separate thread in the same JVM and its bytecode is accessible for the tools.

DOI: 10.1002/stvr

The code snippets contain 136 out of the 202 (67%) possible Java SE 8 JVM bytecode instructions that can appear in class files. However, if instructions only differing in indexes (e.g. `dload_0` vs. `dload_1`) or in types (e.g. `f2l` vs. `i2l`) are not counted separately, then only 16 instructions are missing. The main types of missing instructions are the followings:

- arithmetic and logical instructions (16 out of 36),
- array load and store (12 out of 16),
- load and store from local variables (11 out of 50),
- conversion (9 out of 15),
- jumping and return (4 out of 27),
- pushing and popping values from the stack (10 out of 26),
- other: `invokedynamic`, `multianewarray`, `nop`, `wide` (4 out of 32)

Challenges The features cover the following challenges from Section 2.

- *Path explosion:* loops (B3) and recursion (B5).
- *Complex and external arithmetic functions:* mathematical expressions (B2) and arithmetical functions (L1).
- *Floating-point calculations:* conditions using floats (B2).
- *Flag variables, nested conditions and enumerations*: conditions (B2d), enumerations (Others).
- *Complex strings*: string handling (L2).
- *Objects with internal state*: object usage (O1).
- *Exception-dependent paths*: exceptions (B6).
- *Interaction with the environment:* environment (Env).
- *Multi-threading:* threads (T).
- *Calling native code:* native functions (N).

The following challenges were not covered in the selected features.

- *Pointer operations:* as the targets of the current work were managed languages, pointers were not covered.
- *Non-functional requirements*: were not covered.

The selected features cover also several typical features analyzed in the related work. For example, Grechanik *et al.* [31] gathered the frequency of 32 Java language features from 2 080 projects. Most of these 32 features (e.g. nested or anonymous classes, inheritance or conditional statements) are included in the current work, only assertions and volatile fields are missing. Similarly, Eler *et al.* [22] analyzed how frequent loops, different basic types (integers, arrays, objects...), external calls, and exceptions are in 147 open source Java projects, which are all included in the selected features. However, it should be noted that frequency does not imply importance. For example, Qu and Robinson [8] measured that while floats only appear in a very limited percentage of functions in six large C and Java programs, they prevent successful test generation in many more cases. Moreover, "coverage" of features is only partial. For example, while inheritance is included in the code snippets, their maximal inheritance depth is two, while in the study of Grechanik *et al.* [31] classes with inheritance depth of five were also found. Therefore, while the language keyword and the general concept is included in the selected features, there are sub-cases that are not covered.

We aimed for a set of features and code snippets that are able to check tool support for not only basic, but more complex language concepts and program organizational structures. On the other hand we wanted to keep the number of features and snippets manageable, thus we had to find the right balance (similarly to other testing activities). Nevertheless as the experiments showed the selected features could provide useful insights and are able to identify issues in tools. Once the tools will handle all the selected features, new ones could be added to extend the scope of this work.

DOI: 10.1002/stvr

Table IV. Number of code snippets by categories

Feature	Category	# of code snippets	Total
Core	Basic	62+31+27+18+10+21	169
	Structures	4+4+6+3	17
	Objects	21+2+8+4	35
	Generics	4+6	10
	Library	20+13+3+11+10	57
	Others	1+4+3+4	12
	Total		300
Extra	Environment	4+7+7+4	22
	Multi-threading	3+2+4	9
	Reflection	8+8+4+4+6	30
	Native	2	2
	Total		63

4. IMPLEMENTATION

The features defined in Section 3 were mapped to the Java language, and code snippets were created implementing them: 300 for the core features and 63 for the extra features (Table IV). As it can be seen, the majority of the code snippets focus on the basic features. The reason for this is because the basic features should be individually checked, including all the types and operators.

For each code snippet meta-data (goal, maximum reachable coverage etc.) and sample inputs (with which the maximal reachable coverage can be achieved) were defined. Sample inputs can be used for demonstration and validation purposes.

An example code snippet can be seen in Listing 1. It belongs to feature *O1* in the *Objects* category, and checks whether a tool can instantiate an object and perform state changing methods calls on it to reach a desired state (in the example the `SimpleObject` class has an `int` field and an `addAbs(int)` method to manipulate it). Sample inputs for the snippet can be seen in Listing 2. Three inputs are defined, the first one (`null`) covers the branch with -1 return value, the second the 1 return value, and the third the 0 return value respectively.

Since in Java a sequence of statements can only be defined inside classes, all code snippets are defined in a *final* class which is called a *snippet container*. One or more snippets can be defined in a snippet container. All the code snippets must be directly callable (i.e. they must be *public static* methods) and they can require an arbitrary number of parameters. Annotations can be used to specify the required statement coverage or included method coverage (i.e. methods whose coverages should be also taken in account during evaluation). For example, in the example snippet all statements in the `getResult()` and `getOperationCount()` methods should also be covered. For defining sample inputs we first tried to use annotations too, but in this way complex inputs (like the second one in the example, where also method calls have to be described) could not be reliable specified, thus Java code was chosen as a format.

Listing 1: Example code snippet

```
@SetteRequiredStatementCoverage(value = 100)
@SetteIncludeCoverage(classes = {SimpleObject.class, SimpleObject.class},
  methods = {"getResult()", "getOperationCount()"})
public static int guessObject(SimpleObject obj) {
  if (obj == null) {
    return -1;
  }

  if (obj.getOperationCount() == 2 && obj.getResult() == 3) {
    return 1;
  } else {
    return 0;
  }
}
```


DOI: 10.1002/stvr

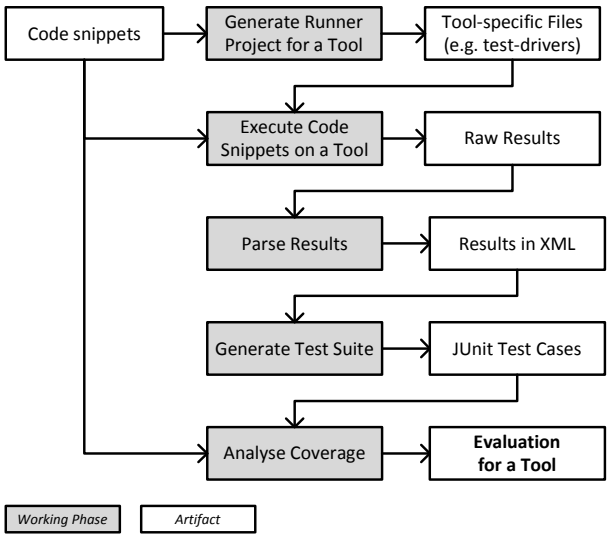


Figure 2. Workflow of the SETTE framework

Listing 2: Defined sample inputs for the example code snippet

```
public static SnippetInputContainer guessObject() {
  SnippetInputContainer inputs = new SnippetInputContainer(1);
  // sample input 1
  inputs.addByParameters((Object) null);
  // sample input 2
  SimpleObject obj = new SimpleObject();
  obj.addAbs(3);
  obj.addAbs(0);
  inputs.addByParameters(obj);
  // sample input 3
  inputs.addByParameters(new SimpleObject());

  return inputs;
}
```

The execution of these snippets for several different tools by hand is extremely expensive and error-prone. First, the tools require different configuration files and test-drivers. Secondly, the format of the output is different for each tool and a tool can have several output channels, e.g. standard output, standard error output or an XML file containing the generated test inputs. In addition, each execution has to be performed separately to guarantee the isolation between test input generations for different code snippets. Some tools report on the achieved coverage, but as there exists several different coverage measurement techniques (e.g. source or bytecode level), this information cannot be reliably used for comparison.

To overcome these problems we have developed the SETTE[‖] framework, with which (i) code snippets can be defined and categorized, (ii) sample inputs for the code snippets can be specified, (iii) the test input generators can be executed automatically on the code snippets, (iv) the results can be collected into a common XML format, and (v) the reached coverage can be measured uniformly using the JaCoCo [32] code coverage library. The SETTE workflow is shown in Fig. 2.

Developing such an evaluator framework requires a great effort since it not only involves job management, manipulation of Java classloaders and integration with the tools and the code coverage library, but involves numerous trials. This is because the output format of the tools (standard output, error output, XML) is often unspecified and there are many rare corner cases. After everything is parsed into a common format, test source code compilation requires attention and larger resources (out of memory compiler errors were encountered several times in the experiments).

[‖]Initially SETTE stood for *Symbolic Execution-based Test Tool Evaluator*.

DOI: 10.1002/stvr

To support additional test generator tools, all the core code snippets have been manually translated to C#.NET code, and differences between the Java and C# were taken into account (e.g. wildcard generic types). However, translating the extra snippets was not that straightforward. One of the snippets has no equivalent in .NET (`Env4_System.setsProperty` for system properties), and some of the features are implemented quite differently in .NET with no direct mapping from Java (e.g. multi-threading is implemented with delegates instead of subclassing).

5. EXPERIMENT PLANNING

To validate the approach experiments were performed on six test input generator tools. This section presents the detailed design of the experiments.

5.1. Objective and Research Questions

The main *objective* of the research was to create a method and framework for comparing and evaluating test input generators. Thus the experiments were designed to answer the following *research question*: Is the approach able to produce fine-grained feedback on a tool's capabilities?

More specifically:

- *RQ1*: Which of the defined core features are supported by each tool?
- *RQ2*: How do the generated test suites compare to each other with respect to typical metrics?
- *RQ3*: How do the tools behave when given more time for test generation?
- *RQ4*: How can the tools handle the extra features?

In RQ1 the outcomes are evaluated using simple categories, e.g. the tool threw an error during test generation or was able to cover all statements in a snippet. RQ2 is concerned with more detailed metrics (size of tests, duration of test generation...). For RQ1 and RQ2 the experiments use a fixed time limit (30 seconds per snippet). Experiments for RQ3 gradually increase this time limit to study its effects. RQ4 is concerned with features for environment, threads, reflection and native code.

Note that the goal was not to compare the techniques itself or draw conclusions whether tool X is generally better than tool Y. This is not even possible with the current approach, as the size of the study objects (the selected code snippets) is too small and they are biased.

5.2. Subjects of the Experiments

Five tools have been chosen for tool evaluation, which have been fully integrated into SETTE, thus the execution and data collection is *automatic* for them.

- CATG [23] performs instrumentation and symbolic execution on Java bytecode.
- EvoSuite [24] uses genetic algorithms and mutation to evolve and reduce test suites.
- jPET [25] translates Java bytecode to Prolog and performs symbolic execution on that.
- SPF [28] does not translate nor instrument the bytecode, but uses a custom Java Virtual Machine, *Java PathFinder* (JPF) for execution.
- Randoop [27] uses feedback-directed random testing to generate tests.

To extend the findings of the experiments a tool for .NET was included and evaluated. In this case the execution was automatic, but some steps in the analysis were manual.

- IntelliTest / Pex [26] is a SE-based test input generator for .NET.

Pex was the academic tool developed by Microsoft Research, which was incorporated in the newest version of Visual Studio as a feature called IntelliTest. IntelliTest uses the Pex engine internally, just not all functionality of Pex is available through its user interface yet (e.g. there is no command line or detailed configuration support). In our initial work [10] the Pex tool was used, but it was replaced with IntelliTest for the current paper.

DOI: 10.1002/stvr

Table V. Configurations of test input generators used in the experiments

Tool	Version	Configuration
CATG	v1.03 (Yices 2.4.1 64-bit)	`./concolic 100 CODE-SNIPPET`
EvoSuite	1.0.3	`-class CODE-SNIPPET -Dsearch_budget=TIMEOUT -Dassertions=false -Dminimization_timeout=10 -Djunit_check_timeout=10`
jPET	0.4	`-c bck 10 -td num -d -100000 100000 -l ff -v 2 -w -tr statements -cc yes`
IntelliTest	Visual Studio 2015	`[PexMethod(Timeout=30)]`
Randoop	2.1.4	`gentest --timelimit=TIMEOUT --inputlimit=5000 --string-maxlen=50 --randomseed=...`
SPF	rev. 4cd8ac11abee (JPF rev. 820b89dd6c97)	Constraint solver: CORAL Listener: `gov.nasa.jpf.symbc.SymbolicListener`

5.2.1. Selection procedure The following procedure was used to select these tools. Our initial work starting in 2013 concentrated on tools using symbolic execution and working on the Java language [10]. Tools were excluded that were not updated recently (e.g. jCUTE, jFuzz, LCT). Not many tools remained. CATG was selected as it was open source, jPET as it used a significantly different approach, and SPF as it builds on the mature JPF framework.

Next the search was broadened with search-based and random tools. EvoSuite and Randoop were added as they both are mature tools used in several case studies.

We tried to contact vendors of commercial Java test generator tools, but were not able to request a version for evaluation. We explored the tools from the SBST Java Unit Testing Tool Contest [33, 34]. GRT and JTExpert seemed to be not stable and robust enough in the initial trialing. T3 seemed a good candidate, however it saves the generated tests only in binary format, thus only limited evaluation could have been performed on it. Therefore the above six tools were used in the evaluations.

5.2.2. Version and configuration General information about the subjects of the experiments were presented in Table I. The used tool versions and configurations are shown in Table V. Note that some parameters not influencing test generation (e.g. naming the resulting tests or specifying class path) are omitted. These can be found in the repository of the SETTE framework.

The following non-default parameters are used. For CATG only the trial count is supplied: the value 100 is commonly used by the tool's developers. EvoSuite uses the following main phases: searching for tests, minimizing the generated tests suite, generating assertions, and checking the resulting tests by compiling and running them. The generations of assertions is disabled, and a fixed 10 seconds timeout is used for minimization and test checking. For the search a variable timeout is used, which will be explained later. For Randoop the length of generated strings are limited to 50 characters and the number of generated tests methods to 5000, because both are sufficient to cover the snippets, and otherwise Randoop would generate extremely large values and tests. For SPF the randomized CORAL constraint solver is used. IntelliTest sadly does not offer the rich configuration options of Pex, it can only be customized with adding .NET attributes to the test classes. These attributes allow to control and limit the test generation similarly to Pex, but report generation and the functionality to generate tests from multiple classes is not available through them. In the experiment only the timeout was specified, and IntelliTest was started inside Visual Studio through its GUI.

The most recent releases of the tools were used as of March 2016. These are different versions from the ones used in our preliminary work [10] (except for CATG and jPET), thus their evaluation results have changed with respect to the previously reported ones.

DOI: 10.1002/stvr

5.3. Process and Data Collection

The objects of the study were the developed code snippets, and the subjects of the study the selected test input generator tools. The process of the experiments was detailed in the workflow of SETTE in Figure 2, which was repeated for each of the tools. To summarize it shortly:

1. The snippet project was transformed to the format required by the tool (e.g. adding configuration or runner files).
2. The tool under study was executed to perform test generation for each of the code snippets separately.
3. Detailed results (e.g. generated test input, log files, achieved code coverage and possible errors raised) were collected for evaluation.

For evaluation purposes the following data was collected.

5.3.1. Status and coverage We defined a status variable to represent the overall outcome of each execution. For each tool and each snippet exactly one flag from the followings was assigned.

- *N/A*: The tool was not able to perform test generation since the snippet's code could not have been specified for the execution or the tool signaled that it cannot deal with the code snippet.
- *EX*: Test input generation was terminated by an exception, which was thrown by the code of the tool or the tool did not caught an exception thrown from the code snippet and stopped.
- *T/M*: The tool reached the specified external time-out and it was stopped by force without result or the execution was terminated by an out of memory error. Note that if a tool stopped the execution itself, the result is categorized as *NC* or *C* instead.
- *NC*: The tool has finished test input generation before time-out, however, the generated inputs have not reached the maximal possible coverage.
- *C*: The tool has finished test input generation before time-out and the generated inputs have reached the maximal possible coverage. If an execution is classified into this category it means that the tool has generated appropriate inputs for the code snippet.

It can be easily decided whether a result of an execution should be categorized into the first three or last two categories. However, to determine whether it goes to *NC* or *C*, the snippet must be executed with the generated inputs and coverage should be measured. The evaluation is automatic and is performed by SETTE. The method of coverage measurement is based on JACOCO [32] and it is uniform for all the tools. Currently SETTE measures statement coverage. (Note that because the snippets were designed in a way that usually every branch has a return statement in it, this is also a good indicator of branch coverage.)

Code coverage are frequently used as a metric by tool developers. However, research [35, 36] suggests that high code coverage is not necessarily correlated with the effectiveness of the tests. Thus other metrics were included in the evaluation.

5.3.2. Size The size of the generated tests are also an important factor, because it increases the cost of test execution and manual oracle definition. There are several choices how to measure "size" (e.g. number of test cases or number of statements).

First, the *number of generated test cases* are collected for each snippet. For the tools which does not generate any test cases but only test inputs (CATG, jPET, SPF), this number means the *number of generated test inputs* since the SETTE framework generates one test case with an assertion for each input.

Next, for those tools that generate test cases (EVOSUITE, INTELLITEST, RANDOOP) the *length of the tests cases* (in bytes) are measured. Comments and code outside the test methods are not counted, but comments and new lines inside a test method are included in this metric.

5.3.3. Duration The duration of each execution for each snippet is collected. For the Java tools time is measured by SETTE from the start of the generation process until process termination (using the `System.currentTimeMillis()` method).

DOI: 10.1002/stvr

5.3.4. Mutation score Mutation analysis [37] can be used to assess the quality of a test suite by injecting faults in the unit under test, and measuring how many faulty versions (mutant) can be differentiated from the original version by the tests. An existing mutation framework is used with the following settings to compute mutation adequacy scores.

- *Mutation tool*: Version v1.1.7 of the MAJOR [38] mutation framework was applied.
- *Source of the mutants*: For the mutation analysis the source of the core snippets were used. Snippets containing unbounded or infinite loops were not included, because MAJOR did not stop when executing these snippets.
- *Number of mutants*: Starting from the above snippets 6 236 mutants were generated using the default settings of MAJOR with all its mutation operators enabled. This fixed set of mutants were later used in the analysis of the test suites from the different tools.
- *Equivalent mutants*: Due to the high number of the mutants, non-killed mutants were not checked manually. Mutants that were not killed by any tool were considered as equivalent with the original. This is an overestimation, but this strategy is commonly used when comparing different test suites [35, 39].
- *Assertions*: The number of killed mutants depends heavily on the number and quality of the assertions in the test code. CATG, jPET and SPF only generate test inputs and no assertions, RANDOOP and INTELLITEST generate assertions mostly stating the return value of the snippet method, while EVOSUITE could use an extra phase to search for detailed assertions. To have a fair comparison, EVOSUITE's assertion generation functionality was disabled, and the SETTE framework adds a JUnit assertion capturing the return value observed during the test execution phase.

5.3.5. Process for IntelliTest For INTELLITEST a different process has to be used. Parametrized tests methods are generated for each snippet inside Visual Studio. Next, INTELLITEST is started manually. Because it does not produce PEX's detailed reports, the data has to be assembled with other tools. OPENCOVER[**] is used to measure statement coverage, and PowerShell scripts process the coverage report and extract the necessary data from the generated tests. As INTELLITEST is invoked through GUI, no reliable duration data could be collected for it (for most snippets the duration of test generation in INTELLITEST is in the 100 ms magnitude). The mutation analysis is carried out by transforming the C# test code to Java and running the analysis using the same method as for the Java tools. This enables to compare all the tools using the same mutant set.

5.4. Experimental Setup

The next section describes the platforms running the experiments, and the chosen time limit and repetition count.

5.4.1. Platform Two platforms were used in executing the experiments. Previously virtual machines running on an internal server cluster were used, however the final experiments were performed on a dedicated hardware to minimize distortions.

1. For all the tools (except INTELLITEST) a headless server running Ubuntu 14.04 64-bit and Oracle's Java 8 implementation was used. The server had 8 GB memory and two quad core 2.5 GHz E5420 Xeon processors. The server had no other background roles.
2. INTELLITEST was executed on a laptop with 12 GB memory and 1 dual core 1.7 GHz i3-4010U processor running Windows 10 64-bit and Microsoft .NET 4.6.1.

5.4.2. Time limit Two sets of experiments were performed with different external time limits enforced by SETTE.

[**]OpenCover, URL: https://github.com/OpenCover/opencover

DOI: 10.1002/stvr

Table VI. Number of snippets with different results when running 10 times with 30 s time limit

Tool	Status	Size	Coverage
CATG	3	3	–
EvoSuite	41	113	56
jPET	–	–	–
IntelliTest	–	4	–
Randoop	8	103	13
SPF	–	–	–

1. *Fixed time limit*: For the first set of experiments a fixed time limit, *30 seconds* was chosen for all 363 snippets.
2. *Variable time limit*: To validate the chosen time limit, in a second set of experiments a subset of snippets were run with different time limits.

The rationale behind choosing 30 seconds was the following. Our experiences have shown that in the given environment the SE-based test generators usually finish in 10 seconds and if a test generator uses more than 20 seconds of runtime, it will run out of memory sooner or later without finishing test input generation. However, it is advised to use a time limit greater than 10 seconds because heavyweight tools like SPF might need a couple seconds to initialize (in case of SPF the JPF JVM has to be started on each execution). On the other hand, EvoSuite and Randoop use all the time for searching and could improve the results with more time given. The developers of EvoSuite and Randoop usually use a time limit of 2 minutes per class in their experiments [27, 29], which is comparable with the current setting (the SETTE suite contains on average 6.93 snippets per class, thus the 30 seconds per snippet limit means a 3.5 minutes per class limit on average).

To validate the chosen time limit, a second set of experiments was performed with variable time limits. For these experiments a subset of the snippets was chosen (129 from the core 300), which are harder to cover for the tools, namely the categories B2, B3, O1–O4, G1, G2, L1–L4 and LO. Experiments with the following time limits were run: 15, 45, 60, and 300 seconds. Then the impact was analyzed of the increasing time limit on the test generation. The variable time limit experiments have not been performed on IntelliTest, as currently it cannot be automated, it can only be invoked through its GUI.

5.4.3. Repetitions As some of the tools utilize random algorithms, it was necessary to repeat the experiments several times [40]. The executions were performed *10 times* for each tool as in previous experiments from the tool's developers [27, 29]. In case of EvoSuite and Randoop a different random seed was used each time.

The variability of the results are listed in Table VI. In case of EvoSuite and Randoop there were some snippets, where some of the executions were able to achieve the maximal coverage (*C*), while others not (*NC*). This needed to be taken into account, as e.g. for EvoSuite depending on which run was analyzed the number of snippets categorized as *NC* differed significantly. To resolve a *C* status was only assigned if more than 5 from the 10 repetitions resulted in *C* (a similar method was used by Shamshiri *et al.* [41]). For size and coverage aggregated values were calculated (mean, median, standard deviance...). In case of CATG, only 3 extra snippets for multi-threading produced different results. For IntelliTest only the number of generated tests differed in some runs. For jPET and SPF the 10 runs resulted in the same status, size and coverage values.

The repetitions resulted in 18 000 observations for the first experiment with the core snippets (6 tools × 300 snippets × 10 repetitions), 3 150 observations for the first experiment with the extra snippets (5 tools × 63 snippets × 10 repetitions), and 25 800 observations from the second experiment (5 tools × 129 snippets × 10 repetitions × 4 time limit). This required approximately 400 hours just for the test generation phase (this does not include the time for preparing the context of an execution, and later running and evaluating the generated tests). Note that the extra snippets were executed only once for IntelliTest.

DOI: 10.1002/stvr

Table VII. Number of snippets categorized into the different statuses for all tools

Tool	N/A	EX	T/M	NC	C
CATG	105	28	28	27	112
EVOSUITE	–	–	–	47	253
INTELLITEST	2	–	–	20	278
jPET	108	3	9	16	164
RANDOOP	–	–	–	112	188
SPF	9	7	26	91	167

5.5. *Threats to Validity*

Reliability of the experiments: For the first five subjects the SETTE framework automated the whole experiment to eliminate human errors. To reduce the risk of having errors in the framework itself, the results were checked also manually (e.g. if an exception was produced then it was not because of the framework). In case of INTELLITEST the status was partly automatically determined with a Powershell script and partly checked by one of two authors.

Knowledge of the tools: These tools are fairly complex and configurable software (e.g. EVOSUITE has 29 options and 296 parameters), and neither tool was developed by the authors. Care was taken to examine the possible options and encountered errors of each tool, but it is likely that some of the otherwise reported code snippets can be handled by the tool with advanced parametrization. However, the results are a good indicator of what results could be produced by a *tool user*.

Selection of subjects: There are several other test generator tools. Initially only Java-based tools were selected because Java was the platform for which the most tools are available. Later the selection was extended to a tool with different platform (INTELLITEST). Note that the findings are about the tools and not about the techniques they use (as shown by the great differences in the results for all symbolic-execution based tools).

6. RESULTS

This section presents the results and discusses them in the context of the research questions. The detailed experimental results can be downloaded from the SETTE website. For each tool the followings were uploaded: the tool's configuration, the tool's full output, the generated test inputs and codes and coverage colored snippet codes to help to validate the results.

The data analysis was performed using the R statistical framework [42] (version 3.2.2).

6.1. *RQ1: Overview of the Tools' Capabilities*

Table VII presents a high-level overview of the results, the number of snippets categorized into the different statuses for the core features. There is a significant difference in how the tools handled the snippets, some had issues (statuses *N/A*, *EX*, *T/M*) with a large number of snippets, while others were able to fully cover the majority of the snippets.

Note that as newer versions of the tools were used and the required coverage for some snippets have been corrected, these results are slightly different from the ones reported in our preliminary work [10]. Moreover, the results of only one execution was reported in our preliminary paper, which limited the precision of the data (e.g. as discussed in Section 5.4.3 EVOSUITE's results are varying in the different executions).

A more detailed breakdown can be seen in Table VIII. For each tool and each feature category the numbers of the code snippets classified as *C*, *NC*, *T/M*, *EX* and *N/A* are displayed.

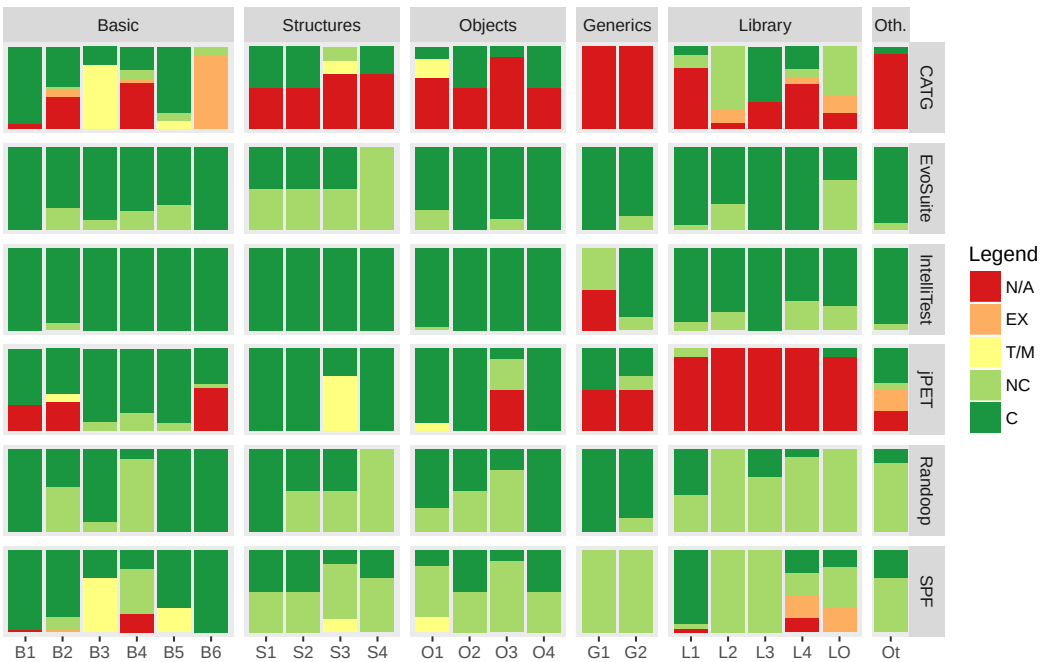
Note that the total numbers should not serve as global indicators for tool quality. For example, if a tool generates only trivial inputs (like zeros) and another only misses one branch, both are classified as *NC*. Moreover, some code snippets represent corner cases that are not frequently seen. Instead, the table should serve as a high-level overview to identify possible issues and then the details should be consulted. Fig. 3 presents a visualization of the results that highlights the data with color codes.

DOI: 10.1002/stvr

Table VIII. Number of snippets categorized into the different statuses by snippet categories for all tools

		Basic						Structures				Objects				Generics		Library					Others
		B1	B2	B3	B4	B5	B6	S1	S2	S3	S4	O1	O2	O3	O4	G1	G2	L1	L2	L3	L4	LO	
Total		62	31	27	18	10	21	4	4	6	3	21	2	8	4	4	6	20	13	3	11	10	12
CATG	C	58	15	6	5	8		2	2		1	3	1	1	2			2		2	3		1
	NC		1		2	1	2			1								3	10		1	6	
	T/M			21		1				1		5											
	EX		3		1		19												2		1	2	
	N/A	4	12		10			2	2	4	2	13	1	7	2	4	6	15	1	1	6	2	11
EvoSuite	C	62	23	24	14	7	21	2	2	3		16	2	7	4	4	5	19	9	3	11	4	11
	NC		8	3	4	3		2	2	3	3	5		1			1	1	4			6	1
	T/M																						
	EX																						
	N/A																						
IntelliTest	C	62	28	27	18	10	21	4	4	6	3	20	2	8	4		5	18	10	3	7	7	11
	NC		3									1				2	1	2	3		4	3	1
	T/M																						
	EX																						
	N/A															2							
jPET	C	42	17	24	14	9	9	4	4	2	3	19	2	1	4	2	2					1	5
	NC			3	4	1	1							3			1	2					1
	T/M		3							4		2											
	EX																						3
	N/A	20	11				11							4		2	3	18	13	3	11	9	3
Randoop	C	62	14	24	2	10	21	4	2	3		15	1	2	4	4	5	11		1	1		2
	NC		17	3	16				2	3	3	6	1	6			1	9	13	2	10	10	10
	T/M																						
	EX																						
	N/A																						
SPF	C	60	25	9	4	7	21	2	2	1	1	4	1	1	2			18			3	2	4
	NC		5		10			2	2	4	2	13	1	7	2	4	6	1	13	3	3	5	8
	T/M			18		3				1		4											
	EX		1																		3	3	
	N/A	2			4													1			2		

Figure 3. Visualization of snippets status by categories for all the tools. Each color represents the percentage of snippets assigned the given status in the given subcategory.



DOI: 10.1002/stvr

6.1.1. CATG has no problem with basic features except that the tool does not support floating-point numbers. Regarding conditional statements and loops CATG is able to handle simple cases such as linear statements and loops with smaller state space, however, it cannot cover fully more complex code parts. CATG cannot generate arrays as input and cannot solve constraints for array indices. In addition, the tool does not catch all the exceptions coming from the code and this usually results in tool shutdown.

In case of structures and objects, CATG is able to handle the fields but cannot generate objects as input. However, when more complex constraint solving is needed, then in most of the cases CATG exceeds the time limit. Generics and the majority of the arithmetic functions are not supported by the tool. CATG is able to generate strings as input, but constraint solving is only supported for the `equals()` method.

6.1.2. EVOSUITE generates test cases directly instead of just listing test input values. The tool handles all the bytecode instructions and terminates when the time limit has expired, resulting in no generations categorized as *N/A*, *EX* or *T/M*. EVOSUITE reaches high coverage on the majority of the code snippets. It can handle snippets for objects and generics well, which are challenging for the other tools. The not covered library cases focused on special features, such as not common string methods, date and UUID guessing. However, EVOSUITE's limit is handling floating-point numbers, solving complex constraints and mathematical problems and covering codes with looping statements.

6.1.3. INTELLITEST handled all the instructions, the tool detects exceptions and shuts itself down after the time limit has expired. Thus, no generations were classified as *EX* and *T/M*. INTELLITEST was able to satisfy statement coverage requirements in most of the cases. However, in some cases it failed to cover all the statements.

Two executions were marked as *N/A* because INTELLITEST was unable to guess a valid generic type when there is a condition for the base class. Three *NC* snippets focused on `float` or `double` precision, 1 on object guessing, 3 on generics, 12 on built-in library features and 1 on enumerations.

*6.1.4. j*PET does not support the majority of the built-in Java objects. Although the tool supports floating-point numbers, it does not support complex conditions, some primitive types, bitwise operators and floating point number literals. Regarding conditional statements jPET underachieves the other tools, but it has good support for loops. Because of the incompleteness of the Prolog translation, jPET is only able to handle half of the exception code snippets.

However, in comparison with CATG and SPF, jPET has good support for arrays, structures and objects. The mechanism of jPET is the following: the tool builds up a heap with constraints and solves the heap during test input generation. This method seemed quite effective, however, input generation can result in invalid inputs, like an array with less elements than its length, an array having elements from different (not compatible) types or an object whose state cannot be reached by using its methods. Support for generics and calls to the Java SE library is limited.

6.1.5. RANDOOP can handle the snippets and no execution results in *N/A*, *EX* or *T/M*. It can cover most of the basic snippets, but conditions, arrays and some of the floating-point snippets present a challenge to it. Regarding structures RANDOOP can cover snippets where the values of the structure's fields are given as parameters. However, RANDOOP cannot fully cover snippets with structures as inputs, it only creates a default structure but does try to modify the fields of the structure before passing it to the snippet. Objects are handled similarly, RANDOOP creates objects using its constructors but does not try to modify the objects state with its methods before passing to the snippet. RANDOOP can cover some of the snippets using arithmetic functions, but it was not able to fully cover the snippets with string operations.

DOI: 10.1002/stvr

Figure 4. Histogram of mean coverage of each snippet aggregated by the 10 runs for each tool. Snippets with N/A, EX and TM statuses were counted as 0%.

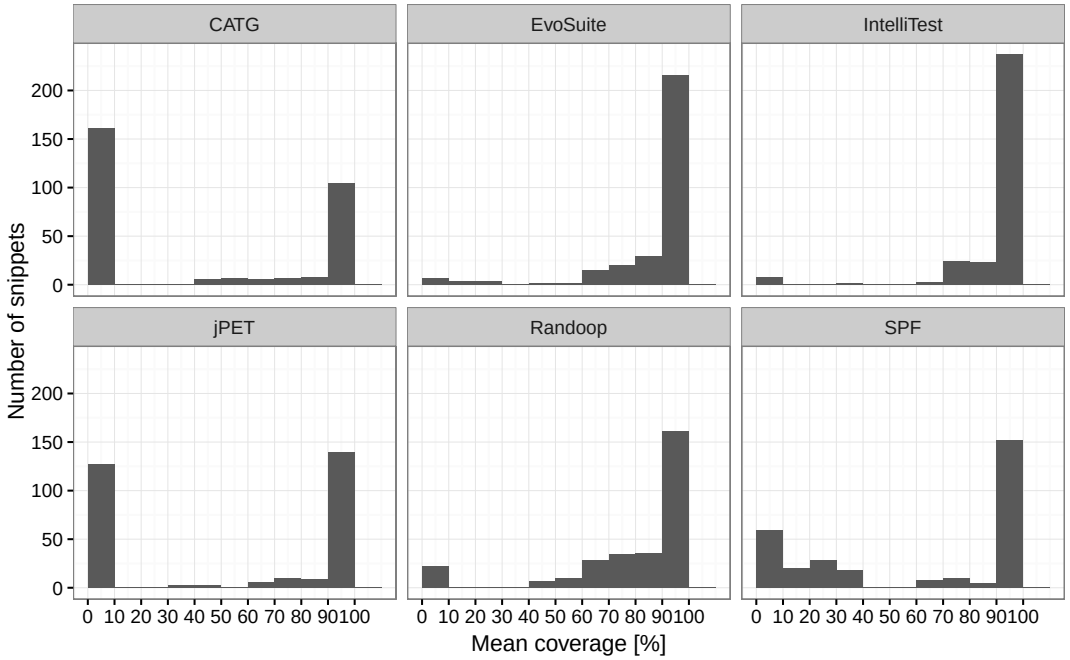


Table IX. Coverage [%] achieved by the tools (aggregates of snippet's mean coverage values)

Tool	Min	Mean	Median	Max	SD
CATG	0.00	42.42	0.00	100.00	47.07
EvoSuite	0.00	89.54	100.00	100.00	21.30
IntelliTest	0.00	92.70	100.00	100.00	19.00
jPET	0.00	54.09	79.00	100.00	47.74
Randoop	0.00	81.86	93.94	100.00	27.33
SPF	0.00	61.99	92.71	100.00	42.61
Manual	0.00	94.75	100.00	100.00	15.67

6.1.6. SPF supports all the basic types and operators except the modulo operator and only has issues with the hardest conditional statements and loops. SPF was unable to generate arrays as inputs and solve constraints for array indices. Exceptions are handled well by the tool.

Similarly to CATG, SPF has limited support for structures and objects. While CATG produces compile time errors when using objects as input, SPF generates null values and does not create any meaningful object. In addition, SPF has better constraint-solving capabilities. The tool is also able to handle the majority of the arithmetical functions, however lacks generics, string and other library support.

6.1.7. Summary of RQ1: RQ1 focused on high-level feedback. As it can be seen the selected features and code snippets were able to detect issues with the subjects. For example, some tools terminate on certain code snippets due to uncaught exceptions. Another common problem is that a tool is not prepared for some cases, like floating-point numbers, cannot handle certain literals, other language elements or bytecode instructions. The experimental results give a detailed list of these issues and provide short code snippets that reproduce them.

DOI: 10.1002/stvr

Table X. Duration [s] achieved by the tools (aggregates of snippet's mean duration values). No duration data was recorded for INTELLITEST.

Tool	Min	Mean	Median	Max	SD	Sum
CATG	0.45	2.00	0.51	27.39	3.49	1200
EVOSUITE	36.11	39.90	36.54	125.69	12.79	11970
jPET	0.04	0.53	0.06	22.31	2.52	372
RANDOOP	2.28	26.40	30.26	44.92	9.04	7919
SPF	0.69	1.39	0.70	10.38	1.29	1159

6.2. RQ2: Comparing the Detailed Metrics of the Generated Tests

Even if two tools fully cover a snippet, their generated test inputs can significantly differ. In order to have a more detailed feedback several metrics were collected during the executions of the tools and from the generated test suites. This section compares the coverage, size and mutation score of the test suites, and the duration of the generation.

When analyzing the data an important question is how to handle executions with *N/A*, *EX* and *T/M* statuses. They can either be removed or counted with 0 (as no information belongs to these observations). In either case they distort results. If they are removed a tool with many removed executions will contain only the easy snippets. On the other hand, if they count as 0, then a tool missing a feature (e.g. handling floats) will distort several snippets, which are otherwise easy to handle for all the other tools. Therefore each of the subsequent analysis steps will define which option was used for *N/A*, *EX* and *T/M* statuses.

The subsequent analyses work in most cases on aggregated data, i.e. for each tool and each snippet first the raw values form the 10 runs are aggregated to compute means. Next, for each tool these 300 mean values are reported or plotted. Therefore e.g. the raw maximum values could be higher than the values in the next sections.

6.2.1. Coverage Table IX and Figure 4 report the coverage achieved by each of the tools for all snippets. In this case missing coverage values are counted as 0%. The *Manual* row in the table represents the sample inputs selected by us that achieve the maximal coverage that could serve as a baseline.

The values are similar to the status overview, EVOSUITE and INTELLITEST were able to handle relatively well the snippets, RANDOOP had more diverse values, while CATG, jPET and SPF had varying results (high standard deviation).

As it can be seen a 40–60% mean coverage can easily be reached by even those tools who had many issues with the snippets (several snippets with *N/A*, *EX* and *T/M* statuses).

6.2.2. Duration In case of the analysis of the duration values, the executions with *N/A*, *EX* and *T/M* status were removed.

Figure 5 presents the distribution, and Table X the descriptive statistics of the mean duration values computed for each snippet and tool.

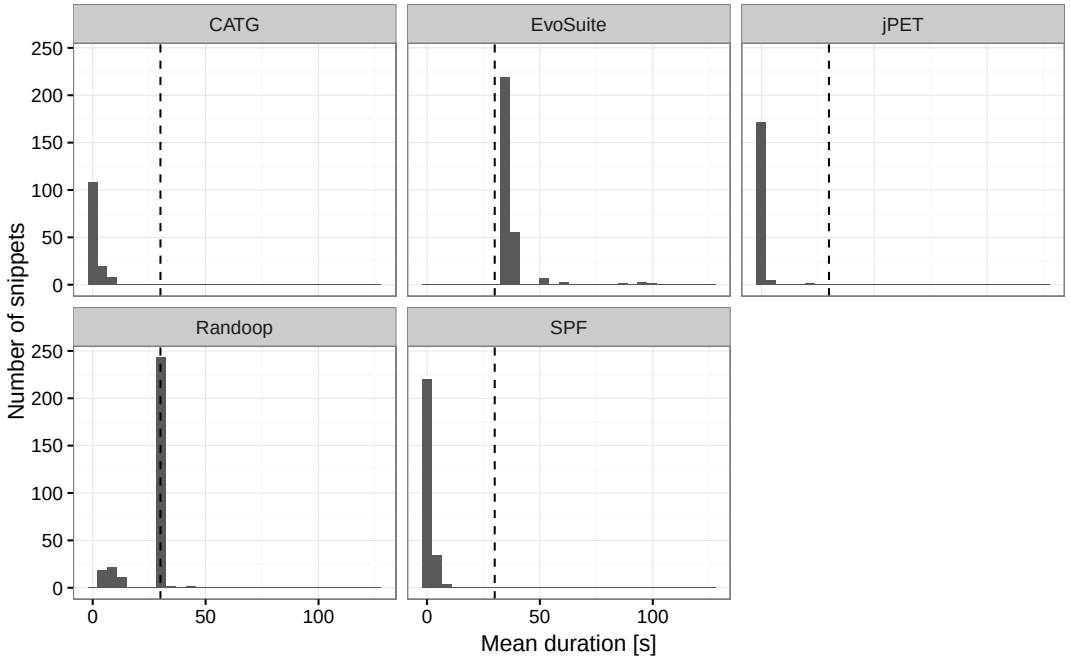
CATG, jPET and SPF were fast, they usually finished well before the time limit (note that however these tools had a large number of removed snippets).

For EVOSUITE the whole generation process (including search, test minimization, and test execution) exceeded the 60 seconds range in 150 from the 3000 executions with a maximum of 205 seconds. These long snippets included unbounded loops or recursion. EVOSUITE in most cases respected the 30 seconds time limit for the search, it only reported in 120 from the 3000 executions that the search exceeded the time limit (with a mean of 6.8 seconds). The test checking phase was responsible for the long delays as it had to execute the generated test cases calling unbounded loops and terminate them after they reached the test checking timeout.

As discussed previously no duration data was recorded for INTELLITEST. However, the duration of PEX was previously measured, and that data and our current informal observations both confirm that it was the fastest tool. If dynamic symbolic execution can find a solution, then it can find it

DOI: 10.1002/stvr

Figure 5. Histogram of mean duration of each snippet aggregated by the 10 runs for each tool. Dashed line represents the 30 seconds time limit. Snippets with N/A, EX and TM statuses were removed.



fast. Moreover, the overall low values were due to the fact that INTELLITEST has numerous built-in default boundaries and timeouts, e.g. the constraint solver calls time out after 1 second or only 500 conditions are analyzed. Therefore INTELLITEST rarely needed more than 0.1 second and only occasionally reached the 30 seconds timeout.

For RANDOOP the stopping condition is reaching the time limit (`timelimit` parameter) or reaching the limit for the number of generated test method candidates (`inputlimit` parameter) or reaching the limit for number of generated tests (`outputlimit` parameter). RANDOOP keeps generating as long as one of those limits has been reached. For 245 snippets RANDOOP reached the time limit, for the others the input limit. There were only 6 snippets that slightly exceeded the 30 seconds time limit, and all those snippets were from B3 containing nested loops.

6.2.3. Size For analyzing the size metric, the executions with *N/A*, *EX* and *T/M* status were removed as no tests were generated for them (and assigning 0 to them would distort the results).

Table XI summarizes the statistics of the generated test suite sizes. Our initial expectation was that the SE-based tools would generate few tests as they target specific paths in the program, while the random and search-based tools would generate (much) larger test suites.

INTELLITEST was confirming the expectation, it generated small test suites. In most cases it generated only those tests that were necessary to reach every block in the code or trigger possible exceptions (null reference, overflow, divide by zero, index out of range...). INTELLITEST only generated larger test suites, if it had issues with covering some parts of the snippet's code, e.g. when generating UUIDs.

CATG and SPF generated relatively small suites. For CATG only some snippets with objects and loops resulted in more than 10 tests. SPF generated more than 10 tests for 19 snippets. The snippets with 103 tests were the three `withContinueBreak` snippets in B3 loops, all the other snippets had at most 30 tests.

DOI: 10.1002/stvr

Table XI. Number of the generated tests by the tools (aggregates per snippet's mean size values)

Tool	Min	Mean	Median	Max	SD	Sum
CATG	1.0	3.44	1.0	46.0	5.64	478.0
EvoSuite	1.0	3.78	3.0	9.0	1.90	1134.1
IntelliTest	1.0	3.64	3.0	12.6	2.38	1083.9
jPET	1.0	26.93	4.0	2047.0	170.79	4847.0
Randoop	0.0	911.32	25.0	4999.0	1577.93	273394.5
SPF	1.0	4.34	1.0	103.0	11.49	1119.0
Manual	1.0	2.31	2.0	8.0	1.38	693.0

jPET generated 1026 and 2047 test inputs for 2 snippets in S3 (structures with loops and arrays), but it had 16 other snippets with size values 20–200. These are also mostly snippets containing loops, either in B3 loops or with objects or function calls.

EvoSuite achieved similar sizes than the SE-based IntelliTest, and its largest test suites were much smaller than for the other SE tools. Thus its whole test suite generation approach and minimization feature worked very well.

The much larger values for Randoop's test suite sizes are consistent with expectations (Randoop had a minimization feature, but it is not working in the current version). For many snippets it generated the maximal number of test cases (limited by its `inputlimit` parameter). The 0 minimum value requires explanation: the snippets with infinite loops are handled specially, if a tool stops in time or detects the infinite loop, then it is considered a valid execution and receives a *C*, even if it does not generate any test inputs.

Even if a tool is able to fully cover a snippet, it does matter how many extra or redundant tests did it generate. Figure 6 compares the size of the tool's test suites to the size of the manually selected sample inputs. The manually selected inputs contain the minimal number of inputs to fully cover the snippet's code. The figure's data contains for every tool, for every snippet with *C* status the size of the tool's generated test suite minus the sample input size. To increase the readability of the plot the y axis was limited to a maximum value of 30, which excluded some snippets: 2 for CATG, 3 for SPF, 10 for jPET and 76 for Randoop. There are negative values for EvoSuite, IntelliTest and Randoop on the figure. The reasons behind this are specific cases in the evaluations. If EvoSuite can only cover a snippet in less than 10 but more than 5 runs, then it receives a *C* status, but due to the *NC* runs with fewer tests the mean size would be less than the size of the *C* runs. IntelliTest is able to fully cover two snippets in Structures with fewer inputs than the Java tools, as in .NET structures are value types, and `null` cannot be assigned to them. Finally, for Randoop the special handling of snippets with infinite loops discussed previously was the cause.

EvoSuite and IntelliTest were relatively close to the sample inputs, while for jPET and Randoop there was a larger difference. The difference for CATG and SPF was also small, but these tools had many snippets not categorized as *C*, which were excluded from the plot.

The size of the generated tests can be compared by measuring the bytes of code inside the generated test methods. Table XII presents the statistics for the mean size in bytes of the generated test methods. The table contains data only for snippets with status *C*. IntelliTest generated the shortest tests, followed by EvoSuite. EvoSuite had longer tests for loops and objects. Randoop not only generated a large number of tests, but some of them were really long (especially for the generics snippets). The data highlights that there is significant difference also in the length of the generated tests (which is important for the understandability of the generated code). Note that CATG, jPET and SPF were not added to the table, because these tools generate only test input values, and the SETTE framework creates test code from the values.

6.2.4. Mutation analysis The analysis was performed on all 10 runs from the core snippets. Mean numbers were calculated from the numbers obtained from Major to average the different runs. The results are presented in Table XIII. From the 6 236 mutants 1 714 were not covered by any of the

DOI: 10.1002/stvr

Figure 6. Boxplots of the difference between the number of manually selected sample inputs and number of tests generated by the tools for each fully covered snippet (status *C*). Y axis was limited to 30.

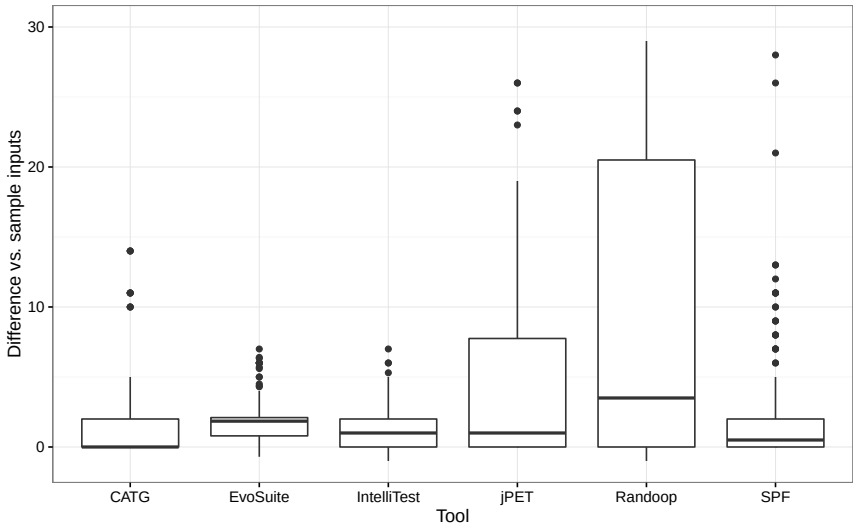


Table XII. Bytes in the generated tests by the tools (aggregates per snippet's mean size values)

Tool	Min	Mean	Median	Max	SD	Sum
EvoSuite	63	163.02	111.45	1090.78	175.46	42060
IntelliTest	52	93.91	79.00	245.00	38.73	24792
Randoop	250	712.06	291.00	10276.00	1313.40	127459

Table XIII. Result of mutation analysis (mean numbers computed from 10 runs). Percentages are with respect to the number of all mutants, while mutation score is computed taking into account the equivalent mutants.

Tool	Covered mutants	Killed mutants	Mutation score
CATG	2 079 (33.3%)	1 285 (20.6%)	0.2842
EvoSuite	4 687 (75.2%)	2 886 (46.3%)	0.6381
IntelliTest	5 198 (83.4%)	3 480 (55.8%)	0.7696
jPET	404 (6.5%)	215 (3.4%)	0.0475
Randoop	4 815 (77.2%)	2 818 (45.2%)	0.6231
SPF	3 344 (53.6%)	2 263 (36.3%)	0.5004

tools (these are considered as equivalent mutants). The mutation score has been calculated using the following formula:

$$score = \frac{(killedMutants)}{(allMutants) - (notKilledByAnyTool)}$$

The analysis results were the same for the different runs from CATG, jPET, SPF and IntelliTest. For EvoSuite the results varied significantly, e.g. the number of killed mutants were between 2 832 and 2 972. For Randoop this range was a bit narrower: 2 796–2 897.

The overall lower number of killed mutants can be attributed to the fact that the tests usually just contain a simple assertion on the return value and thus are able to detect fewer modifications in the snippets. Therefore, the results concentrate on the mutant-killing capabilities of the selected test inputs.

The results are mostly similar to results obtained from the coverage analysis. The only notable exceptions are the results for jPET, which are rather low. The reason behind this is a technical problem: jPET generates only textual output even for constructing arrays of objects as test inputs.

DOI: 10.1002/stvr

The SETTE framework has to parse this and construct JUnit test cases from them. However, in many cases this parsing is not successful (e.g. jPET generates an invalid heap, inconsistent array structure or simply too many inputs that do not compile). In these cases the SETTE framework can check whether the coverage reported by jPET is acceptable, but it is not able create the test code for all the generated inputs. Therefore the number of tests used in mutation analysis for jPET are lower than the actually generated test inputs.

With respect to the other tools EVOSUITE and INTELLITEST can also achieve high mutation score, the same as for coverage values. The very large number of generated tests makes it possible for RANDOOP to cover a high number of mutants, and these tests are also able to kill a high number of mutants. SPF follows these three tools, while CATG's score is lower.

In summary the mutation analysis also confirmed that the test inputs generated by INTELLITEST are the most thorough, followed by EVOSUITE and then RANDOOP.

6.2.5. Summary of RQ2 RQ2 showed that even if the tools are able to cover a snippet there are significant differences in the speed of the generation or the size of the tests suites. The analysis of the metrics collected during the evaluation offer additional details about the capabilities of the tools.

6.3. RQ3: Experiments with variable time limit

In the variable time limit experiments the effect of increasing time limit was analyzed. Table XIV summarizes changes in mean coverage, mean size and status (number of snippets categorized as *C*). In the analysis the executions with *N/A*, *EX* and *T/M* status were counted as 0% for coverage and were removed for calculating size. As discussed in Section 5.4.2 these experiments were not performed for INTELLITEST.

For CATG 15 s is too short for two snippets in the Objects category. Moreover, CATG needs 62 s to fully cover `B3a_withContinueBreak`, therefore it is assigned *C* only in 300 s executions. The coverage also increases slightly with the time limit accordingly.

For EVOSUITE the coverage slightly increases, however the size marginally decreases with increasing the available time for searching better test candidates. From the table it seems that with increasing the time limit to 300 s EVOSUITE is able to fully cover nine more snippets. However, EVOSUITE was able to cover sometimes those snippets in the 30 s experiments, just not in enough repetitions to be eligible for a *C* status. For example, `B2d_threeParamsInt` is fully covered out of 10 runs in 2 for 15 s limit, 3 for 30 s limit, 4 for 45 s limit, 7 for 60 s limit, and finally 8 for 300 s limit. The same was the case for ten other snippets.

However, for two O1 snippets quite the contrary happened, these were assigned *C* for 30 s limit, but resulted in an *NC* for 300 s. In the snippets there is a loop in which a method of the target object is called. The upper limit of the loop is part of the generated test input. In the 300 s executions the minimization task succeeded, in the other executions it timed out and rolled back to the generated test suite. Thus in the 300 s experiments fewer tests were retained. Next in the test checking phase EVOSUITE removed those generated tests in every execution that exercised the loop over 10000 times. In the end there remained no tests in the 300 s executions that stepped into the loop.

For jPET increasing the time limit to 60s does not have any effect on the snippets. However, jPET is able to finish 5 snippets (`B2d_quadraticIntNoSolution`, `B2d_threeParamsInt`, `O1_guessImpossibleResultAndOperationCount`, `B2d_threeParamsIntNoSolution` and `O1_guessResultAndOperationCount`) in the 300 s experiments (approximately at 90, 68, 210, 68 s). These snippets all timed out (*T/M*) in the 30 s experiments. In these cases the Prolog solver simply needed more time to handle all constraints generated from the snippets' code.

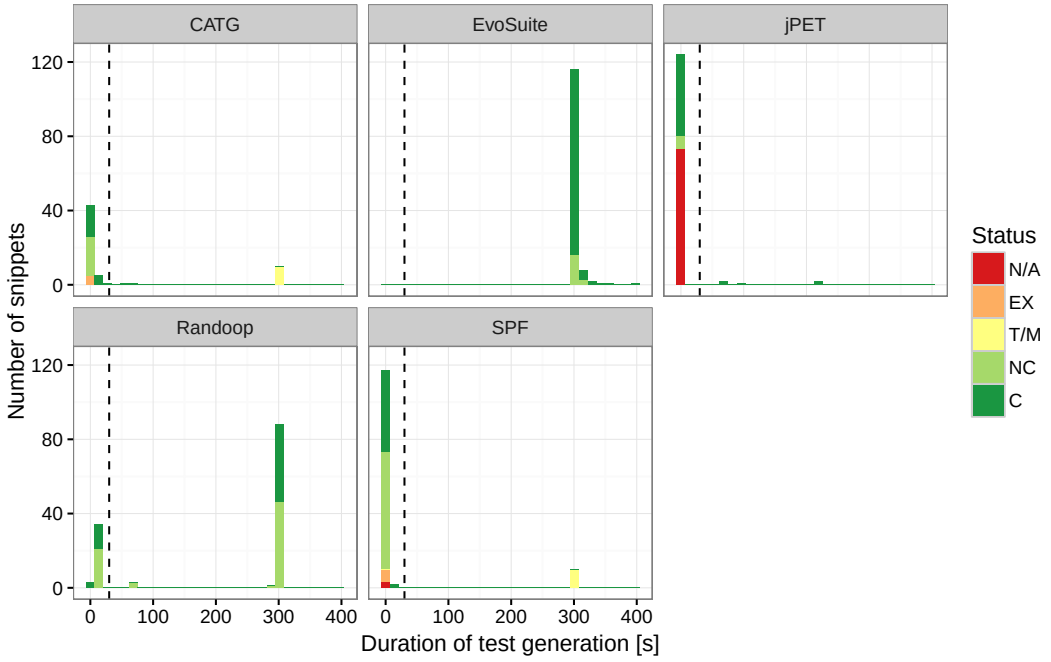
With increasing the time limit RANDOOP generates slightly larger test suites (14.5% increase compared to the 30 s experiments in the mean size), but coverage is only improved minimally. The change in the number of covered snippets is due to snippets `G2_guessInteger` and `G2_guessIntegerNoHelp`. Here the number of executions with *C* status out of the 10 runs are 8 in 30 s, 4 in 45 s, and finally 8 in 300 s). Otherwise there are no major differences in the generated test suites.

DOI: 10.1002/stvr

Table XIV. Effect of increasing the time limit (variable time experiments)

Tool	Metric	15 s	30 s	45 s	60 s	300 s
CATG	Coverage [%]	24.58	26.13	26.13	26.82	27.68
	Size	3.21	4.86	4.86	6.98	9.0
	Nr. of C	21	23	23	24	25
EvoSuite	Coverage [%]	85.82	86.18	86.51	86.60	86.96
	Size	3.73	3.74	3.74	3.67	3.65
	Nr. of C	100	101	103	104	110
jPET	Coverage [%]	31.86	31.86	31.86	31.86	35.09
	Size	14.33	14.33	14.33	14.33	15.27
	Nr. of C	44	44	44	44	49
Randoop	Coverage [%]	71.10	71.40	71.25	71.33	71.40
	Size	959.95	977.52	994.26	1009.32	1101.97
	Nr. of C	57	58	56	58	58
SPF	Coverage [%]	46.28	46.28	46.28	46.28	46.28
	Size	4.07	4.07	4.07	4.07	4.07
	Nr. of C	46	46	46	46	46

Figure 7. Histogram of snippet distributions by durations colored by status with 300s time limit. Dashed line represents 30s, the limit used in the fixed time experiments.



SPF produces the same results for all time limits, expect the snippets with *T/M* status are running longer. The duration of the longest successful execution is 10.56 s, thus the selected 30 s time limit was appropriate for SPF.

Figure 7 depicts the durations of the executions with 300 s time limit. For CATG and jPET the few snippets are visible, which could finish with the increased time limit. EvoSuite tries to use the available time to improve the tests for every snippet. For Randoop the snippets finished before 300 s are those, where the input limit has been reached. For SPF snippets with longer executions are only those that will reach a timeout anyway.

DOI: 10.1002/stvr

Table XV. Number of snippets categorized into the different statuses for all tools (extra features)

Tool	Environment					Multi-threading					Reflection					Native				
	N/A	EX	T/M	NC	C	N/A	EX	T/M	NC	C	N/A	EX	T/M	NC	C	N/A	EX	T/M	NC	C
CATG	1	7	4	10		1		5	3		18			12					2	
EvoSuite				12	10				8	1				23	7		2			
IntelliTest			1	18	2			4	5					26	4				2	
jPET	22					4			5			30					1		1	
Randoop				19	3				8	1				30					2	
SPF		4		17	1			5	4					30					2	

6.3.1. Summary of RQ3 RQ3 analyzed the effect of increasing the time limit available for test generations. Coverage increased minimally (1% on average), but for some of the tools the evaluation status of the snippets changed. It is possible that with a much larger time limit (e.g. 20 minutes per snippet) the results would increase significantly, but from these data sets it seems that the numbers from the 30 s experiments were a good indicator of the capabilities of the tools.

6.4. RQ4: Extra snippets

This section analyses the results of the snippets for handling the environment, multi-threading, reflection and native functions.

Table XV presents the overall categorization of the results for the extra snippets. Note that for these snippets *NC* usually means that the tool called the code with an empty or null parameter, but was not able to handle the given feature.

Because most of the extra snippets were not handled properly by the majority of the tools, only status results are analyzed, and detailed metrics (coverage, size...) are not compared.

6.4.1. Handling the environment Handling the features defined in these snippets could be separated into two tasks. On one hand, the language elements or library calls used in the snippets need to be be processed by the test generators. This is not trivial, half of the tools could not handle this step. After that, the test generator should possibly divert or replace these calls to be able to manipulate their results or it should find a way to replay them when the generated tests are executed.

CATG does not write anything to the standard input, thus the *Env1* snippets are waiting until the time limit is reached. It tries to execute the snippets working with the file system (*Env2*), but only uses empty inputs and could not guess the name of files or directories. CATG throws an exception for the network snippets as it was not able to parse their bytecode.

EvoSuite uses a custom sandbox and Java security manager that by default restricts potential modifications to the environment (e.g. writing to a file). Moreover, it has its own test harness and creates a scaffolding file for every tests that stores e.g. the actual system properties and the number of current threads. EvoSuite has special utility classes that can write to the standard input and with the help of these it can cover snippets in *Env1*. EvoSuite could use strings with valid IPv4 address formats as inputs, and has special wrappers for network addresses and connections and can use these to open virtual network sockets in the tests. With these EvoSuite is able to cover half of the snippets in *Env3*, but does not manage to provide suitable data for the communication protocol. EvoSuite detects in two out of four snippets in *Env4* that they manipulate system dependencies (current time and next random number), and stores the required values for these in the tests.

IntelliTest gives a detailed warning whenever it encounters a dependency that it cannot handle (called testability issues and uninstrumented methods in the tool's output). As IntelliTest is aimed at unit testing, its manual recommends to mock such dependencies. IntelliTest has special mechanisms only for replaying console input and output (using `PexPConsoleInContext` and `PexChoose`), thus these are the only snippets it can fully cover. IntelliTest does not stop when one of the networking snippets waits indefinitely, hence the one *T/M* result.

DOI: 10.1002/stvr

jPET could not handle any snippet in this category, as classes in the Java class library (`File`, `Socket`...) are not supported by the tool.

RANDOOP generates no tests for snippets in *Env1*. For the snippets working with file or directory names it tries a large number of random string. The manual of RANDOOP explicitly warns about running the tool on code working with the file system as the it can delete or create a large number of files. RANDOOP opens various sockets by calling the snippets in *Env3* with random parameters, and is able to trigger numerous exceptions (e.g. address already in use or unknown hostname). RANDOOP does not recognize that the dependencies inside *Env4* need special handling.

SPF throws a `NullPointerException` for all the snippets in *Env1*. SPF calls the snippets in *Env2* and *Env3* with empty or `null` inputs, but reports that the underlying JPF framework does not contain the classes in `java.io`, `java.net`, therefore it could reason about them. It detects only for two out of the four snippets in *Env4* that system properties are accessed.

6.4.2. Multi-threading In most cases the tools could not detect or handle multi-threading code. They simply start the snippets, but they are not aware that its behavior depends on the different threads started inside the code.

CATG only tries to call the snippets, but does not control the threads thus most of the snippets reach a timeout. Moreover, it produces various internal errors for some of the snippets.

EVOSUITE calls the snippets, but is not able to manipulate the threads.

INTELLITEST calls the snippets, but is not able to manipulate the threads. One snippet in *T1*, and most of the ones in *T3* exceed the time limit and INTELLITEST does not stop them forcefully.

jPET could not handle most of the snippets. It calls the ones without parameters, but is not able to manipulate the threads started by the snippets.

RANDOOP can start the snippets, but is not able to manipulate the threads. For the snippet that starts a configurable number of new threads given as a parameter RANDOOP produces an `OutOfMemoryError` by starting too many threads.

SPF calls the snippets but could not extract path conditions from their codes. However, it properly detects the deadlock.

6.4.3. Reflection The tools are usually not able to handle class or method type parameters, and could not guess class or method names either.

CATG could not handle snippets that have a class type as their parameter. CATG only tries the empty string for the other snippets expecting a class name to load.

jPET could not load the bytecode of any snippets in this category, and throws an exception for each of them.

INTELLITEST is able to generate objects with different types and cover 4 snippets in *R3*. However, for the snippets that have `Type` as parameter, it only tries some built-in types from `System.Reflection`, but not the primitive types or types in the current assembly. INTELLITEST could not guess that string parameters are type or method names (*R1b* and *R2b*).

EVOSUITE is able to work with class types, and generates inputs from built-in classes (`String`, `Integer`) and also from custom classes in the tested code. However, it is not able to generate method types or guess that these snippets with string parameters are waiting class or method names (e.g. `java.lang.String`). EVOSUITE generates objects from the classes defined in the tested code. Altogether EVOSUITE is able to cover 3 snippets in *R1*, 0 in *R2*, and 4 in *R3*.

RANDOOP could not generate any tests for the snippets accepting a class or method type. For the other snippets accepting a class or method name string it tries numerous random strings until it reaches its input limit. For the snippets in *R3* accepting an object it tries to instantiate various objects and then casts them to `java.lang.Object`

SPF calls every snippet with `null` inputs, but could not extract any path constraints from the snippets.

6.4.4. Handling native functions The snippets calling the native library were executed separately, because they needed special handling when running the experiments (e.g. compiling and loading

DOI: 10.1002/stvr

the native library). The native functions called by the snippets have an integer parameter, and the functions return different values based on whether the parameter is equal to a hard-coded "magic number" or not.

CATG tried only 0 and 1 as inputs. jPET throw an exception for one of the snippets, and called the other with only 1 as a parameter. INTELLITEST detected that the code was calling an external method and warns that it could not generate inputs to cover that code. EVOSUITE tried to load the native library for its mocking runtime, but was unable and stopped with a `MockRuntimeException` without generating any tests. RANDOOP tried various random integer values, but was not able to guess the magic numbers. SPF generated only one test case for each snippet with 1 as an input value.

To summarize, neither tool was able to handle the native functions.

6.4.5. Summary of RQ4 Snippets for the extra features posed a significant challenge for most of the tools. Except for INTELLITEST and EVOSUITE they usually could not handle the required language constructs and library calls, or were not able to affect the execution of the snippets. RQ4 confirmed that these features are indeed still challenging for test generator tools.

7. DISCUSSION

7.1. Experiences with the tools

While performing the experiments we had to work extensively with the test generator tools. There are differences not only in the capabilities of the tools, but in their usability and level of maturity.

These tools belong to the following informal categories:

1. prototype tool used only by its authors,
2. tool used already by other users for limited case studies,
3. tool used already in several, complex case studies,
4. tool with commercial support used already in several, complex case studies.

Note, there are more exact methods to categorize maturity, e.g. Technology Readiness Levels (TRL) [43]. The goal here is not to evaluate the tools with respect to these, rather to share our subjective experiences about the tools.

Interested readers are recommended to look into the source of the tool-specific `*Runner` and `*Parser` classes in the SETTE framework to see how many special cases needed to be handled for each of the tools.

7.1.1. CATG belongs to the first category, it is a research prototype. It is open source, but there is no documentation for the tool. CATG generates only test inputs written to the standard output as plain text. It is not able to reliably handle exceptions coming from the code under test and can produce various errors.

7.1.2. EVOSUITE belongs to the third category. EVOSUITE is now open source, and is actively developed. It has been extensively used with complex case studies, and is stable and robust. It could handle various exceptions from the code under test and detect infinite loops. EVOSUITE monitors also itself (it starts a master process, and tests are generated in separate child processes), and can tolerate internal errors. Logging is available, and there are lots of configuration parameters (although there is minimal documentation for them and they are not always evident to an outsider). EVOSUITE generates not just test inputs, but full test code. It has several additional features, e.g. test suite minimization, non-trivial assertions or running tests and removing problematic ones. EVOSUITE uses a custom test harness to handle calls to the environment. However, one consequence of being a randomized search-based tool is that every execution could be different, which can be a new concept for some developers. (EVOSUITE has plugins for the Eclipse and IntelliJ IDEs, but we have no experiences with them.)

DOI: 10.1002/stvr

7.1.3. jPET belongs to the first category. It is not developed any more, and only a binary is available. jPET only generates a plain text output describing the test inputs. Objects are generated in the following format:

```
[ref(A),-100000] Heap = heap([ (A,object('.../SimpleObject',
  [field('operationCount:I',0)|B]))|C],D)
```

Moreover, as it can create large output files (in the range of 10 MBs) and sometimes invalid heaps, parsing its output required much effort.

7.1.4. RANDOOP belongs to the third category. It is open source, actively developed, and has a good manual explaining its behavior and parameters. RANDOOP is stable and could handle the exceptions of the tested code. However, due to the its random testing nature it could quickly generate really large number of tests, which can pose its own challenges. For example, just compiling the tests with Ant lasts 5 minutes and it requires increasing memory allocated to Ant to at least 4 GB. The test execution times are also considerably longer than for the other tools. Fortunately, the number of generated tests cases can be limited with various configuration parameters.

7.1.5. PEX / INTELLITEST now belongs to the fourth category, it is commercially supported in Visual Studio 2015. PEX was a really mature tool, with extensive documentation and tutorials, detailed reports, integration with Visual Studio and offered customization through command line parameters or .NET attributes attached to the code. INTELLITEST kept the PEX engine, however, the reporting and customization mechanisms have not been ported yet (or at least they are not available on its public UI). INTELLITEST integrates seamlessly into Visual Studio, but due to lack of a command line interface or an API it is hard to use it in experiments. A unique feature of INTELLITESTamong the evaluated tools are the helpful warning and error messages and great default limits (called boundaries). For example, INTELLITEST tries to guess how various classes can be instantiated, and if it is not successful, then it asks the user to supply a factory method. Every step of the test generation has boundaries with short default values, which makes test generation really fast as INTELLITEST is intended to be used during development by the developer inside an IDE.

7.1.6. SPF belongs to the second category. There are users outside of the core contributors, but it is not as stable as some of the other tools. SPF is open source, and has only some limited documentation. SPF generates plain text output with the input values and the symbolic constraints. It is not able to handle all the exceptions from the code under test. However, with the not so frequent newer versions it is improving.

8. RELATED WORK

This section reviews the surveys, benchmarks and experiments related to test input generators.

8.1. Survey papers

There are several recent *survey papers* about test input generation. Anand *et al.* [2] performed an orchestrated survey about different methods for test generation (namely symbolic execution, model-based testing, combinatorial testing, adaptive random testing and search-based software testing). Regarding symbolic execution Păsăreanu and Visser [44] summarized actual research directions, Cadar *et al.* [45] collected experiences from tool developers and Chen *et al.* [19] listed current challenges. Regarding search-based software testing McMinn [4] surveyed SBST approaches focusing on the different algorithms used, Ali *et al.* [46] investigated the empirical assessment of SBST papers, and Harman *et al.* [21] presented the trends in SBST research and open problems regarding testing non-functional properties. Arcuri *et al.* [5] analyze random testing and present its theoretical and real-world results.

DOI: 10.1002/stvr

These papers give an excellent overview of the topic, but they provide general and not tool-specific observations.

8.2. Benchmarks

The experiments of tool papers usually use their own set of code samples, thus their results are not directly comparable across tools. To overcome this the Software-artifact Infrastructure Repository (SIR) [47] makes available programs together with test suites or fault data commonly used in software testing research (e.g. the so called Siemens suite or the space program). More recently, Fraser and Arcuri recommended the SF100 *benchmark* [48], a representative selection of 100 open source projects from SourceForge. The Defects4J [49] dataset consists of 357 real faults from five open source projects along with developer-written tests. The SBST Java Unit Testing Tool Contest [33, 34] invited tool developers to run their tool on several Java classes selected from open source projects, and the tools were ranked based on the coverage and mutation score achieved and the time utilized. Benchmarks can be created automatically, e.g. RUGRAT [50] is a flexible tool for generating Java programs that can serve as benchmarks for program analysis and testing tools.

8.3. Experiments

Several real-world *experiments* were performed to evaluate test generators. Lakhotia *et al.* [7] investigated the coverage of CUTE and AUSTIN (a search-based tool) on five open source components. Braione *et al.* [51] performed an experiment on an industrial control software using CREST, PEX and AUSTIN. Qu and Robinson [8] measured the coverage of CREST and KLEE on a 3.9M LOC realtime embedded system. Wang *et al.* [52] compared automatically generated tests by the KLEE tool with manual tests on 40 programs from the CoreUtils package. Fraser and Arcuri [29] performed a large-scale evaluation of EVOSUITE on the extended SF110 benchmark. Gay *et al.* [53] compared test inputs generated to achieve different coverage criteria with randomly generated ones on 7 industrial systems. Shamshiri *et al.* [41] performed experiments with RANDOOP, EVOSUITE and AGITARONE on real faults from the Defects4J dataset. Eler *et al.* [22] measured how frequent are some of the challenging features in 147 open source Java projects.

These papers provide a general feedback about the capabilities and limitations of the tools on real code. However, as they experimented on a large code base, it is harder to trace back their findings. Our approach complements these results by providing a small-scale but directed code base.

Galler and Aichernig [9] presented a survey on the capabilities of 7 test data generator tools. The goal and approach of this paper was similar to ours (e.g. they also checked primitive types and objects). Their benchmark suite was smaller (31 code fragments) and its code is not available. However, they also provided valuable feedback, and evaluated several tools which were not covered in our work (AGITARONE [54], CODEPRO ANALYTIX[††], AUTOTEST [55], C++TEST [56] and JTEST [57]). Their general conclusion was the same: most tools handled the simple cases, but some of the tools did not manage more complex constructs (e.g. non-linear conditions over primitive types or constraints over objects' attributes). Their specific results for the tools in common were also similar to ours: PEX performed well by handling all fragments, while RANDOOP had mixed results (although our detailed results suggest that RANDOOP has improved in the recent years by handling more types).

8.4. Related problems

A related problem is comparing *static analysis tools*. The Juliet test suite [58] employed a similar approach to the one used in this paper: 181 security weaknesses were collected (e.g. improper buffer handling) and synthetic C/C++ and Java programs were created for them. The test suite consists of "good" and "bad" program versions, the "bad" ones containing exactly one flaw representing a weakness. The static analysis tools can be then compared based on how many or what types of flaws they can detect.

[††]CodePro AnalytiX was acquired by Google, but it is not available anymore.

DOI: 10.1002/stvr

Another related problem is testing and comparing code *compilers*, although in that case research [59] focused on generating test programs from syntactic and semantic definition rules.

9. CONCLUSION

The goal of this paper was to compare and evaluate test input generator tools. Based on the current challenges and the language constructs of imperative C-like languages a set of features was identified that these tools should cover, and 363 code snippets were designed representing these features. Of these snippets 300 targeted core features (conditionals, objects, generics, etc.), while 63 contained extra features like multi-threading or handling the environment and dependencies.

A framework called SETTE was implemented that can automatically perform experiments and evaluations on test generators using these snippets. Several experiments were performed on six different tools including ones based on symbolic execution (CATG, INTELLITEST, jPET, SPF), search-based (EVOSUITE) and random testing (RANDOOP). Although initially some of the features have specifically targeted symbolic execution, the experiments showed that they could provide feedback on tools with different underlying techniques.

The results showed that the evaluation can identify both strengths and weaknesses in the tools. INTELLITEST and EVOSUITE performed very well on the 300 core snippets both with respect to achieved code coverage and size of the generated test suite. RANDOOP was not able to fully cover a greater number of snippets, but generated tests for all the snippets. The three other tools produced exceptions or timeouts for some of the snippets. The extra snippets posed a greater challenge, most tools were not able to handle them.

Both new tools or code snippets can be easily added to extend our work. For example, currently mostly academic tools were included, but there exists some commercial tools (e.g. AGITARONE or JTEST) that could be integrated with SETTE. The extra snippets only covered the basics of environmental dependencies, multi-threading or native calls.

All the source code and results are available at http://sette-testing.github.io. We hope that our results would provide useful insights both for tool developers and users.

ACKNOWLEDGEMENT

The authors would like to thank Ágnes Salánki for her help with the visualization of the results, and the anonymous reviewers for their detailed and valuable comments.

REFERENCES

1. Institute of Electrical and Electronics Engineers. *Systems and software engineering – Vocabulary* 12 2010, doi:10.1109/IEEESTD.2010.5733835. Standard 24765:2010.
2. Anand S, Burke EK, Chen TY, Clark J, Cohen MB, Grieskamp W, Harman M, Harrold MJ, McMinn P. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Software* 2013; **86**(8):1978 – 2001, doi:10.1016/j.jss.2013.02.061.
3. Godefroid P. Test Generation Using Symbolic Execution. *Annual Conf. on FSTTCS*, 2012; 24–33, doi:10.4230/LIPIcs.FSTTCS.2012.24.
4. McMinn P. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability* 2004; **14**(2):105–156, doi:10.1002/stvr.294.
5. Arcuri A, Iqbal MZ, Briand L. Random testing: Theoretical results and practical implications. *Software Engineering, IEEE Transactions on* 2012; **38**(2):258–277, doi:10.1109/TSE.2011.121.
6. Bounimova E, Godefroid P, Molnar D. Billions and billions of constraints: Whitebox fuzz testing in production. *Proc. of the Int. Conf. on Software Engineering*, ICSE '13, IEEE, 2013; 122–131, doi:10.1109/ICSE.2013.6606558.
7. Lakhotia K, McMinn P, Harman M. An empirical investigation into branch coverage for C programs using CUTE and AUSTIN. *J. Syst. Softw.* Dec 2010; **83**(12):2379–2391, doi:10.1016/j.jss.2010.07.026.
8. Qu X, Robinson B. A case study of concolic testing tools and their limitations. *Int. Symp. on Empirical Software Engineering and Measurement*, ESEM'11, 2011; 117–126, doi:10.1109/ESEM.2011.20.
9. Galler SJ, Aichernig BK. Survey on test data generation tools. *STTT* 2014; **16**(6):727–751, doi:10.1007/s10009-013-0272-3.
10. Cseppentő L, Micskei Z. Evaluating symbolic execution-based test tools. *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th Int. Conf. on*, IEEE, 2015; 1–10, doi:10.1109/ICST.2015.7102587.
11. Myers GJ. *Art of Software Testing*. John Wiley & Sons, Inc.: New York, NY, USA, 1979.

12. Miller EF, Melton RA. Automated generation of testcase datasets. *SIGPLAN Not.* 1975; **10**(6):51–58, doi:10.1145/390016.808424.
13. Howden WE. Methodology for the generation of program test data. *Computers, IEEE Transactions on* 1975; **C-24**(5):554–560, doi:10.1109/T-C.1975.224259.
14. DeMillo RA, Offutt AJ. Constraint-based automatic test data generation. *Software Engineering, IEEE Transactions on* 1991; **17**(9):900–910, doi:10.1109/32.92910.
15. Boyapati C, Khurshid S, Marinov D. Korat: Automated testing based on Java predicates. *Int. Symp. on Software Testing and Analysis*, ISSTA '02, ACM, 2002; 123–133, doi:10.1145/566172.566191.
16. King JC. Symbolic execution and program testing. *Commun. ACM* 1976; **19**(7):385–394, doi:10.1145/360248.360252.
17. Harman M, Mansouri SA, Zhang Y. Search based software engineering: A comprehensive analysis and review of trends techniques and applications. *Technical Report TR-09-03*, Dept. of Computer Science, King's College London 2009.
18. Micskei Z, Madeira H, Avritzer A, Majzik I, Vieira M, Antunes N. Robustness testing techniques and tools. *Resilience Assessment and Evaluation of Computing Systems*. Springer, 2012; 323–339, doi:10.1007/978-3-642-29032-9_16.
19. Chen T, Zhang Xs, Guo Sz, Li Hy, Wu Y. State of the art: Dynamic symbolic execution for automated test generation. *Future Generation Computer Systems* 2013; **29**(7):1758 – 1773, doi:10.1016/j.future.2012.02.006.
20. McMinn P. Search-based software testing: Past, present and future. *Software Testing, Verification and Validation Workshops (ICSTW)*, 2011; 153–163, doi:10.1109/ICSTW.2011.100.
21. Harman M, Jia Y, Zhang Y. Achievements, open problems and challenges for search based software testing. *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th Int. Conf. on*, 2015; 1–12, doi:10.1109/ICST.2015.7102580.
22. Eler MM, Endo AT, Durelli VH. An empirical study to quantify the characteristics of Java programs that may influence symbolic execution from a unit testing perspective. *Journal of Systems and Software* 2016; doi:10.1016/j.jss.2016.03.020.
23. Sen K. CATG web page. https://github.com/ksen007/janala2 2015. Last accessed on 20/6/2016.
24. Fraser G, Arcuri A. Whole test suite generation. *Software Engineering, IEEE Transactions on* 2013; **39**(2):276–291, doi:10.1109/TSE.2012.14.
25. Albert E, Gómez-Zamalloa M, Puebla G. PET: a partial evaluation-based test case generation tool for Java bytecode. *Proc. of workshop on Partial evaluation and program manipulation*, PEPM'10, ACM, 2010; 25–28, doi:10.1145/1706356.1706363.
26. Tillmann N, Halleux J. Pex – white box test generation for .NET. *Tests and Proofs*, *LNCS*, vol. 4966. Springer, 2008; 134–153, doi:10.1007/978-3-540-79124-9_10.
27. Pacheco C, Lahiri SK, Ernst MD, Ball T. Feedback-directed random test generation. *Int. Conf. on Software Engineering*, ICSE'07, 2007; 75–84, doi:10.1109/ICSE.2007.37.
28. Păsăreanu CS, Visser W, Bushnell D, Geldenhuys J, Mehlitz P, Rungta N. Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis. *Automated Software Engineering* 2013; **20**(3):391–425, doi:10.1007/s10515-013-0122-2.
29. Fraser G, Arcuri A. A large-scale evaluation of automated unit test generation using EvoSuite. *ACM Trans. Softw. Eng. Methodol.* 2014; **24**(2):8:1–8:42, doi:10.1145/2685612.
30. Pinto G, Torres W, Fernandes B, Castor F, Barros RS. A large-scale study on the usage of Java's concurrent programming constructs. *Journal of Systems and Software* 2015; **106**:59 – 81, doi:10.1016/j.jss.2015.04.064.
31. Grechanik M, McMillan C, DeFerrari L, Comi M, Crespi S, Poshyvanyk D, Fu C, Xie Q, Ghezzi C. An empirical investigation into a large-scale Java open source code repository. *Proc. of the Int. Symp. on Empirical Software Engineering and Measurement*, ESEM '10, ACM, 2010; 11:1–11:10, doi:10.1145/1852786.1852801.
32. Hoffmann MR. JaCoCo Java code coverage library. http://www.eclemma.org/jacoco/ 2016. Last accessed on 20/6/2016.
33. Bauersfeld S, Vos TEJ, Lakhotia K. Unit testing tool competitions – lessons learned. *Future Internet Testing*, *LNCS*, vol. 8432. Springer, 2014; 75–94, doi:10.1007/978-3-319-07785-7_5.
34. Rueda U, Vos TEJ, Prasetya I. Unit testing tool competition – round three. *Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th Int. Workshop on*, 2015; 19–24, doi:10.1109/SBST.2015.12.
35. Inozemtseva L, Holmes R. Coverage is not strongly correlated with test suite effectiveness. *Int. Conf. on Software Engineering*, ICSE'14, ACM, 2014; 435–445, doi:10.1145/2568225.2568271.
36. Fraser G, Staats M, McMinn P, Arcuri A, Padberg F. Does automated unit test generation really help software testers? A controlled empirical study. *ACM Trans. Softw. Eng. Methodol* 2015; **24**(4):23:1–23:49, doi:10.1145/2699688.
37. Offutt J. A mutation carol: Past, present and future. *Information and Software Technology* 2011; **53**(10):1098 – 1107, doi:10.1016/j.infsof.2011.03.007.
38. Just R. The Major mutation framework: Efficient and scalable mutation analysis for Java. *Proc. of the Int. Symp. on Software Testing and Analysis (ISSTA)*, 2014; 433–436, doi:10.1145/2610384.2628053.
39. Andrews JH, Briand LC, Labiche Y, Namin AS. Using mutation analysis for assessing and comparing testing coverage criteria. *Software Engineering, IEEE Transactions on* 2006; **32**(8):608–624, doi:10.1109/TSE.2006.83.
40. Arcuri A, Briand L. A hitchhiker's guide to statistical tests for assessing randomized algorithms in software engineering. *Software Testing, Verification and Reliability* 2014; **24**(3):219–250, doi:10.1002/stvr.1486.
41. Shamshiri S, Just R, Rojas JM, Fraser G, McMinn P, Arcuri A. Do automatically generated unit tests find real faults? An empirical study of effectiveness and challenges. *Int. Conf. on Automated Software Engineering (ASE)*, ACM, 2015; 201–211.
42. R Core Team. *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing, Vienna, Austria 2015. URL https://www.R-project.org/.
43. Mankins JC. *Technology Readiness Levels*. NASA 1995. White paper.

44. Păsăreanu CS, Visser W. A survey of new trends in symbolic execution for software testing and analysis. *Int. Journal on Software Tools for Technology Transfer* 2009; **11**(4):339–353, doi:10.1007/s10009-009-0118-1.
45. Cadar C, Godefroid P, Khurshid S, Păsăreanu CS, Sen K, Tillmann N, Visser W. Symbolic execution for software testing in practice: preliminary assessment. *Proc. of the 33rd Int. Conf. on Software Engineering*, ICSE '11, ACM, 2011; 1066–1071, doi:10.1145/1985793.1985995.
46. Ali S, Briand LC, Hemmati H, Panesar-Walawege RK. A systematic review of the application and empirical investigation of search-based test case generation. *Software Engineering, IEEE Transactions on* 2010; **36**(6):742–762, doi:10.1109/TSE.2009.52.
47. Do H, Elbaum S, Rothermel G. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering* 2005; **10**(4):405–435, doi:10.1007/s10664-005-3861-2.
48. Fraser G, Arcuri A. Sound empirical evidence in software testing. *Int. Conf. on Software Engineering, ICSE'12*, 2012; 178–188, doi:10.1109/ICSE.2012.6227195.
49. Just R, Jalali D, Ernst MD. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. *Int. Symp. on Software Testing and Analysis*, ISSTA, 2014; 437–440, doi:10.1145/2610384.2628055.
50. Hussain I, Csallner C, Grechanik M, Xie Q, Park S, Taneja K, Hossain BMM. RUGRAT: Evaluating program analysis and testing tools and compilers with large generated random benchmark applications. *Software: Practice and Experience* 2014; doi:10.1002/spe.2290.
51. Braione P, Denaro G, Mattavelli A, Vivanti M, Muhammad A. Software testing with code-based test generators: data and lessons learned from a case study with an industrial software component. *Software Qual J* 2014; **22**(2):311–333, doi:10.1007/s11219-013-9207-1.
52. Wang X, Zhang L, Tanofsky P. Experience report: How is dynamic symbolic execution different from manual testing? A study on KLEE. *Proc. of the 2015 Int. Symp. on Software Testing and Analysis*, ISSTA 2015, ACM, 2015; 199–210, doi:10.1145/2771783.2771818.
53. Gay G, Staats M, Whalen M, Heimdahl MPE. The risks of coverage-directed test case generation. *Software Engineering, IEEE Transactions on* Aug 2015; **41**(8):803–819, doi:10.1109/TSE.2015.2421011.
54. Boshernitsan M, Doong R, Savoia A. From Daikon to Agitator: Lessons and challenges in building a commercial tool for developer testing. *Proceedings of the 2006 Int. Symp. on Software Testing and Analysis*, ISSTA '06, ACM, 2006; 169–180, doi:10.1145/1146238.1146258.
55. Meyer B, Fiva A, Ciupa I, Leitner A, Wei Y, Stapf E. Programs that test themselves. *Computer* 2009; **42**(9):46–55, doi:10.1109/MC.2009.296.
56. Parasoft. Parasoft C/C++test. https://www.parasoft.com/product/cpptest/ 2016. Last accessed on 20/6/2016.
57. Parasoft. Parasoft Jtest. https://www.parasoft.com/product/jtest/ 2016. Last accessed on 20/6/2016.
58. Boland T, Black PE. Juliet 1.1 C/C++ and Java test suite. *Computer* Oct 2012; **45**(10):88–90, doi:10.1109/MC.2012.345.
59. Kossatchev A, Posypkin M. Survey of compiler testing methods. *Programming and Computer Software* 2005; **31**(1):10–19, doi:10.1007/s11086-005-0002-z.