SOFTWARE TESTING, VERIFICATION AND RELIABILITY
*Softw. Test. Verif. Reliab.* 2015; **00**:1–25
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, Hungary*

## SUMMARY

In recent years several tools have been developed to automatically select relevant test inputs from the source 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 we collected a representative set of programming language concepts that should be handled by the tools, mapped them to 300 code snippets that would serve as inputs for the tools, created an automated framework to execute and evaluate these snippets, and performed experiments on six 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 our research could serve as actionable feedback to tool developers and help practitioners assess the readiness of test input generation. Copyright © 2015 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 code of the system under test and select test inputs typically 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* (SE) [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 research prototypes, and the method is not used widely in industry.

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

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

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

Copyright © 2015 John Wiley & Sons, Ltd.


This is the pre-peer reviewed version, which has been published in final form at https://doi.org/10.1002/stvr.1627. The final version was significantly extended, please see the final version.
This article may be used for non-commercial purposes in accordance with Wiley Terms and Conditions for Self-Archiving.

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 have been published in which tools were applied to complex software [6, 7]. 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 [8] containing *fine-grained feedback* on what pieces of code can or cannot be handled by a certain tool. Thus the initial question that motivated our 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.

We applied the following *method* in our research. First, we collected and organized the programming language concepts that should be handled by a test input generator (e.g. recursion, complex structures). Then, we defined minimal code snippets 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.

In our initial paper [9] we concentrated on tools using symbolic execution. This paper extends the scope of the work by including search-based and random testing tools, adding two new tools to the automatic evaluation framework, performing a large number of experiments and extending the analysis from only code coverage to several other properties.

Thus *contributions* of this paper include:

- collecting the relevant and challenging features of imperative programming languages w.r.t. test input generation;
- mapping these features to the Java and .NET languages and platforms and implementing them in 300 code snippets;
- creating an easily extendable framework called SETTE for automatically evaluating the snippets on selected tools;
- designing and conducting a large number of experiment on five Java and one .NET tools,
- analyzing the results by highlight 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`

## 2. OVERVIEW

Structural or white-box testing is a well-known method since the early days of software testing [10]. Even in the 1970s several algorithms and tools were proposed for automatic test input generation starting from the source code [11, 12]. Since then numerous techniques have been developed (e.g.

Copyright © 2015 John Wiley & Sons, Ltd.


DOI: 10.1002/stvr

Table I. List of evaluated test input generator tools

| Name | Platform | OS | Access | Published | Updated | Technique |
|---|---|---|---|---|---|---|
| CATG [24] | Java | – | open source | 2012 | 2015 | SE |
| EvoSuite [25] | Java | – | closed source | 2008 | 2015 | SBST |
| PET/jPET [26] | Java | Linux | open source | 2009 | 2011 | SE |
| Pex [27] | .NET | Windows | closed source | 2008 | 2015 | SE |
| Randoop [28] | Java | – | open source | 2007 | 2015 | RT |
| SPF [29] | Java | – | open source | 2012 | 2015 | SE |

using mutation analysis [13] or pre-/postconditions [14], etc.). In this work we focus 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) [15]. 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 forms a search space and the test adequacy criterion is coded as a fitness function [16].

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 [17]. 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*. For SE [2, 18, 19, 20], currently the most important ones to deal with include path explosion (exponentially increasing number of possible execution paths), complex and external arithmetic functions (due to the limitations of SMT solvers), floating-point calculations, pointer operations, interaction with the environment and multi-threading. Interaction with environment and multi-threading are also challenging for an SBST tool [21]. Further SBST challenges include predicates with only a flag variable, nested conditions and enumerations [22], complex strings and objects with internal state [4], or nun-functional requirements [23].

**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).

In our experiments we concentrated on tools for Java, we evaluated the following tools *CATG* [24], *EvoSuite* [25], *PET* [26], *Randoop* [28], and *Symbolic PathFinder (SPF)* [29] and included additional results for *Pex* [27]. Table I contains some details about the tools.

**Our approach** Our goal was to compare test input generators. The overview of our approach is illustrated on Fig. 1.

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

Copyright © 2015 John Wiley & Sons, Ltd.

*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

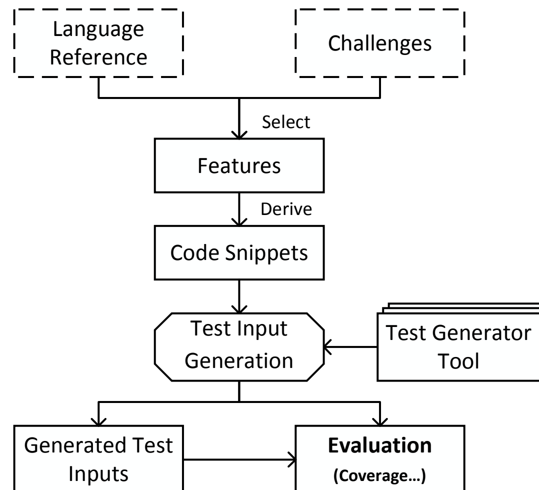



Figure 1. Our approach for comparing test input generators

1. We have collected the common language elements and program organizational structures for C/C++, Java and .NET languages ranging from basic data types and operators to complex concepts like handling string manipulations or class inheritance. We would refer to these collectively as *"features"*. We paid attention to include the ones responsible for the challenges above.
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. An ideal tool should generate such a set of inputs for each snippet, which reaches the maximal possible coverage.
4. Using these results a detailed feedback can be given on one tool and several tools can be compared.

The next sections detail the features (Section 3), the code snippets (Section 4), the planning of the experiment (Section 5) and the discussion of the results (Section 6).

## 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. Our guidelines during the selection of the features were the following:

- *Coverage:* in order to get basic and detailed feedback on the tools, the most important 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.
- *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.

Copyright © 2015 John Wiley & Sons, Ltd.

*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

Table II. Selected 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 |

- *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.

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 code which can be called several times, but does not belong to any high-level constructs, 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.

Table II lists the selected features, whose details will be discussed in the next subsections.

### 3.1. 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

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

Copyright © 2015 John Wiley & Sons, Ltd.


DOI: 10.1002/stvr

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, complex loops and embedded 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.

### *3.2. 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

- 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.

### *3.3. 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 are not only structures with functions, but they 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 objects, iii) guess the concrete type or create dummy classes 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 an object whose integer field is ensured to be positive, a test input generator should never create an instance of the object, 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.

### *3.4. 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 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.

### *3.5. 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

Copyright © 2015 John Wiley & Sons, Ltd.


DOI: 10.1002/stvr

contains 4024 classes), our target was 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.

### 3.6. 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 perform symbolic execution even in 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.

### 3.7. 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 84% 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
- `native`: used when the bytecode has attached native implementations
- `transient`: used to prevent the serialization of certain fields
- `strictfp`: ensures that floating-point precision is the same on any platform
- `synchronized`, `volatile`: used in connection with multi-threading

**Challenges** The features cover the following challenges from Section 2:

- *Path explosion:* loops (B3) and recursion (B5)
- *Complex and extern 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)

The following challenges were not covered in the selected features.

- *Interaction with the environment:* was not covered
- *Pointer operations:* as our targets were managed languages we have not covered them
- *Multi-threading:* was not covered
- *Non-functional requirements*: was not covered

We aimed for a set of features and code snippets that is 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 our 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 our work.

Copyright © 2015 John Wiley & Sons, Ltd.

*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

## 4. IMPLEMENTATION

We mapped the features defined in Section 3 to the Java language, and created 300 code snippets implementing them. The details of the snippets in each category are shown on Table III and Table IV. The category and method names could serve as an overview about the detailed features selected. The columns NCSS and CCN give the non commenting source statements and the cyclomatic complexity number[¶]. As it can be seen, the majority of the code snippets focuses 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.

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;
  }
}
```

Listing 2: Defined sample inputs for the example code snippet

```
public static SnippetInputContainer guessObject() {
  SnippetInputContainer inputs = new SnippetInputContainer(1);

  inputs.addByParameters((Object) null);

  SimpleObject obj = new SimpleObject();
  obj.addAbs(3);
  obj.addAbs(0);
  inputs.addByParameters(obj);

  inputs.addByParameters(new SimpleObject());

  return inputs;
}
```

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 sample snippet all statements in

[¶]NCSS and CCN were measured with JavaNCSS (`http://www.kclee.de/clemens/java/javancss/`). Note: some snippets call other functions, these are not included in the numbers.

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table III. Details of code snippets by categories

| Category | | | Method | NCSS | CCN |
|---|---|---|---|---|---|
| Basic | B1 Types and operators | B1a PrimitiveTypes | oneParamBoolean(boolean) | 2 | 1 |
| | | | twoParamBoolean(boolean,boolean) | 2 | 1 |
| | | | oneParamByte(byte) | 2 | 1 |
| | | | twoParamByte(byte,byte) | 2 | 1 |
| | | | oneParamChar(char) | 2 | 1 |
| | | | twoParamChar(char,char) | 2 | 1 |
| | | | oneParamDouble(double) | 2 | 1 |
| | | | twoParamDouble(double,double) | 2 | 1 |
| | | | oneParamFloat(float) | 2 | 1 |
| | | | twoParamFloat(float,float) | 2 | 1 |
| | | | oneParamInt(int) | 2 | 1 |
| | | | twoParamInt(int,int) | 2 | 1 |
| | | | oneParamLong(long) | 2 | 1 |
| | | | twoParamLong(long,long) | 2 | 1 |
| | | | oneParamShort(short) | 2 | 1 |
| | | | twoParamShort(short,short) | 2 | 1 |
| | | B1b Constants | constBoolean() | 2 | 1 |
| | | | constByte() | 2 | 1 |
| | | | constChar() | 2 | 1 |
| | | | constFloat() | 2 | 1 |
| | | | constInt() | 2 | 1 |
| | | | constLong() | 2 | 1 |
| | | | constDouble() | 2 | 1 |
| | | | constShort() | 2 | 1 |
| | | B1c Arithmetical | add(int,int) | 2 | 1 |
| | | | addAssignment(int,int) | 3 | 1 |
| | | | divide(int,int) | 2 | 1 |
| | | | divideAssignment(int,int) | 3 | 1 |
| | | | modulo(int,int) | 2 | 1 |
| | | | moduloAssignment(int,int) | 3 | 1 |
| | | | multiply(int,int) | 2 | 1 |
| | | | multiplyAssignment(int,int) | 3 | 1 |
| | | | substract(int,int) | 2 | 1 |
| | | | substractAssignment(int,int) | 3 | 1 |
| | | | postfixMinusMinus(int) | 3 | 1 |
| | | | postfixPlusplus(int) | 3 | 1 |
| | | | unaryMinus(int) | 2 | 1 |
| | | | unaryMinusMinus(int) | 3 | 1 |
| | | | unaryPlus(int) | 2 | 1 |
| | | | unaryPlusPlus(int) | 3 | 1 |
| | | B1d Relational | and(boolean,boolean) | 2 | 2 |
| | | | equal(int,int) | 2 | 1 |
| | | | greater(int,int) | 2 | 1 |
| | | | greaterOrEqual(int,int) | 2 | 1 |
| | | | negate(boolean) | 2 | 1 |
| | | | notEqual(int,int) | 2 | 1 |
| | | | or(boolean,boolean) | 2 | 2 |
| | | | smaller(int,int) | 2 | 1 |
| | | | smallerOrEqual(int,int) | 2 | 1 |
| | | B1e Bitwise | and(int,int) | 2 | 1 |
| | | | andAssignment(int,int) | 3 | 1 |
| | | | negate(int) | 2 | 1 |
| | | | or(int,int) | 2 | 1 |
| | | | orAssignment(int,int) | 3 | 1 |
| | | | shiftLeft(int,int) | 2 | 1 |
| | | | shiftLeftAssignment(int,int) | 3 | 1 |
| | | | shiftRight(int,int) | 2 | 1 |
| | | | shiftRightAssignment(int,int) | 3 | 1 |
| | | | shiftRightUnsigned(int,int) | 2 | 1 |
| | | | shiftRightUnsignedAssignment(int,int) | 3 | 1 |
| | | | xor(int,int) | 2 | 1 |
| | | | xorAssignment(int,int) | 3 | 1 |
| | B2 Conditionals | B2a IfElse | oneParamBoolean(boolean) | 5 | 3 |
| | | | twoParamsBoolean(boolean,boolean) | 17 | 16 |
| | | | oneParamInt(int) | 10 | 7 |
| | | | twoParamsInt(int,int) | 20 | 19 |
| | | | oneParamDouble(double) | 11 | 7 |
| | | | twoParamsDouble(double,double) | 20 | 19 |
| | | B2b Tern. | oneParamBoolean(boolean) | 2 | 2 |
| | | | twoParamsBoolean(boolean,boolean) | 2 | 3 |
| | | | oneParamInt(int) | 2 | 2 |
| | | | twoParamsInt(int,int) | 2 | 3 |
| | | B2c | simple(int) | 16 | 4 |
| | | | missingBreaks(int) | 14 | 4 |
| | | | withReturn(int) | 14 | 6 |
| | | B2d Linear | oneParamInt(int) | 5 | 3 |
| | | | oneParamIntNoSolution(int) | 5 | 3 |
| | | | twoParamsInt(int,int) | 7 | 4 |
| | | | twoParamsIntNoSolution(int,int) | 7 | 4 |

| Category | | | Method | NCSS | CCN |
|---|---|---|---|---|---|
| Basic (cont'd) | B2 Conditionals | B2d Linear | threeParamsInt(int,int,int) | 8 | 5 |
| | | | threeParamsIntNoSolution(int,int,int) | 8 | 5 |
| | | | oneParamFloat(float) | 5 | 3 |
| | | | twoParamsFloat(float,float) | 7 | 4 |
| | | | threeParamsFloat(float,float,float) | 8 | 8 |
| | | | oneParamDouble(double) | 5 | 3 |
| | | | twoParamsDouble(double,double) | 7 | 4 |
| | | | threeParamsDouble(double,double,double) | 8 | 8 |
| | | B2d Nonlinear | quadraticInt(int,int) | 7 | 5 |
| | | | quadraticIntNoSolution(int,int) | 7 | 5 |
| | | | quadraticFloat(float,float) | 7 | 7 |
| | | | quadraticFloatNoSolution(float,float) | 7 | 7 |
| | | | quadraticDouble(double,double) | 7 | 7 |
| | | | quadraticDoubleNoSolution(double,double) | 7 | 7 |
| | B3 Loops | B3a While | withLimit(int) | 7 | 3 |
| | | | withConditionAndLimit(int) | 8 | 4 |
| | | | withConditionNoLimit(int) | 8 | 3 |
| | | | withContinueBreak(int) | 12 | 4 |
| | | | complex(int,int,int,int) | 22 | 10 |
| | | | infinite(int) | 2 | 2 |
| | | | infiniteNotOptimalizable(int) | 8 | 5 |
| | | | nestedLoop(int,int) | 11 | 7 |
| | | | nestedLoopWithLabel(int,int) | 18 | 10 |
| | | B3b For | withLimit(int) | 5 | 3 |
| | | | withConditionAndLimit(int) | 6 | 4 |
| | | | withConditionNoLimit(int) | 6 | 3 |
| | | | withContinueBreak(int) | 9 | 4 |
| | | | complex(int,int,int,int) | 19 | 10 |
| | | | infinite(int) | 2 | 2 |
| | | | infiniteNotOptimalizable(int) | 8 | 5 |
| | | | nestedLoop(int,int) | 8 | 7 |
| | | | nestedLoopWithLabel(int,int) | 15 | 10 |
| | | B3c DoWhile | withLimit(int) | 7 | 3 |
| | | | withConditionAndLimit(int) | 8 | 4 |
| | | | withConditionNoLimit(int) | 8 | 3 |
| | | | withContinueBreak(int) | 12 | 4 |
| | | | complex(int,int,int,int) | 22 | 10 |
| | | | infinite(int) | 2 | 2 |
| | | | infiniteNotOptimalizable(int) | 8 | 5 |
| | | | nestedLoop(int,int) | 11 | 7 |
| | | | nestedLoopWithLabel(int,int) | 18 | 10 |
| | B4 Arrays | B4a SafeArrays | fromParams(int,int,int) | 6 | 5 |
| | | | indexParam(int) | 8 | 6 |
| | | | guessLength(int) | 6 | 3 |
| | | | fromParamsWithIndex(int,int,int,int) | 8 | 8 |
| | | | guessOneArray(int[]) | 7 | 7 |
| | | | guessOneArrayWithLength(int[]) | 7 | 7 |
| | | | twoArrays(int[],int[][]) | 10 | 10 |
| | | | iterateWithFor(numbers2[][]) | 11 | 7 |
| | | | iterateWithForeach(numbers[][]) | 13 | 7 |
| | | B4b UnsafeArrays | fromParams(int,int,int) | 6 | 5 |
| | | | indexParam(int) | 6 | 3 |
| | | | guessLength(int) | 6 | 3 |
| | | | fromParamsWithIndex(int,int,int,int) | 6 | 5 |
| | | | guessOneArray(int[]) | 5 | 4 |
| | | | guessOneArrayWithLength(int[]) | 5 | 5 |
| | | | twoArrays(int[],int[][]) | 8 | 5 |
| | | | iterateWithFor(numbers2[][]) | 8 | 5 |
| | | | iterateWithForeach(numbers[][]) | 10 | 5 |
| | B5 Functions | B5a Call | simple(int,int) | 2 | 1 |
| | | | useReturnValue(int,int) | 5 | 3 |
| | | | conditionalCall(int,int,boolean) | 5 | 3 |
| | | | simple(int,int) | 2 | 1 |
| | | | useReturnValue(int,int) | 5 | 3 |
| | | | conditionalCall(int,int,boolean) | 5 | 3 |
| | | B5b Rec. | simple(int) | 2 | 1 |
| | | | fibonacci(int) | 2 | 1 |
| | | | simple(int) | 5 | 3 |
| | | | fibonacci(int) | 8 | 5 |
| | B6 Exceptions | B6a Checked | always() | 2 | 2 |
| | | | conditionalAndLoop(int) | 9 | 6 |
| | | | call(int,int) | 2 | 1 |
| | | | recursive(int) | 2 | 1 |
| | | | tryCatch(int,int) | 4 | 4 |
| | | | tryCatchFinally(int,int,int) | 10 | 7 |

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

Table IV. Details of code snippets by categories (cont'd)

| Category | | | Method | NCSS | CCN |
|---|---|---|---|---|---|
| Basic (cont'd) | B6 Exceptions | B6b Unchecked | always() | 2 | 2 |
| | | | conditionalAndLoop(int) | 9 | 6 |
| | | | call(int,int) | 2 | 1 |
| | | | recursive(int) | 2 | 1 |
| | | | tryCatch(int,int) | 4 | 4 |
| | | | tryCatchFinally(int,int,int) | 10 | 7 |
| | | B6c CommonRuntime | arithmeticException(boolean) | 4 | 3 |
| | | | arrayIndexOutOfBoundsException(boolean) | 4 | 3 |
| | | | classCastException(boolean) | 4 | 3 |
| | | | illegalArgumentException(boolean) | 4 | 3 |
| | | | illegalStateException(boolean) | 4 | 3 |
| | | | indexOutOfBoundsException(boolean) | 4 | 3 |
| | | | nullPointerException(boolean) | 4 | 3 |
| | | | securityException(boolean) | 4 | 3 |
| | | | unsupportedOperationException(boolean) | 4 | 3 |
| Structures | S1 Basic | | useStructureParams(int,int) | 5 | 1 |
| | | | useStructure(CoordinateStructure) | 4 | 3 |
| | | | returnStructureParams(int,int) | 7 | 1 |
| | | | returnStructure(CoordinateStructure) | 6 | 3 |
| | S2 Cond. | | oneStructureParams(int,int) | 17 | 13 |
| | | | oneStructure(CoordinateStructure) | 16 | 15 |
| | | | twoStructuresParams(int,int,int,int) | 23 | 13 |
| | | | twoStructures(CoordinateStructure,CoordinateStructure) | 19 | 16 |
| | S3 With Loops | | withLimitParams(int,int,int) | 11 | 4 |
| | | | withLimit(CoordinateStructure,int) | 10 | 6 |
| | | | noLimitParams(int,int,int) | 11 | 3 |
| | | | noLimit(CoordinateStructure,int) | 10 | 5 |
| | | | arrayOfStructuresParams(int[],int[][]) | 15 | 8 |
| | | | arrayOfStructures(CoordinateStructure[]) | 10 | 5 |
| | S4 | | guessParams(int,int,int,int) | 16 | 10 |
| | | | guessCoordinates(CoordinateStructure,CoordinateStructure) | 16 | 13 |
| | | | guess(SegmentStructure) | 13 | 14 |
| Objects | O1 Simple | | oneOperationParams(int) | 4 | 1 |
| | | | oneOperationWithCheck(SimpleObject,int) | 5 | 3 |
| | | | oneOperationNoCheck(SimpleObject,int) | 3 | 1 |
| | | | twoOperationsParams(int,int) | 4 | 1 |
| | | | twoOperationsWithCheck(SimpleObject,int,int) | 5 | 3 |
| | | | twoOperationsWithNocheck(SimpleObject,int,int) | 3 | 1 |
| | | | guessResultParams(int,int,int) | 9 | 3 |
| | | | guessResult(SimpleObject,int,int,int) | 10 | 5 |
| | | | guessImpossibleResultParams(int,int,int) | 9 | 3 |
| | | | guessImpossibleResult(SimpleObject,int,int,int) | 10 | 5 |
| | | | guessOperationCountParams(int) | 8 | 4 |
| | | | guessOperationCount(SimpleObject,int) | 9 | 6 |
| | | | guessImpossibleOperationCountParams(int) | 8 | 4 |
| | | | guessImpossibleOperationCount(SimpleObject,int) | 9 | 6 |
| | | | guessResultAndOperationCountParams(int,int) | 8 | 5 |
| | | | guessResultAndOperationCount(SimpleObject,int,int) | 9 | 7 |
| | | | guessImpossibleResultAndOperationCountParams(int,int) | 8 | 5 |
| | | | guessImpossibleResultAndOperationCount(...) | 9 | 7 |
| | | | guessObject(SimpleObject) | 7 | 6 |
| | | | guessImpossibleObject(SimpleObject) | 7 | 6 |
| | | | fullCoverage(SimpleObject,int,int,int) | 8 | 6 |
| | O2 | | guessResultParams(int,int,int) | 9 | 3 |
| | | | guessResult(SimpleObjectDelegate,int,int,int) | 10 | 5 |
| | O3a | | guessResultParams(int,int,int) | 9 | 3 |
| | | | guessResult(SimpleExtendedObject,int,int,int) | 10 | 5 |
| | O3b If. | | guess(MyInterface,int) | 8 | 5 |
| | | | validate(MyInterface,int) | 8 | 6 |
| | | | guessImpossible(MyInterface,int) | 8 | 5 |
| | O3c | | guess(MyAbstract,int) | 8 | 5 |
| | | | validate(MyAbstract,int) | 8 | 6 |
| | | | guessImpossible(MyAbstract,int) | 8 | 5 |
| | O4 Over. | | guessResultParams(int,int,int) | 9 | 3 |
| | | | guessResult(SimpleObjectOverride,int,int,int) | 10 | 5 |
| | | | guessImpossibleParams(int,int,int) | 9 | 3 |
| | | | guessImpossible(SimpleObjectOverride,int,int,int) | 10 | 5 |

| Category | | Method | NCSS | CCN |
|---|---|---|---|---|
| Generics | G1 Func. | guessType(T) | 14 | 10 |
| | | guessTypeAndUse(T) | 22 | 14 |
| | | guessTypeWithExtends(T) | 14 | 10 |
| | | guessTypeWithExtendsAndUse(T) | 22 | 14 |
| | G2 Objects | guessInteger(Integer) | 5 | 3 |
| | | guessIntegerNoHelp(GenericTriplet) | 5 | 3 |
| | | guessImpossible(Double) | 5 | 3 |
| | | guessDescendant(IntegerTriplet) | 5 | 3 |
| | | guessSafe(SafeGenericTriplet) | 5 | 3 |
| | | guessSafeNoHelp(GenericTriplet) | 5 | 3 |
| Library | L1 Arithmetics | abs(int,int) | 5 | 4 |
| | | absImpossible(int,int) | 5 | 4 |
| | | minMax(int,int) | 5 | 4 |
| | | minMaxWithOrder(int,int) | 5 | 5 |
| | | minMaxImpossible(int,int) | 5 | 5 |
| | | sqrt(double) | 6 | 4 |
| | | sqrtImpossible(double) | 5 | 3 |
| | | cbrt(double) | 6 | 4 |
| | | powGuessBase(double) | 6 | 4 |
| | | powGuessExponent(double) | 6 | 4 |
| | | powGuessBaseAndExponent(double,double) | 6 | 4 |
| | | log10GuessArgument(double) | 6 | 4 |
| | | logGuessBase(double) | 6 | 4 |
| | | logGuessArgument(double) | 6 | 4 |
| | | logGuessBaseAndArgument(double,double) | 6 | 4 |
| | | sin(double) | 6 | 4 |
| | | sinImpossible(double) | 5 | 3 |
| | | cos(double) | 6 | 4 |
| | | cosImpossible(double) | 5 | 3 |
| | | tan(double) | 6 | 4 |
| | L2 Strings | equality(String) | 14 | 10 |
| | | equalityIgnoreCase(String) | 14 | 10 |
| | | add(String,String) | 11 | 8 |
| | | addWithCondition(String,String) | 20 | 16 |
| | | length(String) | 11 | 8 |
| | | charAt(String) | 11 | 8 |
| | | regionEquality(String) | 11 | 8 |
| | | compareTo(String) | 14 | 12 |
| | | compareToIgnoreCase(String) | 14 | 12 |
| | | startsEnds(String) | 11 | 7 |
| | | indexOf(String) | 11 | 10 |
| | | substring(String) | 11 | 9 |
| | | switchString(String) | 10 | 7 |
| | L3 | guessInteger(Integer) | 8 | 5 |
| | | integerOverflow(Integer,Integer) | 6 | 4 |
| | | guessDouble(Double) | 8 | 5 |
| | L4 Collections | guessSize(int) | 8 | 3 |
| | | guessElements(int,int) | 8 | 4 |
| | | guessSizeAndElements(int,int,int) | 8 | 5 |
| | | guessIndices(int,int) | 11 | 4 |
| | | guessElementAndIndex(int,int) | 11 | 3 |
| | | guessVectorWithSize(Vector) | 5 | 3 |
| | | guessGenericVectorWithSize(Integer) | 5 | 3 |
| | | guessGenericVectorWithElement(Integer) | 5 | 4 |
| | | guessListWithSize(List) | 5 | 3 |
| | | guessGenericListWithSize(Integer) | 5 | 3 |
| | | guessGenericListWithElement(Integer) | 5 | 4 |
| | LO Other | inheritsAPIGuessOnePrimitive(int) | 9 | 5 |
| | | inheritsAPIGuessOneObject(FingerNumber) | 11 | 7 |
| | | inheritsAPIGuessTwoPrimitives(int,int) | 11 | 5 |
| | | inheritsAPIGuessTwoObjects(...,...) | 11 | 8 |
| | | associatesAPIGuessValidDateFormat(String) | 6 | 3 |
| | | associatesAPIGuessDate(String) | 9 | 5 |
| | | guessValidUUID(String) | 7 | 5 |
| | | guessUUID(String) | 10 | 8 |
| | | regexCaseSensitive(String) | 7 | 5 |
| | | regexCaseInsensitive(String) | 9 | 5 |
| Others | AC | test(int) | 8 | 1 |
| | Enum | guessEnum(State) | 17 | 12 |
| | | guessEnumString(State) | 17 | 12 |
| | | guessEnumOrdinal(State) | 17 | 12 |
| | | switchEnum(State) | 14 | 12 |
| | 3rd p. | minMax(int,int) | 5 | 4 |
| | | minMaxWithOrder(int,int) | 5 | 5 |
| | | minMaxImpossible(int,int) | 5 | 5 |
| | Varargs | guess(int) | 7 | 7 |
| | | guessWithLength(int) | 7 | 7 |
| | | iterateWithFor(int) | 11 | 7 |
| | | iterateWithForeach(int) | 13 | 7 |

Copyright © 2015 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

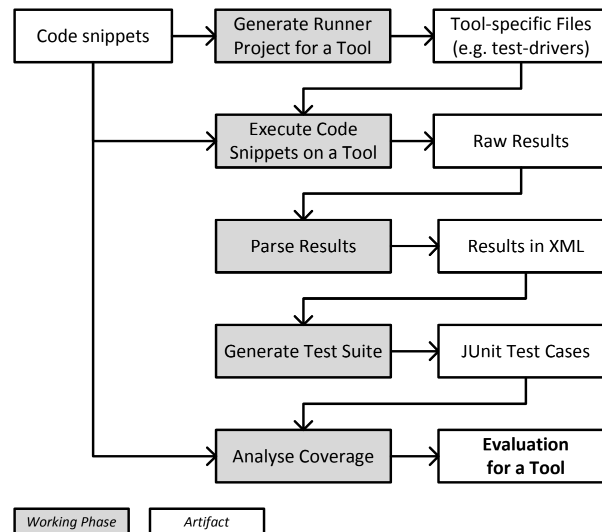



Figure 2. Workflow of the SETTE framework

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 we chose Java code as a format.

The execution of 300 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 [30] 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 that the output format of the tools (standard output, error output, XML) is often unspecified and certain cases rarely happen so we had to rely on our experimental data. After everything is parsed into a common format, test source code compilation requires attention and larger resources (we encountered out of memory compiler errors several times). In addition, uncommon phenomenon may still occur, such as a generated test case does not finish in a reasonable time. In one such case the code snippet was not an infinite loop and it happened only twice among ten full evaluations (evaluation 10 times for the 300 code snippets).

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

Copyright © 2015 John Wiley & Sons, Ltd.

*Prepared using* stvrauth.cls
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 | 0.2.0 | `-Dsearch_budget=TIMEOUT -class CODE-SNIPPET` |
| jPET | 0.4 | `-c bck 10 -td num -d -100000 100000 -l ff -v 2 -w -tr statements -cc yes` |
| Pex | 0.94.51006.1 | `pexwizard /NoMoles /TestFramework:nunit CODE-SNIPPET pex /TimeOut:TIMEOUT CODE-SNIPPET-TEST` |
| SPF | rev. 64083a81f440 (JPF rev. 36f3e39fcb4c) | Constraint solver: CORAL<br>Listener: `gov.nasa.jpf.symbc.SymbolicListener` |
| Randoop | 1.3.6 | `gentest --timelimit=TIMEOUT --forbid-null=false --null-ratio=0.5 --inputlimit=5000 --string-maxlen=50 --randomseed=...` |

## 5. EXPERIMENT PLANNING

To validate our approach we performed experiments 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 our 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 questions*: Is the approach able to produce fine-grained feedback on a tool's capabilities?

More specifically:

- *RQ1*: Which of the defined features are supported by each tool?
- *RQ2*: How do the generated test suites compare to each other with respect to typical metrics?

RQ1 is concerned with tool-specific information, while RQ2 is summarizing the insights obtained for the different tools. Note that our 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 our 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* [24] performs instrumentation and symbolic execution on Java bytecode.
- *EvoSuite* [25] uses genetic algorithms and mutation to evolve and reduce test suites.
- *jPET* [26] translates Java bytecode to *Prolog* and performs symbolic execution on that.
- *SPF* [29] does not translate nor instrument the bytecode, but uses a custom Java Virtual Machine, *Java PathFinder (JPF)* for execution.
- *Randoop* [28] uses feedback-directed random testing to generate tests.

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

- *Pex* [27] is a SE-based test input generator for .NET.

To support *Pex* all the 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).

General information about the subjects of the experiments were presented in Table I. The used tool versions and configurations are shown in Table V.

Copyright © 2015 John Wiley & Sons, Ltd.

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 Section 4, it was repeated for each of the tools.

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, each snippet was assigned exactly one flag from the followings.

- *N/A*: The tool was not able to perform test generation since the tool's input could not have been specified for the execution or the tool signaled that it cannot deal with the certain 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 [30] 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 [31, 32] suggest that high code coverage is not necessarily correlated with the effectiveness of the tests. Thus we included other metrics in our evaluation.

*5.3.2. Size* The size of the generated tests are also an important factor, because it increases the cost of manual oracle definition and test execution. There are several choices how to measure "size" (e.g. number of test cases or number of statements). We collect the *number of generated test cases* for each snippet, as for most of the snippets the generated tests contain only one statement. For the tools which does not generate any test cases but only test inputs, this number means the *number of generated test inputs* since the SETTE framework generates one test case with an assertion for each input.

*5.3.3. Duration* We collect the duration of each execution for each snippet. For the Java tools time is measured by SETTE from the start of the generation process until the process termination (using the `System.currentTimeMillis()` method). For Pex, the duration included in Pex's report is reused.

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

*5.3.4. Mutation score* Mutation analysis [33] can be used to assess the quality of a test suite by injecting faults in the unit under test, and measuring how many tests can differentiate the original version and the faulty versions (the mutants). We used an existing mutation framework with the following settings to compute mutation adequacy scores.

- *Mutation tool*: Version v1.1.5 of the *Major*** mutation framework was applied.
- *Number of mutants*: Starting from the snippets' source code 13697 mutants were generated using the default settings of *Major* with all its mutation operators. This fixed set of mutants were later used in the analysis of the tests 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 [31, 34].
- *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 *Pex* generate assertions mostly stating the return value of the snippet method, while *EvoSuite* uses an extra phase to search for detailed assertions. For the tools which does not generate any test cases or assertions, SETTE will add an assertion which checks whether the return value of the method call using the generated inputs is the same as the expected result which is calculated by SETTE based on the code snippets during runtime.

### *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.

1. For all the tools (except *Pex*) *virtual machines* were used running Ubuntu 14.04 64-bit and Oracle's Java 8 implementation. The virtual machines were given 2 GB memory and 1 processor core. The virtual machines were running in a shared environment, where the host machines had 32 GB memory and 2 quad core 2.5 GHz L5420 Xeon processors. The deployment of the virtual machines and the overall load of the hosts were unknown to us.
2. *Pex* was executed on a laptop with 12 GB memory and 1 dual core 1.7 GHz i3-4010U processor running Windows 8.1 64-bit and Microsoft .NET 4.0.

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

1. *Fixed time limit*: For the first set of experiments a fixed time limit, *30 seconds* was chosen.
2. *Variable time limit*: To account for the random nature of *EvoSuite* and *Randoop*, 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). *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 [28, 21] per class in their experiments, which is comparable

** *Major* mutation framework: http://mutation-testing.org/
††Note: *EvoSuite* performs minimization and assertion generation after finishing the search for tests, we did not count these steps in the specified time limit.

Copyright © 2015 John Wiley & Sons, Ltd.

Table VI. Number of repetitions with different results from the 10 runs with 30 s time limit

| Tool | Status | Size | Coverage |
|---|---|---|---|
| CATG, jPET, SPF | 0 | 0 | 0 |
| EvoSuite | 42 | 109 | 49 |
| Pex | 0 | 1 | 1 |
| Randoop | 0 | 16 | 0 |

with our setting (our 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).

For the variable time limit experiments we chose a subset of the snippets (129 from 300), which are harder to cover for the tools, namely the categories B2, B3, O1–O4, G1, G2, L1–L4 and LO. We run experiments with the following time limits: 15, 45, 60, 300 seconds.

In case of *CATG*, *jPET* and *SPF* when the generation reaches the time limit, SETTE terminates these processes and since none of the tools have a usable output, these cases are considered a timout. The two other Java tools, *EvoSuite* and *Randoop*, are able to watch the length of the generation process and they finish the test generation within the time limit. It must be noted that *EvoSuite* may significantly use more time for one execution than the time limit, because the tool also re-executes the test cases and performs test minimization which is not carried out by the other tools. However, the generation always finishes near the time limit. Pex is also able to take care of finishing within the specified time.

*5.4.3. Repetitions* As some of the tools utilize random algorithms, it was necessary to repeat the experiments several times [35]. The executions were performed *10 times* for each tool as in previous experiments from the tool's developers [28, 21]. 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* there were several snippets, where some of the executions were able to achieve the maximal coverage (C), while for not (NC). In this cases, we assigned a C status if more than 5 from the 10 repetitions resulted in C (a similar method was used in [45]). For size and coverage we calculated aggregated values (mean, median, standard deviance. . . ).

This resulted in 18 000 observations for the first experiment (6 tools × 300 snippets × 10 repetitions), and 10320 observations from the second experiment (2 tools × 129 snippets × 10 repetitions × 4 time limit).

### *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 *Pex* the status was partly automatically determined with a Powershell script and partly checked and categorized independently by the two authors. The main reason for this is that Pex includes the coverage of those methods which were not meant to be covered and the coverage for the same .NET code with the same inputs may be different from the coverage of the Java code.

*Knowledge of the tools:* These tools are fairly complex and configurable software (e.g. *EvoSuite* has 29 options and 296 parameters, *Pex* has 118 command-line parameters and many other options), 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, our 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 we selected Java-based tools because Java was the platform for which the most tools are available. Later we extended our selection to a tool with different platform (*Pex*). Note that our findings are about the tools and not

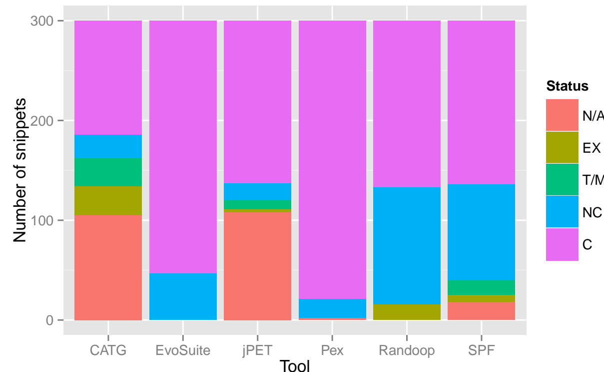
Copyright © 2015 John Wiley & Sons, Ltd.

*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

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

| Status | CATG | EvoSuite | jPET | Pex | Randoop | SPF |
|---|---|---|---|---|---|---|
| N/A | 105 | – | 108 | 2 | – | 18 |
| EX | 29 | – | 3 | – | 16 | 7 |
| T/M | 28 | – | 9 | – | – | 15 |
| NC | 24 | 52 | 17 | 19 | 117 | 96 |
| C | 114 | 248 | 163 | 279 | 167 | 164 |

Figure 3. Number of snippets categorized into the different statuses by tools




about the techniques they use (as shown by the great differences in the results for all symbolic-execution based tools).

## 6. RESULTS AND DISCUSSION

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: we uploaded for each tool the tool's configuration, the tool's full output, the generated test inputs and codes and coverage colored snippet codes to help to validate our results.

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

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

Figure 3 and Table VII presents a high-level overview of the results, the number of snippets categorized into the different statuses for each tool.

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 percentage 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. 4 presents a visualization of the results that highlights the data with colors.

**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

[‡‡]R statistical framework: https://www.r-project.org/

Copyright © 2015 John Wiley & Sons, Ltd.


DOI: 10.1002/stvr

Table VIII. Detailed status by categories for all the 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 | 14 | 6 | 5 | 9 | 2 | 2 | 2 | | 1 | 3 | 1 | 1 | 2 | | | 2 | | 2 | 3 | | 1 |
| | NC | | 1 | | 2 | | | | | 1 | | | | | | | | 3 | 10 | | 1 | 6 | |
| | T/M | | | 21 | | 1 | | | | 1 | | 5 | | | | | | | | | | | |
| | EX | | 4 | | 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 | 21 | 24 | 13 | 9 | 21 | 4 | 4 | 6 | | 11 | 2 | 7 | 2 | 4 | 5 | 17 | 9 | 3 | 7 | 6 | 11 |
| | NC | | 10 | 3 | 5 | 1 | | | | | 3 | 10 | | 1 | 2 | | 1 | 3 | 4 | | 4 | 4 | 1 |
| | T/M | | | | | | | | | | | | | | | | | | | | | | |
| | EX | | | | | | | | | | | | | | | | | | | | | | |
| | N/A | | | | | | | | | | | | | | | | | | | | | | |
| **jPET** | C | 42 | 17 | 24 | 14 | 10 | 9 | 4 | 4 | 2 | 3 | 19 | 2 | 1 | 2 | 2 | 2 | | | | | 1 | 5 |
| | NC | | | 3 | 4 | | 1 | | | | | | | 3 | 2 | | 1 | 2 | | | | | 1 |
| | T/M | | 3 | | | | | | | 4 | | 2 | | | | | | | | | | | |
| | EX | | | | | | | | | | | | | | | | | | | | | | 3 |
| | N/A | 20 | 11 | | | | 11 | | | | | | | 4 | | 2 | 3 | 18 | 13 | 3 | 11 | 9 | 3 |
| **Pex** | C | 62 | 31 | 27 | 18 | 10 | 21 | 4 | 4 | 6 | 3 | 20 | 2 | 8 | 4 | | 4 | 18 | 11 | 3 | 8 | 4 | 11 |
| | NC | | | | | | | | | | | 1 | | | | 2 | 2 | 2 | 2 | | 3 | 6 | 1 |
| | T/M | | | | | | | | | | | | | | | | | | | | | | |
| | EX | | | | | | | | | | | | | | | | | | | | | | |
| | N/A | | | | | | | | | | | | | | | 2 | | | | | | | |
| **Randoop** | C | 62 | 14 | 24 | 2 | 10 | 21 | 2 | 2 | 2 | | 6 | 1 | 1 | 1 | 4 | | 11 | | 1 | 1 | | 2 |
| | NC | | 17 | 3 | 6 | | | 2 | 2 | 2 | 3 | 15 | 1 | 7 | 3 | | 6 | 9 | 13 | 2 | 10 | 10 | 6 |
| | T/M | | | 9 | | 2 | | | | | | 4 | | | | | | | | | | | |
| | EX | | | | 10 | | | | | 2 | | | | | | | | | | | | | 4 |
| | N/A | | | | | | | | | | | | | | | | | | | | | | |
| **SPF** | C | 60 | 26 | 9 | 4 | 6 | 19 | 2 | 2 | | 1 | 4 | 1 | 1 | 2 | | | 18 | | | 3 | 2 | 4 |
| | NC | | 4 | | 14 | | | 2 | 2 | 4 | 2 | 13 | 1 | 7 | 2 | 4 | 6 | 1 | 13 | 3 | 5 | 5 | 8 |
| | T/M | | | | | | | | | | | | | | | | | | | | | | |
| | EX | | 1 | | | | | | | | | | | | | | | | | | 3 | 3 | |
| | N/A | 2 | | 9 | | 2 | 2 | | | 2 | | | | | | | | 1 | | | | | |

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.

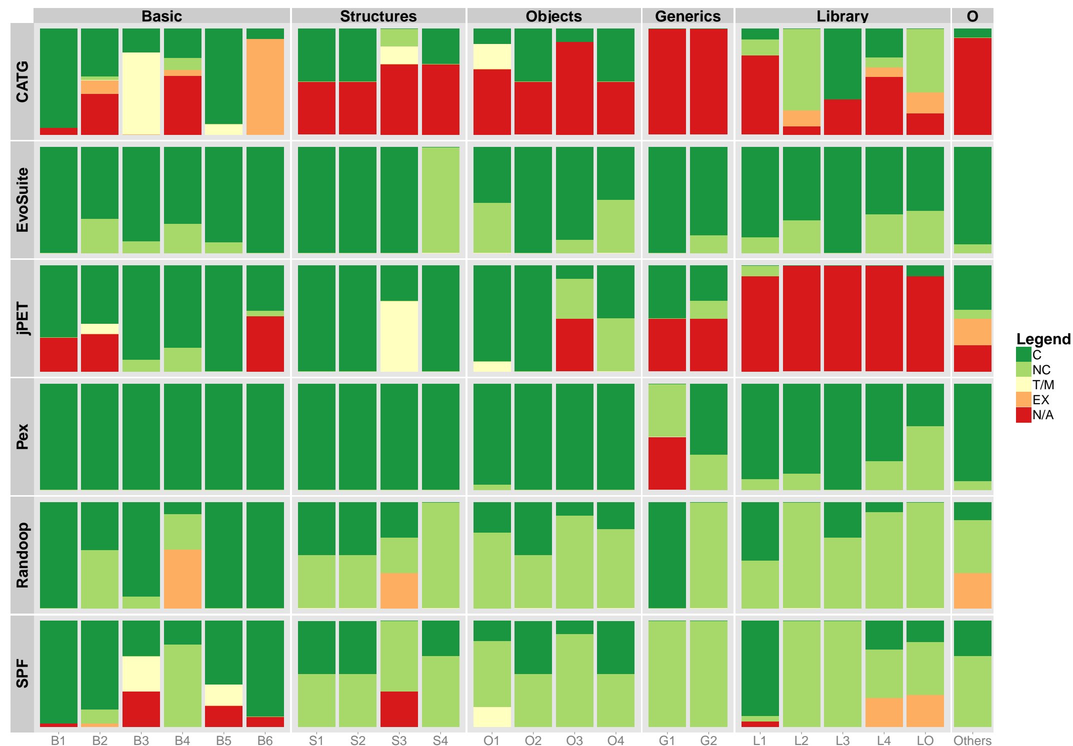
In case of structures and objects, *CATG* is able to handle the fields but cannot generate objects as input. However, when more complex constraint solving is needed, then in most of the cases *CATG* exceeds the time limit. Generics and the majority of the arithmetic functions are not supported by the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

**jPET** 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 the best 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 the best 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.

Copyright © 2015 John Wiley & Sons, Ltd.

Figure 4. Visualization of the results




**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.

**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 and it is the only tool who is completely able to cover all the snippets for objects and generics. The not covered library cases focused on special features, such as not common string methods, date and UUID guessing. However, *EvoSuite*'s limit is solving complex constraints and mathematical problems and covering codes with looping statements.

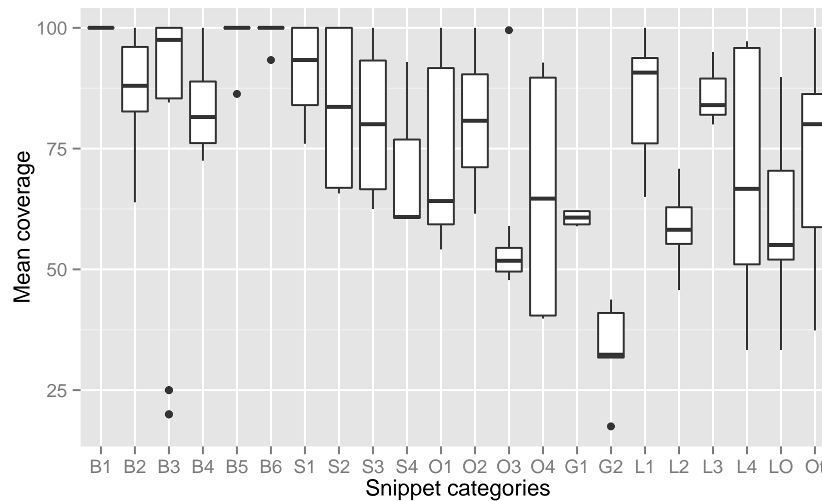
**Pex** handled all the instructions, the tool detects exceptions and shuts itself down after the time limit has expired. Thus, no generations were classified as *EX* and *T/M*. *Pex* was able to satisfy statement coverage requirements in most of the cases, however, we also found some cases when it failed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

Two executions were marked as *N/A* because *Pex* was unable to guess a valid generic type when there is a condition for the base class. Two *NC* snippets focused on `float` precision (the tool was able to handle all the snippets using `double`), 1 on object guessing (see later), 4 on generics, 15 on built-in library features and 1 on enumerations.

Copyright © 2015 John Wiley & Sons, Ltd.

*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

Figure 5. Mean coverage along snippets aggregated by categories (all tools)




**Summary:** RQ1 focused on tool-specific 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.

### *6.2. RQ2: Comparing the Generated Tests*

*6.2.1. Coverage* When analyzing coverage 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% coverage (as no coverage 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 with high coverage. On the other hand, if they count as 0% coverage, 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.

Figure 5 presents the data aggregated by categories. In this case we removed the executions with N/A, EX and T/M status. Then, for each snippet and tool, the mean of the achieved coverage was computed (thus averaging the 10 runs). The figure shows that there are certain categories that are easy to handle for most tools (e.g. B1 types and operators, B5 function or B6 exceptions), while for others there are great differences along tools (e.g. O4 overriding). Snippets in the generics category were especially hard.

Table IX and Figure 6 report the coverage achieved by each of the tools for all snippets. In this case we counted the missing coverage values as 0%. The values are similar to the status overview, *EvoSuite* and *Pex* were able to handle relatively well the snippets, *Randoop* had more diverse values, while *CATG*, *jPET* and *SPF* had varying results (high standard deviation). The *Manual* row represents the sample inputs selected by us that achieve the maximal coverage.

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

Figure 7 presents the distribution of the mean duration values computed for each snippet and tool. Note that the figure was limited to the 0–60 range, which is twice of the time limit of an execution. *EvoSuite* was the only tool that exceeded this limit significantly, but it had snippets where

Copyright © 2015 John Wiley & Sons, Ltd.


DOI: 10.1002/stvr

Figure 6. Distribution of mean coverage along snippets for each tool

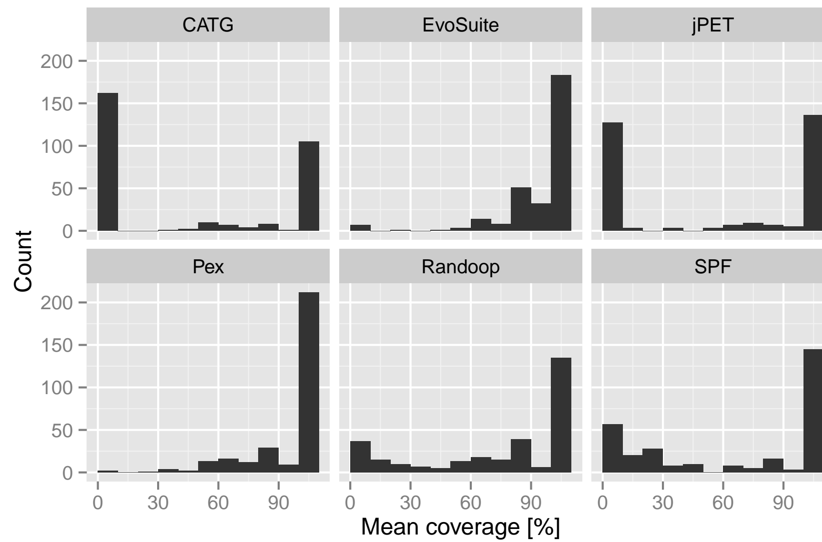



Table IX. Coverage [%] achieved by the tools (aggregates of snippet's mean coverage values)

| Tool | Min | Mean | Median | Max | SD |
|---|---|---|---|---|---|
| CATG | 0.00 | 42.27 | 0.00 | 100.00 | 47.19 |
| EvoSuite | 0.00 | 91.14 | 100.00 | 100.00 | 17.99 |
| jPET | 0.00 | 53.60 | 76.50 | 100.00 | 47.79 |
| Pex | 0.00 | 91.08 | 100.00 | 100.00 | 17.45 |
| Randoop | 0.00 | 70.83 | 85.71 | 100.00 | 36.34 |
| SPF | 0.00 | 62.33 | 88.20 | 100.00 | 42.19 |
| Manual | 0.00 | 92.67 | 100.00 | 100.00 | 18.01 |

the whole generation process (including search, test minimization, test execution and assertions) lasted 417 seconds.

*Pex* and *jPET* were the two fastest tools (note that *jPET* had a large number of removed snippets). For *Pex* even the median of the duration is 0.08 seconds, which shows that if dynamic symbolic execution can find a solution, then it can find it fast. Moreover, the overall low values were due to *Pex* has numerous built-in default boundaries and timeouts, e.g. the constraint solver calls time outs after 1 second. *Randoop* almost always used all of its allocated time for searching for test inputs.

*6.2.3. Size* For analyzing the size variable, the N/A, EX and T/M experiments were removed.

Table X summarizes the statistics of the generated test suite sizes. Notice the large maximal values in the case of *jPET* and *Randoop*. *jPET* generated 1026 and 2047 test inputs for 2 snippets in S3 (array of structures), but it had 13 other snippets with size values 20–200. The 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). The minimization feature of *EvoSuite* was working well, it achieved similar sizes than *Pex*.

Figure 8 compares the tool's test suites to the manually selected sample inputs. The figure's data contains for every tool, for every snippet the sample input size minus the tool's generated size. Thus a negative value means that the tool generated fewer tests (cases with NC status). *EvoSuite* and *Pex* were relatively close to the sample inputs, while for *jPET* and *Randoop* there was a larger difference.

*6.2.4. Mutation analysis* The mutation analysis has been carried out for *CATG*, *EvoSuite* and *SPF* using the *Major* framework. We were not able to perform mutation analysis for *jPET* and *Randoop*.

Copyright © 2015 John Wiley & Sons, Ltd.

*Prepared using* stvrauth.cls
DOI: 10.1002/stvr

Figure 7. Distribution of mean duration along snippets for each tool

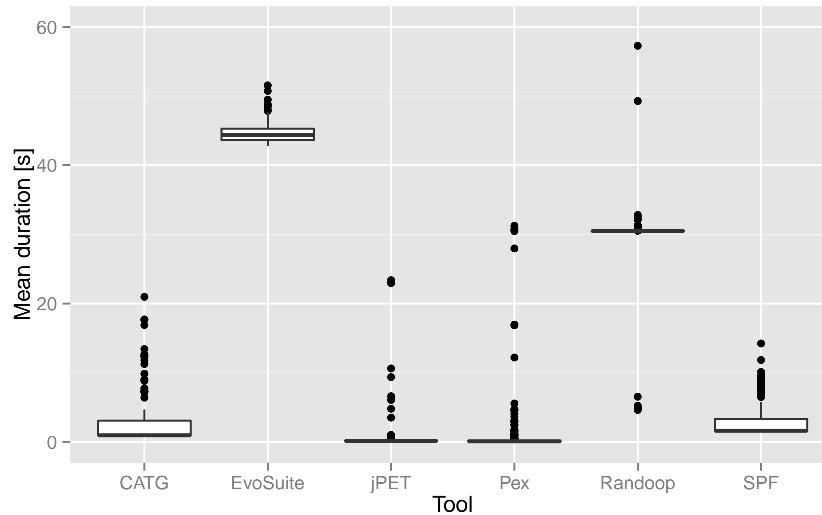
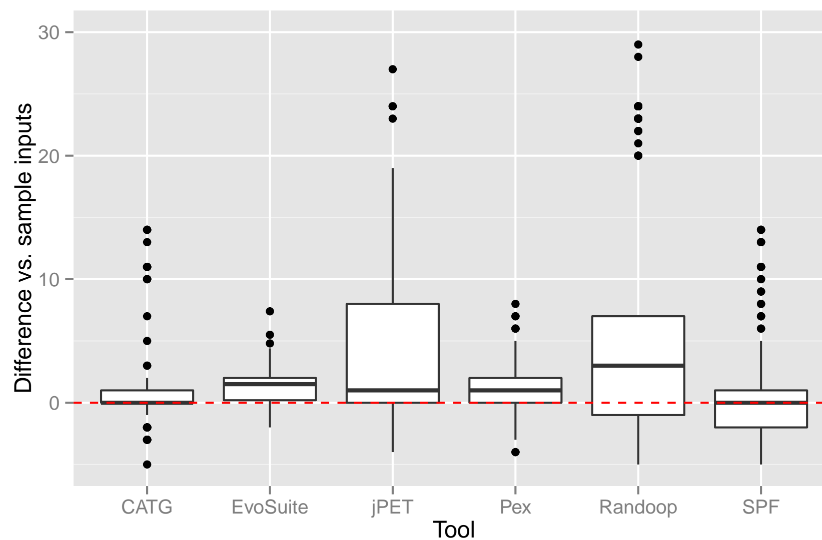



Table X. Size of the generated tests by the tools (aggregates per snippet's mean size values)

| **Tool** | **Min** | **Mean** | **Median** | **Max** | **SD** | **Sum** |
|---|---|---|---|---|---|---|
| CATG | 1.0 | 3.04 | 1.0 | 16.0 | 3.65 | 420.0 |
| EvoSuite | 0.1 | 3.74 | 3.1 | 9.5 | 1.8 | 1123.3 |
| jPET | 1.0 | 26.37 | 4.0 | 2047.0 | 170.77 | 4747.0 |
| Pex | 1.0 | 3.52 | 3.0 | 9.0 | 2.20 | 1050.3 |
| Randoop | 0.0 | 535.27 | 8.0 | 4999.0 | 1393.37 | 152018.7 |
| SPF | 1.0 | 4.10 | 1.0 | 103.0 | 11.24 | 1068.0 |
| Manual | 1.0 | 2.29 | 2.0 | 8.0 | 1.38 | 687 |

Figure 8. Difference between the sample input size and the test size generated by the tools




Unfortunately *jPET* generates numerous test inputs which cannot be translated to compilable Java code. *Randoop* generates more than 150 000 test cases for the 300 snippets which have to be run on

Copyright © 2015 John Wiley & Sons, Ltd.


DOI: 10.1002/stvr

Table XI. Mutation analysis

| Tool | Covered mutants | Killed mutants | Mutation score |
|---|---|---|---|
| CATG | 2028 (18.8%) | 1284 (9.37%) | 0.2946 |
| EvoSuite | 6016 (43.92%) | 3682 (26.88%) | 0.8449 |
| SPF | 3389 (24.74%) | 2235 (16.32%) | 0.5128 |

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

| Tool | Metric | 15s | 30s | 45s | 60s | 300s |
|---|---|---|---|---|---|---|
| EvoSuite | Coverage [%] | 85.11 | 85.94 | 86.14 | 86.45 | 87.72 |
| | Size | 3.41 | 3.51 | 3.53 | 3.56 | 3.68 |
| Randoop | Coverage | 57.84 | 58.18 | 58.18 | 58.18 | 58.18 |
| | Size | 143.62 | 146.60 | 146.7 | 147.77 | 155.95 |

all the mutants. In addition, *Major* might generate such mutants which will result in infinite loops for the generated inputs. (During our measurements, the mutation analysis has been running for 5 hours without writing any more lines to the output but using 100% of the CPU.)

The analysis was performed only once on one set of test suites (from the first experiment with 30 second timeout). From the 13 697 mutants 9 339 were not covered by any of the three tools (we consider these as equivalent mutants). The results can be see in Table XI. The mutation score has been calculated using the following formula:

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

The results show that the tests generated by EvoSuite not only achieve higher coverage, they are also able to detect a much larger number of mutants. 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.

*6.2.5. Experiments with variable time limit* In the variable time limit experiments the change of the mean coverage and size was analyzed with increasing time limits. Table XII summarizes the results. Both *EvoSuite* and *Randoop* were able to increase the achieved coverage and generate more tests. However, the increase was under 3% compared to the 30 s baseline, even when the time limit was increased to five times. 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.

## 7. RELATED WORK

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

### 7.1. 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 [20] summarized actual research directions, Cadar *et al.* [19] collected experiences from tool developers and Chen *et al.* [18] listed current challenges. Regarding search-based software testing McMinn [4] surveyed SBST approaches focusing on the different algorithms used, Ali *et al.* [36] investigated the empirical assessment of SBST papers, and Harman *et al.* [23] presented the trends in SBST research and

Copyright © 2015 John Wiley & Sons, Ltd.

DOI: 10.1002/stvr

open problems regarding testing non-functional properties. Arcuri *et al.* [5] analyze random testing and present its theoretical and real-world results.

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

### 7.2. 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) [37] 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* [38], a representative selection of 100 open source projects from SourceForge. The Defects4J [39] dataset consists of 357 real faults from five open source projects along with developer-written tests. The SBST Java Unit Testing Tool Contest [40, 41] 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 also be created automatically, e.g. RUGRAT is a flexible tool for generating Java programs that can serve as benchmarks for program analysis and testing tools [42].

### 7.3. Experiments

Several real-life *experiments* were performed to evaluate test generators. Lakhotia *et al.* [6] investigated the coverage of CUTE and AUSTIN (a search-based tool) on five real-life open source components. Braione *et al.* [43] performed an experiment on an industrial control software using CREST, Pex and AUSTIN. Qu and Robinson [7] measured the coverage of CREST and KLEE on a 3.9M LOC realtime embedded system. Wang *et al.* return value compared automatically generated tests by the KLEE tool with manual tests on 40 programs from the CoreUtils package. Fraser and Arcuri [21] perfomed a large-scale evaluation of EvoSuite on the extended SF110 benchmark. Gay *et al.* [44] compared test inputs generated to achieve different coverage criteria with randomly generated ones on 7 industrial systems. Shamshiri *et al.* [45] performed experiments with Randoop, EvoSuite and Agitar on real faults from the Defects4J dataset.

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 [8] 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 simple types, structures). Their benchmark suite was smaller and its code is not available. However, they also provided valuable feedback, and evaluated several tools which were not covered in our work.

### 7.4. Related problems

A related problem is comparing *static analysis tools*. The Juliet test suite [46] 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.

Another related problem is testing and comparing code *compilers*, although in that case research focused on generating test programs from syntactic and semantic definition rules [47].

## 8. 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 we identified a set of

Copyright © 2015 John Wiley & Sons, Ltd.
*Prepared using* stvrauth.cls

DOI: 10.1002/stvr

features that these tools should cover, and designed 300 code snippets representing these features. Initially we created these snippets for the Java platform, but later they were easily translated to .NET. We implemented a framework called SETTE that can automatically perform experiments and evaluations on test generators using these snippets. We performed several experiments on six different tools including ones based on symbolic execution, search-based and random testing. The results show that the evaluation can identify both strengths and weaknesses in the tools. 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. Currently interaction with the environment is not covered in our snippets, this would be an important future extension.

We made all source code and experimental results available online. Both new tools or code snippets can be easily added to extend our work. 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.

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 M, 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. 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.
7. 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.
8. Galler SJ, Aichernig BK. Survey on test data generation tools. *STTT* 2014; **16**(6):727–751, doi:10.1007/s10009-013-0272-3.
9. Cseppentő L, Micskei Z. Evaluating Symbolic Execution-based Test Tools. *Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on*, IEEE, 2015; 1–10, doi:10.1109/ICST.2015.7102587.
10. Myers GJ. *Art of Software Testing*. John Wiley & Sons, Inc.: New York, NY, USA, 1979.
11. Miller EF, Melton RA. Automated generation of testcase datasets. *SIGPLAN Not.* 1975; **10**(6):51–58, doi:10.1145/390016.808424.
12. Howden W. 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.
13. DeMillo R, Offutt A. Constraint-based automatic test data generation. *Software Engineering, IEEE Transactions on* 1991; **17**(9):900–910, doi:10.1109/32.92910.
14. Boyapati C, Khurshid S, Marinov D. Korat: Automated testing based on Java predicates. *International Symposium on Software Testing and Analysis*, ISSTA '02, ACM, 2002; 123–133, doi:10.1145/566172.566191.
15. King JC. Symbolic execution and program testing. *Commun. ACM* 1976; **19**(7):385–394, doi:10.1145/360248.360252.
16. 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.
17. 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 Berlin Heidelberg, 2012; 323–339, doi:10.1007/978-3-642-29032-9_16.
18. 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.
19. 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.
20. 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.
21. 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.

Copyright © 2015 John Wiley & Sons, Ltd.

22. 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.
23. 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 International Conference on*, 2015; 1–12, doi:10.1109/ICST.2015.7102580.
24. Sen K. CATG web page. https://github.com/ksen007/janala2 2013. Last accessed on 24/10/2014.
25. Fraser G, Arcuri A. Whole test suite generation. *IEEE Transactions on Software Engineering* 2013; **39**(2):276 –291, doi:10.1109/TSE.2012.14.
26. 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.
27. 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.
28. Pacheco C, Lahiri S, Ernst M, Ball T. Feedback-directed random test generation. *Int. Conf. on Software Engineering*, ICSE'07, 2007; 75–84, doi:10.1109/ICSE.2007.37.
29. 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.
30. Hoffmann MR. JaCoCo Java code coverage library. http://www.eclemma.org/jacoco/ 2014. Last accessed on 24/10/2014.
31. 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.
32. 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* 2014; To appear.
33. 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.
34. Andrews J, Briand L, Labiche Y, Namin A. 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.
35. 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.
36. Ali S, Briand L, Hemmati H, Panesar-Walawege R. 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.
37. 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.
38. 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.
39. Just R, Jalali D, Ernst MD. Defects4J: A database of existing faults to enable controlled testing studies for Java programs. *International Symposium on Software Testing and Analysis*, ISSTA, 2014; 437–440, doi:10.1145/2610384.2628055.
40. Bauersfeld S, Vos TE, 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.
41. Rueda U, Vos TE, Prasetya I. Unit testing tool competition – round three. *Search-Based Software Testing (SBST), 2015 IEEE/ACM 8th International Workshop on*, 2015; 19–24, doi:10.1109/SBST.2015.12.
42. Hussain I, Csallner C, Grechanik M, Xie Q, Park S, Taneja K, Hossain BM. RUGRAT: Evaluating program analysis and testing tools and compilers with large generated random benchmark applications. *Software—Practice & Experience* 2014; To appear.
43. 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.
44. Gay G, Staats M, Whalen M, Heimdahl M. 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.
45. 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)*, 2015. To appear.
46. Boland T, Black P. Juliet 1.1 C/C++ and Java test suite. *Computer* Oct 2012; **45**(10):88–90, doi:10.1109/MC.2012.345.
47. 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.

Copyright © 2015 John Wiley & Sons, Ltd.