Evaluating Symbolic Execution-based Test Tools

Lajos Cseppentő and Zoltán Micskei
Budapest University of Technology and Economics
Email: lajos.cseppento@inf.mit.bme.hu, micskeiz@mit.bme.hu

***Abstract*—In recent years several symbolic execution-based 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 no 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 four Java and one .NET test generator tools. The results highlight the strengths and weaknesses of each tool, 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.**

I. 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 generate test cases from behavioral models. *Code-based methods* start from the source code of the system under test and select test inputs that maximize the achieved code coverage. Code-based methods primarily generate only *test inputs* without expected outputs and rely on assertions, exceptions to detect issues.

One of the active research topics in test input generation is the use of *symbolic execution* (SE) [2]. 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]. Tools have been developed 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 SE-based test generation is undecidable in general, the development of such tools faces several theoretical challenges. Additionally, as modern programming languages and platforms offer a wide variety of complex language constructs and features, supporting all of them is a significant development task. Thus the available test generators have varying capabilities.

Typically the publication of a new tool includes also experimental results to assess the tool's capabilities. However, some of the developers use their own sample programs, while others conduct case studies on open source software. The used third-party software are usually different, thus *comparing* the abilities of the tools is not trivial. The tool papers always describe the benefits of the new tool and the innovation carried out during development. Nonetheless, only some of them mention all limits of the tool. Moreover, several reviews and surveys have been published in which tools were applied to complex software [4], [5]. These surveys 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 [6] 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 code snippets that target these features and serve as inputs for the test input generators. Next, the tools were executed with 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. Using all the snippets, detailed feedback could be obtained, which makes possible to compare the tools.

Thus *contributions* of this paper include (i) collecting the relevant and challenging features of imperative programming languages w.r.t. test input generation, (ii) mapping these features to the Java and .NET platforms and implementing them in 300 code snippets, (iii) creating the Symbolic Execution-based Test Tool Evaluator (SETTE) framework for automatically evaluating the snippets, and (iv) experimental results of comparing four Java and one .NET tools, which highlight the significant differences and the 'hard code parts', i.e. the features which are difficult to tackle for most of the tools.

978-1-4799-7125-1/15/$31.00 ©2015 IEEE. Personal use of this material is permitted. Permission from IEEE must be obtained for all other uses.
In Proc. of IEEE Int. Conf. on Software Testing, Verification and Validation (ICST) 2015. DOI: 10.1109/ICST.2015.7102587
An extended version of this paper appeared in: "Evaluating code-based test input generator tools", STVR. 2017. DOI: 10.1002/stvr.1627

TABLE I
List of Test Input Generator Tools

Name	Platform	Evaluated	Access	Published	Updated	Input	Output
CATG	Java	●	open source	2012	2014	source code	input values
CAUT	C		closed source	2010	2014	source code	input values
CodePro AnalytiX	Java		closed source	2001	2010	source code	test code
EvoSuite	Java	●	closed source	2008	2014	bytecode	test code
CREST	C		open source	2008	2014	source code	input values
Jalangi	JavaScript		open source	2013	2014	source code	input values
KLEE	C		open source	2008	2014	LLVM bytecode	input values
LCT	Java		open source	2010	2012	bytecode	input values
Palus	Java		open source	2010	2012	bytecode	test code
PET/jPET	Java	●	open source	2009	2011	source code	input values
Pex	.NET	●	closed source	2008	2010	IL code	test code
SPF	Java	●	open source	2012	2014	bytecode	input values

II. Overview

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) [7]. 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. Thus, theoretically all the reachable paths and the inputs activating them can be discovered. The goal of test input generation using symbolic execution is to produce a set of test inputs, which achieve maximal possible code coverage (statement, branch, etc.).

However, like other hard problems, the practical application of symbolic execution faces also several *challenges* [2], [8], [9], [10]. 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.

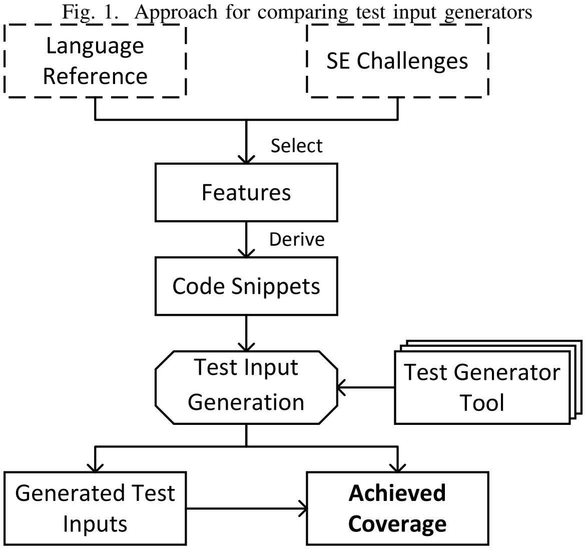
Tools In the last decade several SE-based test input generator tools have been published, Table I lists some of them[1]. Some tools are open source, some tools are still actively developed while others are not available any more. The tools also vary in their input language and platform (C, Java and .NET). In our experiments we concentrated on tools for Java (see later in Section V), we evaluated the following tools (marked in the evaluated column): *CATG* [11], *PET* [12], and *Symbolic PathFinder (SPF)* [13] and included additional results for *Pex* [14] and *EvoSuite* [15].

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

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 and code coverage achieved by these inputs on the code snippet 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 III), the code snippets (Section IV), the experiments executing the snippets (Section V) and the discussion of the results (Section VI).

Fig. 1. Approach for comparing test input generators



[1] For a more complete list of code-based test input generator tools see e.g. `http://mit.bme.hu/~micskeiz/pages/cbtg.html`

III. 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.
- *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.

Primitive Types and Operators (B)

As our former experiences showed, it is not obvious that a SE 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[2] are checked with safe and unsafe snippets: safe snippets handle illegal indices and null references, while unsafe snippets do not. The ability of the tool to detect common exceptions is also checked in this category. The main questions were whether a tool is able to

B1 handle all the basic language elements,
B2 solve simple and complex arithmetic problems,
B3 generate inputs for simple loops, loops with inner state, 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.

TABLE II
Features of Comparison

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	

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.

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

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

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.

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.

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[3] 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.

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.

Evaluating the Selection of Features

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

The features cover the following challenges from Section II:

- *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)
- *Pointer operations:* pointers can be evaluated in B1, S and O (however as our targets were managed languages we have not covered them)
- *Interaction with the environment:* was not covered
- *Multi-threading:* was not covered

We aimed for a set of features and code snippets that is able to check SE 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 easily added to extend the scope of our work.

IV. Implementation

We mapped the features defined in Section III to the Java language, and created 300 code snippets implementing them. The numbers of snippets in each category are shown on Table III. 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. As it can be seen, the majority of the code snippets focuses on the basic features. The reason for this is because the majority of the basic features should be individually checked, including all the types and operators.

The execution of 300 snippets for several different tools by hand is extremely expensive and error-prone. First, the

[3] `http://docs.oracle.com/javase/7/docs/api/`

TABLE III
NUMBER OF CODE SNIPPETS BY CATEGORIES

Category	# of code snippets	Total
Basic	62+31+27+18+10+21	169
Structures	4+4+6+3	17
Objects	21+2+8+4	35
Generics	4+6	10
Library	20+13+3+11+10	57
Others	1+4+3+4	12

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 (*Symbolic Execution-based Test Tool Evaluator*) 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 [16] code coverage library. With SETTE not only SE tools can be evaluated but other test generators too.

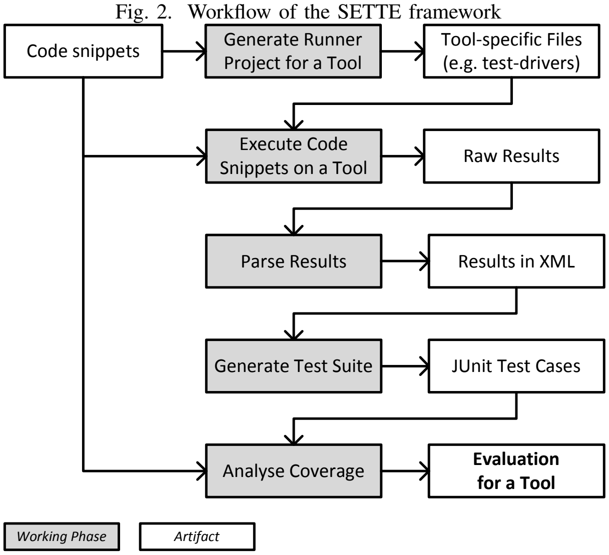
Listing 1. Sample code snippet

```
public final class BasicControlFlow {
  public static int ifWithElse(int x) {
    if (x > 0) {
      return 1;
    } else if (x < 0) {
      return -1;
    } else {
      return 0;
    }
  }
}
```

A sample code snippet can be seen in Listing 1. 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. The usage of annotations and modifiers are strict and validation is performed by SETTE. In addition, sample inputs (i.e. inputs for a snippet reaching the desired coverage for demonstration and validation purposes) and included method coverage (i.e. methods whose coverages should be also taken in account during evaluation) can be defined.

The SETTE framework and the code snippets are publicly available from the tool's website[4]. The SETTE workflow is shown in Fig. 2.

Fig. 2. Workflow of the SETTE framework



V. EXPERIMENTS AND RESULTS

To validate our approach we performed experiments on several test input generator tools.

Research Questions

The main *objective* of our research was to create a method and framework for comparing and evaluating test input generators. The experiments were designed to answer the following *research questions*:

RQ1 Is the approach able to produce fine-grained feedback on a tool's capabilities?

RQ2 Which are the 'hard code parts', i.e. the cases which were solved by only one or none of the tools correctly?

Method

The tools under study were executed to perform test generation for each of the code snippets separately. The detailed results (generated test input, log files, achieved code coverage and possible errors raised) were collected, and 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.

[4] SETTE website: `http://sette-testing.github.io`

TABLE IV
CONFIGURATIONS OF TEST INPUT GENERATORS USED IN THE EXPERIMENTS

Tool	Version	Configuration
CATG	v1.03 (Yices 2.2.2 64-bit)	`./concolic 100 CODE-SNIPPET`
jPET	0.4	`-c bck 10 -td num -d -100000 100000 -l ff -v 2 -w -tr statements -cc yes`
SPF	rev. 64083a81f440 (JPF rev. 36f3e39fcb4c)	Constraint solver: CORAL Listener: `gov.nasa.jpf.symbc.SymbolicListener`
EvoSuite	`evosuite-20141014.jar`	`-generateTests -Dassertions=false -Dsearch_budget=SECONDS` `SECONDS`: Basic: 10000; Structures, Generics, Others: 600; Objects: 1200; Library: 1800

Pex	0.94.51006.1	`pexwizard /NoMoles /TestFramework:nunit CODE-SNIPPET`
		`pex /TimeOut:30 CODE-SNIPPET-TEST`

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 [16] and it is uniform for all the tools. Currently SETTE measures statement coverage.

Subjects of the Experiments

Three tools have been chosen for tool evaluation, which have been fully integrated into SETTE, thus the execution and coverage measurement is *automatic* for them.

- *CATG* [11] performs instrumentation and symbolic execution on Java bytecode.
- *jPET* [12] translates Java bytecode to *Prolog* and performs symbolic execution on that.
- *SPF* [13] does not translate nor instrument the bytecode, but uses a custom Java Virtual Machine, *Java PathFinder (JPF)* for execution.

To extend the findings of our experiments two significantly different tools have been included and evaluated *manually*.

- *EvoSuite* [15] uses genetic algorithms and mutation to evolve and reduce test suites.
- *Pex* [14] 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 was collected in Table I. The used tool versions and configurations are shown in Table IV.

Experimental Setup

The test execution for *CATG*, *jPET* and *SPF* was performed on an *Ubuntu 14.04 64-bit* virtual machine by SETTE, since it was the only platform which supported all the test input generators. The virtual machine was given 4 GB memory and 2 processor cores of 3.3 GHz (Intel). For execution we used Oracle's Java 7 implementation. *EvoSuite* and *Pex* were executed on the *Windows 8.1 64-bit* host OS using Oracle Java 7 and Microsoft .NET 4.0.

The chosen external *time limit* for the execution of one code snippet was 30 seconds except for *EvoSuite*. Our experiences have shown that in the given environment the first three 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). For *EvoSuite* we have run test generation separately for the main six categories, but not separately for each code snippet. Our experiences have shown that a *number of code snippets times 30 seconds* is a good choice for search budget time limit, but it should be at least 10 minutes.

The executions were performed three times for each tool and the reached coverage was the same in all of the cases. In case of *EvoSuite*, which uses randomization, the generated outputs differed in terms of input values but not in achieved statement coverage (even if we tried to increase the available time by giving a 5 minute time frame for one code snippet).

Results

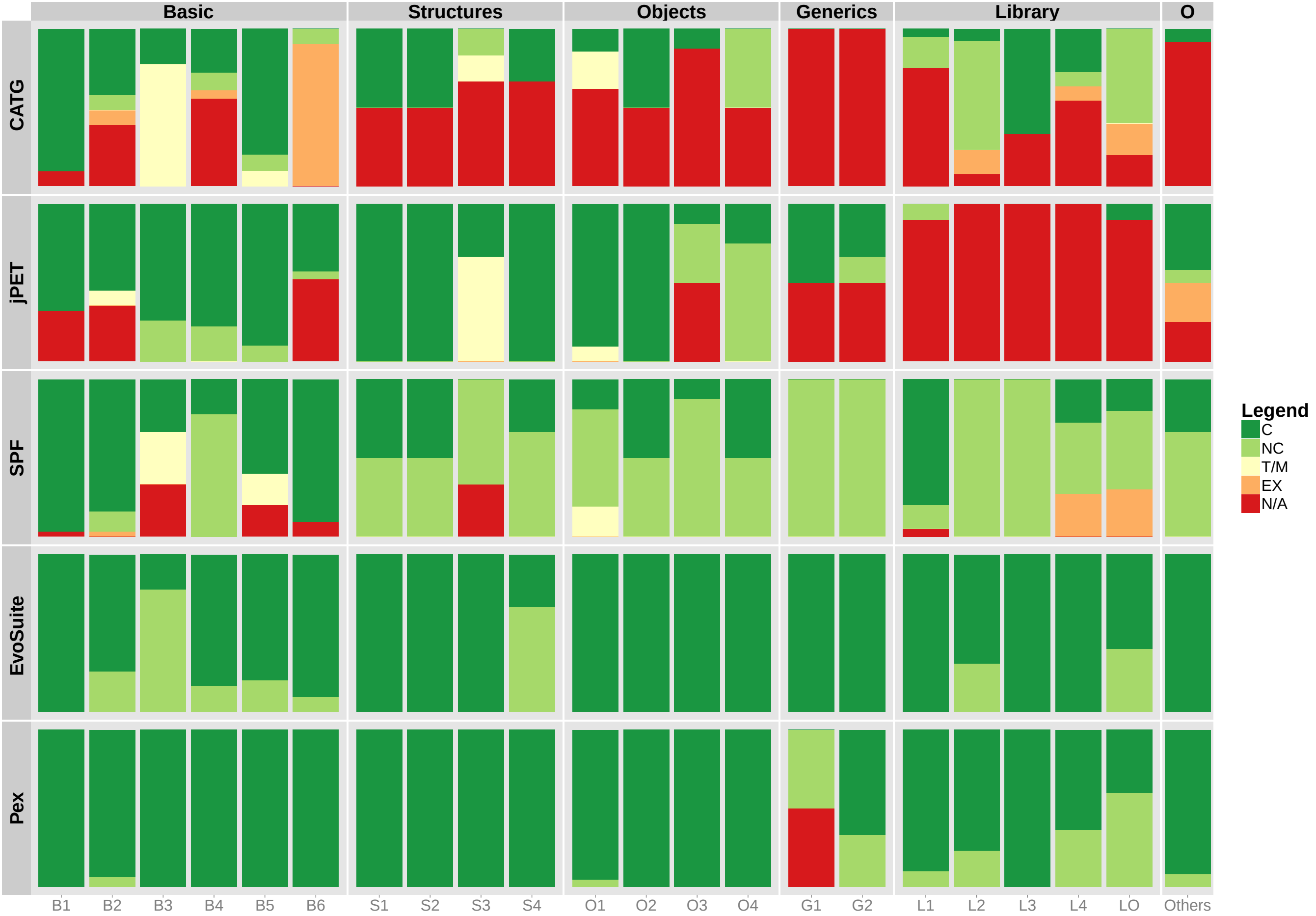
The summary of the results of the experiments can be seen in Table V. For each tool and each feature category the numbers of the code snippets classified as *C*, *NC*, *T/M*, *EX* and *N/A* are displayed. The categorization of the results were the same among three executions of the experiments. The detailed 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.

Note that the total percentage numbers should not serve as global indicators for tool quality. For example, if a tool

TABLE V
DETAILED RESULTS OF THE EXPERIMENTS

		Basic						Structures				Objects				Generics		Library					Others	Total
		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	300 (100%)
CATG	C	56	13	6	5	8		2	2		1	3	1	1				1	1	2	3		1	**105 (35%)**
	NC		3		2	1	2			1					2			4	9		1	6		31 (10.3%)
	T/M			21		1				1		5												28 (9.3%)
	EX		3		1		19												2		1	2		29 (9.7%)
	N/A	6	12		10			2	2	4	2	13	1	7	2	4	6	15	1	1	6	2	11	107 (35.7%)
jPET	C	42	17	20	14	9	9	4	4	2	3	19	2	1	1	2	2					1	5	**156 (52%)**
	NC			7	4	1	1							3	3		1	2					1	24 (8%)
	T/M		3							4		2												9 (3%)
	EX																						3	3 (1%)
	N/A	20	11				11							4		2	3	18	13	3	11	9	3	108 (36%)
SPF	C	60	26	9	4	6	19	2	2		1	4	1	1	2			16			3	2	4	**161 (53.7%)**
	NC		4		14			2	2	4	2	13	1	7	2	4	6	3	13	3	5	5	8	99 (33%)
	T/M			9		2						4												15 (5%)
	EX		1																		3	3		7 (2.3%)
	N/A	2		9		2	2			2								1						18 (6%)
EvoSuite	C	62	23	6	15	8	19	4	4	6	1	21	2	8	4	4	6	20	9	3	11	6	12	**254 (84.7%)**
	NC		8	21	3	2	2				2								4			4		46 (15.3%)
	T/M																							0
	EX																							0
	N/A																							0
Pex	C	62	29	27	18	10	21	4	4	6	3	20	2	8	4		4	18	10	3	7	4	11	**275 (91.7%)**
	NC		2									1				2	2	2	3		4	6	1	23 (7.7%)
	T/M																							0
	EX																							0
	N/A															2								2 (0.6%)

Fig. 3. Visualization of the results



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

Our detailed observations is presented in Section VI.

Threats to Validity

Reliability of the experiments: For the first three 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 *EvoSuite* and *Pex* the output was checked and categorized independently by the two authors.

Knowledge of the tools: These tools are fairly complex and configurable software (e.g. *EvoSuite* has 257 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: As listed in Table I 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*), and a tool using different underlying technique for test generation (*EvoSuite*).

VI. Discussion

This section discusses the results in the context of the two research questions.

RQ1: Feedback on the tools' capabilities

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

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

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.

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.

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.

RQ2: Hard code parts

For *CATG*, *jPET* and *SPF* there were 114 code snippets which were perfectly supported by only one of them and 68 which were supported by none of them. Considering all the five tools, 5 cases were not handled by any of them and 20 by only one of them. We have chosen and categorized the most important cases in this subsection.

Loops: the basic cases were supported by all the tools except *EvoSuite*, however, complex loops forced *CATG* and *SPF* into timeout. Only *Pex* was able to handle all the loops, however, it has sometimes reached its time limit. It must be mentioned that we intentionally implemented loops with infinite state space. *CATG* and *SPF* did not recognize infinite loops and exceeded the timeout during exploration.

Arrays: the usage of arrays is supported by all the tools, however, only *jPET*, *EvoSuite* and *Pex* are able to create arrays as inputs. In addition, there are several cases which are challenging to solve. One example is when a tool has to guess the index for a certain array element, like in Listing 2. Only *EvoSuite* and *Pex* were able to cover this snippet.

Listing 2. Code snippet for guessing array index

```
@SetteRequiredStatementCoverage(value = 100)
public static int indexParam(int index) {
  int[] numbers = new int[] { 1, 2, 3, 4, 5, ... };

  if (numbers[index] == 5) {
    return 1;
  } else {
    return 0;
  }
}
```

Structures and objects: all the tools support structures and objects, but only *jPET*, *EvoSuite* and *Pex* are able to create them as inputs. The basic code snippets focusing on structures and objects were formulated in two different ways: one accepted a structure/object as a parameter, while the other accepted the fields of the object and created the structure/object. A tricky code snippet can be seen in Listing 3. The `SimpleObject` object is a dependency, whose `addAbs(int)` method always adds the absolute value of the parameter to an internal field. In this code snippet the test input generator has to guess that number, which is when added 5 times results 10. *CATG* and *SPF* exceeds the time limit and *Pex* cannot reach maximal coverage. *jPET* is able to satisfy the condition and *EvoSuite* can generate an appropriate test suite. However, *jPET* fails on the other version of the method, which takes a `SimpleObject` instance instead of creating it.

Listing 3. Code snippet for complex method calls

```
@SetteRequiredStatementCoverage(value = 100)
@SetteIncludeCoverage(
    classes = {
        SimpleObject.class, SimpleObject.class },
    methods = {
        "getResult()", "getOperationCount()" })
public static int guessResultAndOperationCountParams
        (int x, int oc) {
  SimpleObject obj = new SimpleObject();

  for (int i = 0; i < oc; i++) {
    obj.addAbs(x);
  }

  if (obj.getResult() == 10
          && obj.getOperationCount() == 5) {
    return 1;
  } else {
    return 0;
  }
}
```

Generics was only fully supported in *EvoSuite*.

Complex arithmetic functions: *SPF* was able to handle many of the complex arithmetic functions. The constraint solver of *CATG* was unable to solve constraints in most of the cases. *jPET* could not initiate calls to arithmetic functions since they were not implemented in *Prolog*. *EvoSuite* was not able to solve some conditions and *Pex* only failed on the square root and cubic root snippets.

Strings have limited support in the first three tools (only *CATG* supports strings and this tool only supports the `String.equal(String)` method), and even *EvoSuite* and *Pex* cannot solve all the cases.

Advanced library features: 4 of the 5 snippets which were not covered by any of the tools above focus on advanced library features, such as guessing a date satisfying a format or an UUID.

3rd party library was only handled well by *SPF*, *EvoSuite* and *Pex*.

Summary: RQ2 was concerned with the code snippets for which one or none of the SE-based tools were able to generate inputs which produce maximum coverage. Using the Table V it is easy to identify those snippets that are hard to handle. In our results more than the half of the code snippets belonged to this group for pure SE-based Java tools (however, these cases can be probably traced back to a smaller number of faults). The collection of these problems could highlight the challenges, which are still active for most of the available tools.

VII. Related Work

There are several recent *survey papers* about using symbolic execution for testing purposes. Anand *et al.* [2] performed an orchestrated survey about different methods for test generation, Păsăreanu and Visser [10] summarized actual research directions, Cadar *et al.* [9] collected experiences from tool developers and Chen *et al.* [8] listed current challenges. These papers give an excellent overview of the topic, but they provide general and not tool-specific observations. Galler and Aichernig [6] presented a survey on the capabilities of 7 test data generator tools. Their goal was similar to ours but the benchmark suite is not available.

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 Fraser and Arcuri recommended the SF100 benchmark [17], a representative selection of 100 open source projects from SourceForge. Lakhotia *et al.* [4] investigated the coverage of CUTE and AUSTIN (a search-based tool) on five real-life open source components. Braione *et al.* [18] performed an experiment on an industrial control software using CREST, Pex and AUSTIN. Qu and Robinson [5] measured the coverage of CREST and KLEE on a 3.9M LOC realtime embedded system. 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.

A related problem is comparing *static analysis tools*. The Juliet test suite [19] 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 [20].

Our approach used the *code coverage* obtained by the generated test inputs as one of the success factors for a given tool. However, recent research [21], [22], [23] suggested that high code coverage is not necessarily correlated with the effectiveness of the tests. Yet as the tool developers reported coverage in their experiments we also used this metric.

VIII. Conclusion

The goal of this paper was to compare and evaluate test input generator tools. Based on the current challenges for symbolic execution and the language constructs of imperative C-like languages we identified a set of 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 experiments on five different tools. The results show that the evaluation can identify both strengths and weaknesses in the tools. Although some of the features have specifically targeted symbolic execution, the experiments with the EvoSuite tool showed that they could provide feedback on tools with different underlying techniques. Currently we are working on including other tools in the analysis and extending our evaluation method by measuring the ability of the tools to detect injected faults.

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.

Acknowledgment

The authors would like to thank Ágnes Salánki for the help with the visualization of the results. This work was partially supported by the ARTEMIS JU and the Hungarian National Research, Development and Innovation Fund in the frame of the R5-COP project.

References

[1] Institute of Electrical and Electronics Engineers, *Systems and software engineering – Vocabulary*, 12 2010, standard 24765:2010.

[2] S. Anand, E. K. Burke, T. Y. Chen, J. Clark, M. B. Cohen, W. Grieskamp, M. Harman, M. J. Harrold, and P. McMinn, "An orchestrated survey of methodologies for automated software test case generation," *J. Syst. Software*, vol. 86, no. 8, pp. 1978 – 2001, 2013.

[3] P. Godefroid, "Test Generation Using Symbolic Execution," in *Annual Conf. on FSTTCS*, 2012, pp. 24–33.

[4] K. Lakhotia, P. McMinn, and M. Harman, "An empirical investigation into branch coverage for C programs using CUTE and AUSTIN," *J. Syst. Softw.*, vol. 83, no. 12, pp. 2379–2391, Dec. 2010.

[5] X. Qu and B. Robinson, "A case study of concolic testing tools and their limitations," in *Int. Symp. on Empirical Software Engineering and Measurement*, ser. ESEM'11, 2011, pp. 117–126.

[6] S. J. Galler and B. K. Aichernig, "Survey on test data generation tools," *STTT*, vol. 16, no. 6, pp. 727–751, 2014.

[7] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[8] T. Chen, X.-s. Zhang, S.-z. Guo, H.-y. Li, and Y. Wu, "State of the art: Dynamic symbolic execution for automated test generation," *Future Generation Computer Systems*, vol. 29, no. 7, pp. 1758 – 1773, 2013.

[9] C. Cadar, P. Godefroid, S. Khurshid, C. S. Păsăreanu, K. Sen, N. Tillmann, and W. Visser, "Symbolic execution for software testing in practice: preliminary assessment," in *Proc. of the 33rd Int. Conf. on Software Engineering*, ser. ICSE '11. ACM, 2011, pp. 1066–1071.

[10] C. S. Păsăreanu and W. Visser, "A survey of new trends in symbolic execution for software testing and analysis," *Int. Journal on Software Tools for Technology Transfer*, vol. 11, no. 4, pp. 339–353, 2009.

[11] K. Sen, "CATG web page," https://github.com/ksen007/janala2, 2013, last accessed on 24/10/2014.

[12] E. Albert, M. Gómez-Zamalloa, and G. Puebla, "PET: a partial evaluation-based test case generation tool for java bytecode," in *Proc. of workshop on Partial evaluation and program manipulation*, ser. PEPM'10. ACM, 2010, pp. 25–28.

[13] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, and N. Rungta, "Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis," *Automated Software Engineering*, vol. 20, no. 3, pp. 391–425, 2013.

[14] N. Tillmann and J. Halleux, "Pex – white box test generation for .NET," in *Tests and Proofs*, ser. LNCS. Springer, 2008, vol. 4966, pp. 134–153.

[15] G. Fraser and A. Arcuri, "Whole test suite generation," *IEEE Transactions on Software Engineering*, vol. 39, no. 2, pp. 276 –291, 2013.

[16] M. R. Hoffmann, "Jacoco Java code coverage library," http://www.eclemma.org/jacoco/, 2014, last accessed on 24/10/2014.

[17] G. Fraser and A. Arcuri, "Sound empirical evidence in software testing," in *Int. Conf. on Software Engineering, ICSE'12*, 2012, pp. 178–188.

[18] P. Braione, G. Denaro, A. Mattavelli, M. Vivanti, and A. Muhammad, "Software testing with code-based test generators: data and lessons learned from a case study with an industrial software component," *Software Qual J*, vol. 22, no. 2, pp. 311–333, 2014.

[19] T. Boland and P. Black, "Juliet 1.1 C/C++ and Java test suite," *Computer*, vol. 45, no. 10, pp. 88–90, Oct 2012.

[20] A. Kossatchev and M. Posypkin, "Survey of compiler testing methods," *Programming and Computer Software*, vol. 31, no. 1, pp. 10–19, 2005.

[21] L. Inozemtseva and R. Holmes, "Coverage is not strongly correlated with test suite effectiveness," in *Int. Conf. on Software Engineering*, ser. ICSE'14. ACM, 2014, pp. 435–445.

[22] A. Pretschner, D. Holling, R. Eschbach, and M. Gemmar, "A generic fault model for quality assurance," in *Model-Driven Engineering Languages and Systems*, ser. LNCS. Springer, 2013, vol. 8107, pp. 87–103.

[23] G. Fraser, M. Staats, P. McMinn, A. Arcuri, and F. Padberg, "Does automated unit test generation really help software testers? a controlled empirical study," *ACM Trans. Softw. Eng. Methodol*, 2014, to appear.