Liu XJ, Yu P, Ma XX. An empirical study on automated test generation tools for Java: Effectiveness and challenges. JOURNAL OF COMPUTER SCIENCE AND TECHNOLOGY 39(3): 715-736 May 2024. DOI: 10.1007/s11390-023-1935-5

An Empirical Study on Automated Test Generation Tools for Java: Effectiveness and Challenges

Xiang-Jun Liu (刘相君), Ping Yu* (余　萍), *Member, CCF*
and Xiao-Xing Ma (马晓星), *Senior Member, CCF, Member, ACM, IEEE*

State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
Department of Computer Science and Technology, Nanjing University, Nanjing 210023, China

E-mail: liuxiangjun@smail.nju.edu.cn; yuping@nju.edu.cn; xxm@nju.edu.cn

Received September 24, 2021; accepted November 21, 2023.

Abstract Automated test generation tools enable test automation and further alleviate the low efficiency caused by writing hand-crafted test cases. However, existing automated tools are not mature enough to be widely used by software testing groups. This paper conducts an empirical study on the state-of-the-art automated tools for Java, i.e., EvoSuite, Randoop, JDoop, JTeXpert, T3, and Tardis. We design a test workflow to facilitate the process, which can automatically run tools for test generation, collect data, and evaluate various metrics. Furthermore, we conduct empirical analysis on these six tools and their related techniques from different aspects, i.e., code coverage, mutation score, test suite size, readability, and real fault detection ability. We discuss about the benefits and drawbacks of hybrid techniques based on experimental results. Besides, we introduce our experience in setting up and executing these tools, and summarize their usability and user-friendliness. Finally, we give some insights into automated tools in terms of test suite readability improvement, meaningful assertion generation, test suite reduction for random testing tools, and symbolic execution integration.

Keywords automated test generation, search-based software testing, random testing, symbolic execution

1 Introduction

Unit testing is an essential task in the software development life cycle[1]. However, writing high-quality unit test suites manually is time-consuming and laborious. Automated tools produce test suites in order to cover code and find behaviors that generate exceptions or runtime errors. Their automatically-generated test cases with embedded assertions (i.e., a typical type of test oracles[2]) can be used to check correctness and detect faults in the systems under test. Considering the time spent on software testing, test case generation proves to be an effective way of reducing the workload of developers. Most programming languages have their own unit testing frameworks such as JUnit① for Java, Check② for C, and unittest③ for Python. These frameworks can help to popularise unit testing. Meanwhile, many corresponding unit test generation approaches have emerged, such as search-based algorithm, random testing and symbolic execution. And a large number of automated test case generation tools for C and Java language with different approaches have begun to emerge. In recent years, automated test generation tools oriented to Java language have aroused public concern and become research hot-spot. Additionally, they are the focus of the workshop on Search-Based Software Testing (SBST④). Therefore, we decide to conduct an empirical study on automated tools for Java.

Most of the existing empirical studies lack discus-

Regular Paper
This work was supported by the National Natural Science Foundation of China under Grant Nos. 62072225 and 62025202.
*Corresponding Author
①https://junit.org/junit4/, May 2024.
②https://libcheck.github.io/check/, May 2024.
③https://docs.python.org/3/library/unittest.html, May 2024.
④https://sbst22.github.io, May 2024.

sions on automated tools that combine symbolic execution techniques. Symbolic execution makes contributions to handling dynamic data structures, which assists in generating more accurate and comprehensive test inputs. In this scenario, if state-of-the-art automated test generation tools could combine with symbolic execution in a better way, test cases are expected to contain more structural information, and thereby improve test quality.

When it comes to the metrics for evaluation, metrics are usually not comprehensive enough, for example, only including code coverage, or only including real fault detection, etc. To fill the gap and address the above limitations, our paper comprehensively evaluates and compares six automated test generation tools using different test generation approaches, which are EvoSuite[3], Randoop[4], JDoop[5], JTeXpert[6], T3[7] and Tardis[8]. Among these tools, EvoSuite and JTeXpert are search-based testing tools, Randoop and T3 are random testing tools, and JDoop and Tardis are tools that integrate symbolic execution with test case generators. Moreover, apart from code coverage, mutation score and real fault detection, we also consider the test suite size and readability as metrics in our experiments. We attempt to analyze the characteristics of the faults that can be detected by all tools, one of these tools or none of these tools. We evaluate automated test generation tools from various aspects and metrics, point out their strengths and weaknesses, and give some suggestions on these tools in order to make much more comprehensive comparisons. In summary, this paper makes the following main contributions.

• We propose a test framework to apply six automated test generation tools and evaluate the quality of unit tests produced by them.

• We discuss about the differences between JDoop and Randoop, and Tardis and EvoSuite. And we give our opinions on the effectiveness of combining test generation approaches with symbolic execution.

• We apply three automated test generation tools on the Defects4J dataset[⑤], and analyze their effectiveness of detecting real faults in the projects.

• We share our experience in applying the six automated tools and give suggested improvements from various aspects on automatic unit test generation.

The rest of this paper is organized as follows. Section 2 introduces the taxonomy of testing techniques. Section 3 illustrates tool selection principles and gives the brief introduction of selected tools. Section 4 presents our experiment methodology. Section 5 analyzes experimental results of each tool. Section 6 proposes some suggestions for future work. Section 7 discusses the potential threats to validity. Section 8 concludes this paper.

2 Taxonomy of Testing Techniques

Automated unit test case generation approaches can reduce costs and improve software development efficiency. We classify different testing techniques and their associated tools into five types, i.e., search-based testing, random testing, symbolic execution for testing, model-based testing, and hybrid techniques.

Search-Based Testing. Search-based software testing (SBST)[9] uses search algorithms to automatically generate test data by optimizing the fitness function and maximizing the achievement of test goals. The fitness function can determine the best solution from the search space of test inputs, which plays an important role in test minimization and optimization. Search-based test generation tools can define their own test goals and fitness functions. Taking EvoSuite[3] as an example, it chooses branch coverage of the whole test suite as the default test goal for coverage. Therefore, its fitness function is designed to calculate the branch distance between class under test (CUT) and test suite in order to find the optimal test suite. Search-based testing tools include JTeXpert[6], EvoSuite[3], TestFul[10], and so on.

Random Testing. As a fundamental approach for testing, random testing[1] is scalable and easy to implement. It randomly selects inputs from a program's input space and checks whether the program's behaviors on each input are correct. Random testing assists in creating error-revealing test inputs. Random testing is simple in concept and quick to find bug candidates. However, it usually produces a very large number of tests, while it ignores part of the program behaviors, and can only find basic bugs compared with other techniques. Random testing tools include Randoop[11], T3[7], JCrasher[12], etc.

Symbolic Execution for Software Testing. Symbolic execution[13] takes symbolic values instead of concrete values as inputs, and its output is a mathematical expression of these symbols. The main goal of

[⑤]https://github.com/rjust/defects4j, Mar. 2024.

symbolic execution in software testing is to explore much more different program paths. It differs from other automated test generation approaches in the use of program analysis and constraint solvers. Symbolic execution tools for Java include Symbolic PathFinder[14], JDART[15], etc.

Model-Based Testing. Model-based testing[16] depends on explicit behaviour models which encode the intended behaviour of the system under test (SUT) or its environment, and generates test cases automatically. As test suites are derived from models and not from source code, model-based testing is usually regarded as one form of black-box testing. There are several different approaches of model-based testing, such as axiomatic approaches, finite state machine (FSM) approaches, and so on. Model-based testing tools include JSXM[17], MBTsuite⑥, etc.

Hybrid Techniques. Hybrid approaches aim at making use of benefits of different test generation approaches, i.e., the advantages of one can overcome the limitations of another. Lakhotia *et al.*[18] incorporated dynamic symbolic execution (DSE) into search-based testing so as to generate better test cases by handling dynamic data structures in an effective way. Sen[19] attempted the concolic testing by combining random testing and symbolic execution with the aim of generating concrete test inputs with better coverage. Hybrid tools include SUSHI[8], Tardis[8], JDoop[5], etc.

3 Automated Unit Test Generation Tools Selection

In this section, we illustrate selection principles and describe the selected tools in our experiment. We exclude pure symbolic execution tools (e.g., JDART[15], and Symbolic PathFinder[14]) because they almost only produce test inputs. Table 1 shows an overview of unit test generation tools that we include or exclude in our experiment, where N/A means there is no specified version of the tool, and "Executable Jar" means

Table 1. Overview of Automated Test Generation Tools

Tool	Technique	Artifact	Updated	Input Format	Output Format	Selected	Version
EvoSuite[3]	Search-based testing	Open source	2021	Java binary code	JUnit4 test cases	Yes	1.2.0
Randoop[11]	Random testing	Open source	2022	Java binary code	JUnit4 test cases	Yes	4.3.0
JDoop[5]	Random testing + concolic execution	Open source	2018	Java source and binary Code	JUnit4 test cases	Yes	2.0
JTeXpert[6]	Search-based testing	Executable Jar	2016	Java source and binary code	JUnit4 test cases	Yes	1.4
T3[7]	Random testing	Open source	2019	Java binary code	Binary test suites trace files (.tr)	Yes	N/A
Tardis[20]	Search-based testing + symbolic execution	Open source	2021	Java binary code	JUnit4 test cases	Yes	0.1.0
JCrasher[12]	Random testing	Executable Jar	2007	Java binary code	JUnit3 test cases	No	2.1.3
Tpalus[21]	Random testing + symbolic execution	Executable Jar	2010	Java binary code	JUnit3 test cases	No	0.2
GRT[22]	Random testing	Not open source	2015	Java binary code	JUnit4 test cases	No	N/A
TestFul[10]	Search-based testing	Open source	2010	Java binary code	JUnit3 test cases	No	1.0.4
SUSHI[8]	Search-based testing + symbolic execution	Open source	2021	Java binary code	JUnit4 test cases	No	0.2.0
GraphWalker[23]	Model-based testing	Open source	2021	Finite state machines	Test paths	No	4.3.1
JSXM[17]	Model-based testing	Open source	2016	EFSM (stream X-machines)	JUnit test cases	No	N/A

⑥https://www.mbtsuite.com, May 2024.

the tool only provides an executable package, without open source code. In the table, we mark the selected and unselected tools, and record the tool version used in our experiment.

3.1 Selection Principles

First, a tool should provide command-line interface for our experiment. All tools mentioned in Table 1 are command-line tools, and thus other automatic tools that cannot generate test cases in a command-line way are directly excluded. Second, the selected tools are supposed to provide open source code or executable jars. Third, we discard tools that have not been maintained since 2016. Fourth, test suites generated by these tools should be in JUnit4 format, or can be loaded in JUnit4 format (e.g., T3). As shown in Table 1, JCrasher[12], Tpalus[21] and TestFul[10] have been out of maintenance for many years, which are all excluded. GRT[22] is also excluded because it does not provide source code or executable package. Additionally, when two tools use the same approach, we choose the one that has the better performance of test generation. For example, Tardis[20] is based on SUSHI and aims at overcoming some limitation of SUSHI, and thus we only choose Tardis in this experiment.

Model-based testing tools rely on an extra predefined model for system under test, and require users to create MBT models from requirement or system specifications. We exclude model-based testing tools in our experiment because few programs are developed with models or formal specification. Furthermore, tests generated by model-based tools are usually not in JUnit format. For example, GraphWalker only generates useful test paths for a specified model, and test cases generated by JSXM are in XML format and have to be transformed into JUnit test cases by using Java Test Transformer.

3.2 Overview of Selected Tools

Based on selection principles, we finally choose six state-of-the-art automated unit test generation tools. EvoSuite[3] and Randoop[11] are the most commonly used automated test generation tools in academia. The other tools are also representative in the field of test generation, and valuable for the empirical study and future research.

EvoSuite. EvoSuite[3] is a search-based test generation tool. It leverages a genetic algorithm to search the program state space, and evolve and generate tests. The genetic algorithm applied in its test generation module treats the test suite as a chromosome, using search operators (e.g., crossover, mutation) to evolve individuals for each population. During the evolutionary process, the fitness function makes contributions to minimizing the test suite and choosing the best.

Randoop. Randoop[11] is one of the most used automated tools based on feedback-directed random testing. Randoop selects method sequences at random and creates test sequences incrementally. Futhermore, the newly generated sequences with no contract violations are outputted as regression tests. The error-revealing test reveals the code that violates the contract and indicates an error. Currently, Randoop mainly checks for a default set of contracts, e.g., *reflexivity of equality*, *contracts over Object.clone*().

JDoop. JDoop[⑦] combines the Java PathFinder's concolic execution engine JDART and the random testing generator Randoop. It can create test cases automatically in a hybrid way. First, JDoop collects test cases generated by Randoop, and then randomly selects some test cases for the next step. Second, JDART executes concolic testing and generates new concrete values. Finally, concrete inputs are written into test files, and have an impact on the next round execution of Randoop.

JTeXpert. JTeXpert[6] can automatically construct the whole test suites in the JUnit format for each CUT by using a search heuristic. This tool takes source code and dependencies of Java projects as input. JTeXpert utilizes a source code analyzer to collect program information, a testcase candidates builder to explore useful sequences, and a random search approach to randomly generating candidate test cases for each uncovered branch. Thus, JTeXpert can reach many branches that may be covered accidentally, thereby improving code coverage.

T3. T3[7] randomly generates a great number of test sequences for Java classes. T3 consists of two tools: the generator tool and the replay tool. It can generate clean test sequences without throwing any exception and inject oracles in these clean sequences. Additionally, T3 imposes pair-wise testing[7] in order to find bugs that are caused by faulty interactions between methods in the CUT.

[⑦] https://github.com/psycopaths/jdoop/, May 2024.

Tardis. Tardis[20] consists of a Java bytecode symbolic executor (JBSE)[24] and a customized version of the search-based test generator based on EvoSuite. Tardis integrates the two main modules through a novel approach, and can produce tests in an automated way. The path condition information generated by the path explorer would be fed to the test generation module. Like JDoop, Tardis also needs to allocate time for symbolic execution and test generation.

4 Experiment Methodology

In this section, we present our experiment methodology. With the goal of comparing the state-of-the-art automated test generation tools from different perspectives, we define the evaluation metrics and summarize the following four research questions.

- *RQ*1 (*Quality of Test Cases*). What code coverage, mutation score, test suite size, and readability are achieved by each automated tool given different time budgets?
- *RQ*2 (*Effects of Combination with Symbolic Execution*). Does combining test generation approaches with symbolic execution improve the quality of automatically-generated test cases?
- *RQ*3 (*Real Fault Detection*). How many existing faults can be detected by these automated tools on different projects? Which category of faults is easier to be found?
- *RQ*4 (*Ease of Use*). How much effort does it take for developers to set up these automated tools?

We conduct our experiment on Ubuntu 18.04 with Intel Xeon® Gold 5117 CPU @ 2.00 GHz with 125 GB of RAM, and use OpenJDK's Java VM (JVM) version 1.8.0_292.

4.1 RQ1: Quality of Test Cases

4.1.1 Selected CUTs for RQ1 Experiment

Table 2 shows the detailed information of six well-known open-source projects which we select in our experiment. These projects are Apache BCEL[8], jsoup[9], ZXing Core[10], Apache Commons Lang[11], JFreeChart[12], and Apache Commons Collections[13]. We choose these projects as benchmarks because they are mature and canonical. All of the projects are hosted on GitHub and used by hundreds or even thousands of artifacts. Additionally, these projects are developed by different organizations and have different functionalities. Moreover, selecting multiple projects can increase the scientificity and diversity of our experiment.

We apply EvoSuite, Randoop, JDoop, JTeXpert, T3, and Tardis to generate test cases for the classes in different projects. Since abstract classes and interface classes do not have much practical value for this experiment, we exclude them and choose classes with more branches and statements. Finally, we select 75 CUTs from the six open source projects as mentioned above. Table 3 shows the characteristics of these selected CUTs.

4.1.2 Overview of Test Workflow

We write scripts to automatically run tools, generate unit tests and compute evaluation metrics. We perform 3 (repetitions) × 6 (automated tools) × 75 (CUTs) × 5 (timebudgets) = 6 750 runs in total. Therefore, this setting requires about 19 days of generation time.

Table 2. Overview of Benchmarks Under Study

Benchmark	Version	#LOC	#Cls.	#Bran.	#Stat.	Description
Apache BCEL	6.0	60 547	376	6 020	15 680	A library to analyze and manipulate Java class files
jsoup	1.11.3	18 076	67	3 712	7 319	A library to fetch URLs and manipulate data
ZXing Core	3.3.2	38 145	231	7 890	13 311	A core barcode encoding/decoding library
Apache Commons Lang	3.13	88 285	216	10 052	15 976	A package of utility classes for Java platform
JFreeChart	1.5.2	218 039	641	21 852	52 378	A comprehensive free chart library
Apache Commons Collections	4.5	74 020	351	5 975	13 499	A package of many powerful data structures

Note: #LOC means the total lines of code; #Cls. means the total number of classes; #Bran. means the number of branches in the bytecode (measured by JaCoCo); #Stat. means the number of statements.

[8] https://commons.apache.org/proper/commons-bcel/, May 2024.
[9] https://jsoup.org/, May 2024.
[10] https://zxing.github.io/zxing/, May 2023.
[11] https://commons.apache.org/proper/commons-lang/, May 2024.
[12] https://www.jfree.org/jfreechart/, May 2024.
[13] https://commons.apache.org/proper/commons-collections/, May 2024.

Table 3. Characteristics of Selected CUTs

Benchmark	#CUTs	#Stat.	#Bran.	#Met.	#Cyc. Comp.
Apache BCEL	15	1 650	1 078	280	832
jsoup	15	1 765	1 156	322	905
ZXing Core	10	1 240	892	89	538
Apache Commons Lang	10	2 113	1 609	353	1 171
JFreeChart	15	961	516	161	419
Apache Commons Collections	10	411	260	91	222

Note: #CUTs means the number of selected CUTs; #Stat. means the number of statements; #Bran. means the number of branches; #Met. means the number of methods; #Cyc. Comp. means the number of cyclomatic complexity.

The workflow of this experiment is shown in Fig.1. The overall experiment procedure is as follows.

• First, we use the command line tools to produce test suites. The inputs of the test generation module are the CUT's name and the time budget. We run each tool three times with five different time budgets: 10 s/class, 60 s/class, 180 s/class, 360 s/class, and 600 s/class. The outputs of test generation module are test files, which include Java files and binary trace files (outputted by T3).

• Second, we remove test cases that cannot be successfully compiled or wrong. Illegal test cases are supposed to be modified in order to make the whole test suite compilable and executable.

• Third, we write scripts to automatically evaluate each automated tool. The inputs of the scripts are the tool name, time budget, CUT name, test files, and dependencies. The outputs of the scripts are files containing the results of the code coverage, mutation score, and test suite size. JaCoCo[14] is used for code coverage analysis. PITEST[15] is used to perform mutation testing and evaluate tools. In terms of the test suite size, we collect the number of test cases in the test file with the help of JavaParser.

• Finally, we collect and analyze all of the outputs. In this step, we compute the average of experimental results and rank them for evaluation.

4.1.3 Metrics

As for the metrics to evaluate the quality of test cases, we draw on and extend the findings of Grano *et al.*[25].

1) *Code Coverage*[26]. Code coverage describes the proportion of source code tested in the program, and helps measure the software quality. In our experiment, code coverage includes branch coverage, line coverage, cyclomatic complexity coverage, method coverage, and instruction coverage.

2) *Mutation Score*[27]. In mutation testing, some modifications would be injected into the program. Each mutated version is called a mutant. If a test detects and rejects mutants, we consider that it kills the mutants. Mutation score calculates the proportion of

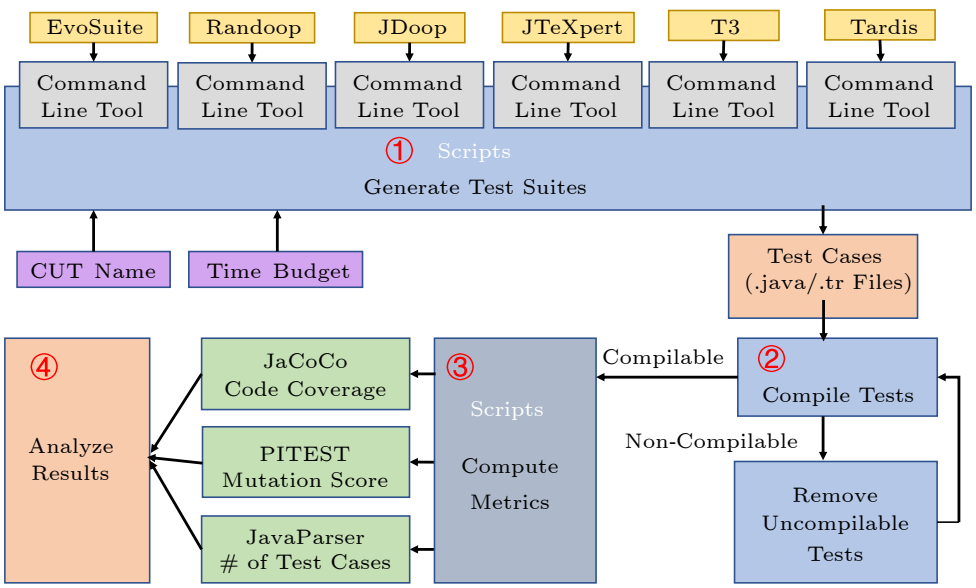

Fig.1. Experimental flowchart of six automated test generation tools. #: number.

[14] https://github.com/jacoco/jacoco/, May 2024.
[15] https://pitest.org/, May 2024.

the number of killed mutants to the total number of mutants.

3) *Test Suite Size*. Test suite size refers to the total number of test cases included in a suite. Generally, a smaller test suite is easier to understand. The larger the test suite, the higher the test budget and time cost.

4) *Readability*[28]. Readability reflects the comprehensibility of test cases. We use Readability Checker[16] to compute readability of tests. It implements three software readability metrics, including B&W, comments ratio (CR), and software readability ease score ($SRES$).

- *B&W*[29]. The model proposed by Buse and Weimer[29] outputs a readability score in the range [0, 1]. The higher the B&W score, the better the readability.
- *Comments Ratio* (CR)[30]. CR is a metric from the aspect of comments. The lower the metric CR, the better the readability. The proposed formula is as follows, where LOC represents the number of total lines of code, and LOM represents the number of lines with comments:

$$CR = LOC/LOM.$$

- *Software Readability Ease Score* ($SRES$)[31]. $SRES$ is less sensitive to comments and whitespace, but correlates well with human readability experience. The lower the metric $SRES$, the better the readability. The proposed formula is as follows, where ASL refers to the average sentence length and AWL refers to the average word length:

$$SRES = ASL - 0.1 \times AWL.$$

4.2 RQ2: Effects of Combination with Symbolic Execution

The main purpose of combining symbolic execution and a test generator is to achieve the desired effect that can not only capture relevant structural information but also generate test cases, thereby making up for their own shortcomings.

Fig.2 shows the workflow of JDoop. JDoop adopts a hybrid approach that combines concolic testing with random testing. It relies on JDART and Randoop. The procedure can be divided into three parts. First, JDoop collects the generated test cases during the execution of Randoop. Second, JDoop randomly selects a small part of test cases to perform concolic execution in JDART. JDART records symbolic constraints on executed program paths, replaces concrete values with symbolic variables, and then uses constraint solvers to generate new concrete test inputs with the goal of exploring feasible distinct execution paths and obtaining better coverage than random testing. Third, new generated concrete test input values are written into test files, and also applied for the next execution.

Fig.3 shows the workflow of Tardis. The symbolic executor JBSE integrated in Tardis identifies execution conditions of system under test, and interacts

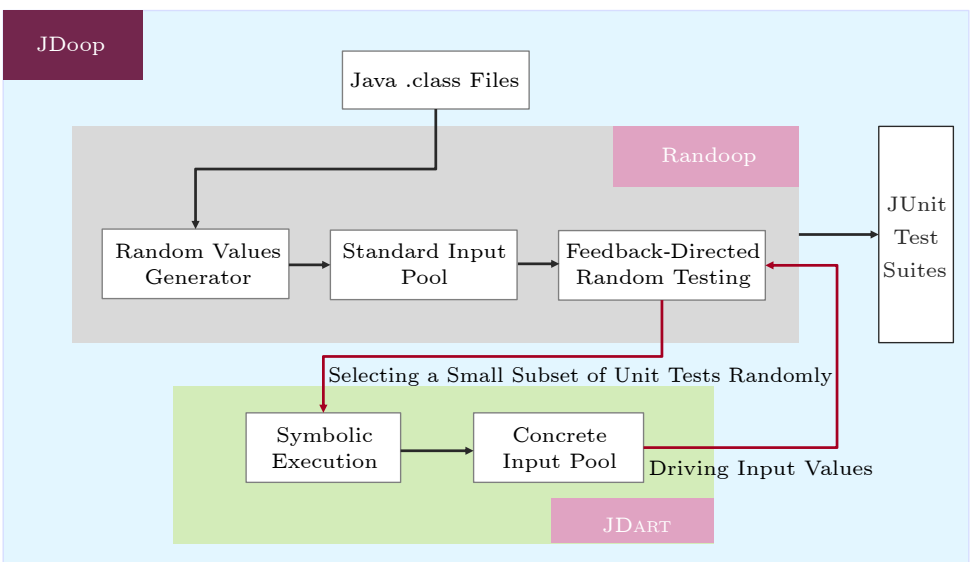

Fig.2. Workflow overview of JDoop with symbolic execution.

[16] https://github.com/cdtpinto/readabilitychecker/, May 2024.

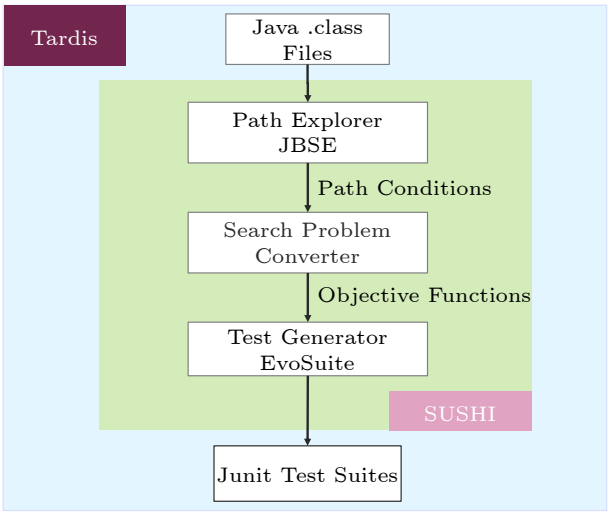

Fig.3. Workflow overview of Tardis with symbolic execution.

with the test case generator EvoSuite for optimising legal method sequences incrementally. Tardis is different from those approaches that use symbolic execution techniques to explore alternative values of the initial test inputs. Tardis makes full use of symbolic execution to generate path conditions that characterise the dependencies between program paths and complex input structures (e.g., Object), converts path conditions into the objective function of optimisation problems[8], and then exploits EvoSuite to generate concrete method sequences.

We compare JDoop with Randoop, and Tardis with EvoSuite by the experimental results of RQ1. As random testing tools, JDoop and Randoop can directly generate test suites for hundreds of CUTs in an entire project at a time under the specified time budget, and thus we add an additional experiment for further evaluation. The benchmarks used in the additional experiment are mentioned in Table 2. We focus on branch coverage and line coverage in this experiment (RQ2), because JDoop attempts to improve branch coverage by leveraging concrete test input values, and line coverage is a basic metric that can check every executable statement. Besides, we want to analyze whether applying concolic testing has an impact on the branch coverage and line coverage, and discover whether the quality of automatically-generated tests is related to different benchmarks.

4.3 RQ3: Real Fault Detection

Revealing real bugs in the project is the key point of software testing. However, test suites with a high code coverage or high mutation score are not certain to do well in detecting real bugs in the project.

Thus, we evaluate the real fault detection ability of automatic tools. We use the Defects4J dataset[17] to compute the real fault detection rate of an automatically-generated test suite. The Defects4J dataset consists of 17 open-source projects with 835 bugs, and distinguishes between active and deprecated bugs. We choose four projects with 187 real bugs from the Defects4J dataset for our experiment. These four projects have their identifiers named as jsoup, Lang, Chart and Collections, which are also included in the previous experiment for RQ1. Table 4 shows the number of bugs of each project under test. Defects4J has two versions of the program: the buggy version and the fixed version. Each buggy program version contains exactly one real fault.

Table 4. Overview of Projects from Defects4J Dataset

Identifier	Project Name	#Bugs
jsoup	jsoup	93
Lang	Apache Commons Lang	64
Chart	JFreeChart	26
Collections	Apache Commons Collections	4

Note: #Bugs means the number of bugs in the dataset.

We leverage the Defects4J test framework and make some changes to the test scripts so that JTeXpert can be successfully applied. We first generate test suites for a program, and then run it on the buggy version to check whether the specific bug can be detected. Note that Defects4J removes all flaky tests, uncompilable tests and failed tests before running the test suite. If a test produces different results when we retry to run it for many times, then the test can be regarded as a flaky test. The result of bug detection is "Pass", "Fail", or "Broken". "Pass" means the bug is detected by tests, and "Fail" means the bug is not found by tests. "Broken" means the tests have their own problems and cannot work properly when running on the buggy program so that they fail to determine whether the bug is detected.

Considering that almost all the tests generated by JDoop are "Broken" when performing fault detection, the results have little research value, and thus we discard JDoop from our experiment. T3 produces binary trace files as the test suite, which is not suitable for the Defects4J framework. Tardis disables to successfully generate tests for the buggy projects due to its

[17] https://github.com/rjust/defects4j, Mar. 2024.

own flaws. Finally, we only select EvoSuite, Randoop and JTeXpert as our target tools to evaluate real fault detection ability, and attempt to find what category of bugs is easier to find.

In brief, we conduct our experiments on different time budgets, which are set to 120 s and 360 s, respectively. As for Randoop, the time budget is the global time budget for the whole project. Since EvoSuite and JTeXpert are search-based test generation tools, the time budget is set for one CUT at a time. In order to alleviate randomness and contingency, we run each tool 10 times and aggregate the number of real bugs detected by all of the generated test suites. In total, it takes approximately 22 days to produce test suites for the selected buggy version projects.

4.4 RQ4: Ease of Use

The purpose of this empirical study aims to make readers easy to select and apply these tools, and give researchers in the field a basic summary. First, we describe the challenges we face during the tools setup process, and show our efforts to run automated test generation tools successfully. Second, we sum up the barriers and problems encountered in our experiments. Third, we analyze the usability and friendliness of these tools, and their combinability with modern IDE.

5 Experimental Results

In this section, we outline and analyze the experimental results of each automated test generation tool on different time budgets to answer the RQs mentioned in Section 4.

5.1 Answering RQ1: Quality of Test Cases

We take the average of three rounds of experimental results in order to avoid the randomness to some extent. We compare and analyze results on the given time budget from four aspects: code coverage, mutation score, test suite size, and readability, respectively.

5.1.1 *Results of Code Coverage*

Different test generation approaches influence the quality of automatically-generated tests. In this experiment, each tool generates unit tests for one CUT at a time. Given an automated tool T, a time budget B, and a coverage metric M, we provide the following formula to evaluate the average code coverage result (ACCR) for each CUT j on each execution i, where the total number of CUTs (named as N) is 75 and the total number of executions (named as R) is 3:

$$\mathrm{ACCR}_{<T,\ B,\ M>} = \frac{\sum_{i=1}^{R}\sum_{j=1}^{N} cov_{<T,\ B,\ M,\ i,\ j>}}{N \times R}.$$

Table 5 presents the ACCR of branch coverage, line coverage, complexity coverage, method coverage and instruction coverage of each automated test generation tool on five different time budgets. EvoSuite performs the best with the highest coverage in all aspects. JTeXpert ranks only second to EvoSuite in code coverage. The results of EvoSuite and JTeXpert far exceed the other tools. They are search-based tools and can explore the state space of the program the more fully when the time budget increases. T3, Randoop and JDoop have little difference in code coverage. Among these three tools, T3 outperforms Randoop and JDoop. But T3 is lack of producing negative tests[7], which explains the lower code coverage of T3 compared with search-based test generators. Randoop generates more but redundant test cases as time budget grows higher, which only has little benefit to code coverage. JDoop integrates Randoop as the test generator. Since JDoop injects symbolic execution in order to maximize branch coverage, the code coverage of unit tests generated by JDoop for one CUT at a time is higher than that of Randoop. Tardis attempts to make use of path information generated by the path explorer in order to produce tests for a specific path with the goal of increasing code coverage. However, the results are unsatisfying, and are the worst among the results of all the six tools. In Subsection 5.1.2, we analyze the reasons and give our own opinions.

5.1.2 *Results of Mutation Score*

Fig.4 shows mutation coverage and mutation score of each automated test generation tool on five different time budgets. The results of test suite executed by PITEST include *NO_COVERAGE*, *SURVIVED* and *KILLED*. *NO_COVERAGE* is the same as *SURVIVED* except that there are no tests that execute the line of code where the mutation is created[18].

[18] https://pitest.org/quickstart/, May 2024.

Table 5. Average Code Coverage for Six Automated Test Generation Tools on Different Time Budgets

Time Budget (s)	Tool	Average Branch Coverage (%)	Average Line Coverage (%)	Average Complexity Coverage (%)	Average Method Coverage (%)	Average Instruction Coverage (%)
10	EvoSuite	63.78	76.86	73.30	89.64	76.28
	Randoop	27.12	39.06	35.07	56.77	37.21
	JDoop	27.86	42.05	37.07	61.35	40.14
	JTeXpert	35.90	53.58	45.93	70.24	52.09
	T3	29.73	43.21	37.51	60.43	41.03
	Tardis	18.46	37.57	29.94	60.88	35.45
60	EvoSuite	71.84	82.90	81.86	93.02	82.59
	Randoop	30.08	43.34	39.44	61.67	41.43
	JDoop	30.76	43.96	39.70	61.71	42.17
	JTeXpert	54.62	70.35	63.98	85.19	68.82
	T3	32.97	48.52	43.24	66.37	46.54
	Tardis	21.98	42.26	32.93	67.77	40.45
180	EvoSuite	74.23	84.76	84.42	94.27	84.76
	Randoop	30.61	43.57	39.96	61.70	41.72
	JDoop	31.30	44.06	40.17	61.59	42.28
	JTeXpert	55.51	71.19	65.04	84.95	69.75
	T3	32.40	47.77	42.00	66.81	45.81
	Tardis	22.24	42.76	33.98	69.02	40.90
360	EvoSuite	76.72	86.96	86.71	95.94	86.88
	Randoop	30.87	43.70	40.23	61.70	41.85
	JDoop	31.63	44.31	40.78	61.81	42.57
	JTeXpert	56.52	72.37	65.92	86.44	70.87
	T3	34.28	49.01	43.11	65.99	46.85
	Tardis	22.99	43.21	33.57	68.16	41.50
600	EvoSuite	77.27	87.51	87.25	95.99	87.41
	Randoop	31.00	43.75	40.33	61.70	41.90
	JDoop	31.97	44.47	41.91	61.84	42.77
	JTeXpert	57.73	73.09	66.82	86.16	71.71
	T3	32.61	47.84	41.86	65.66	45.80
	Tardis	22.99	44.93	34.80	70.09	42.79

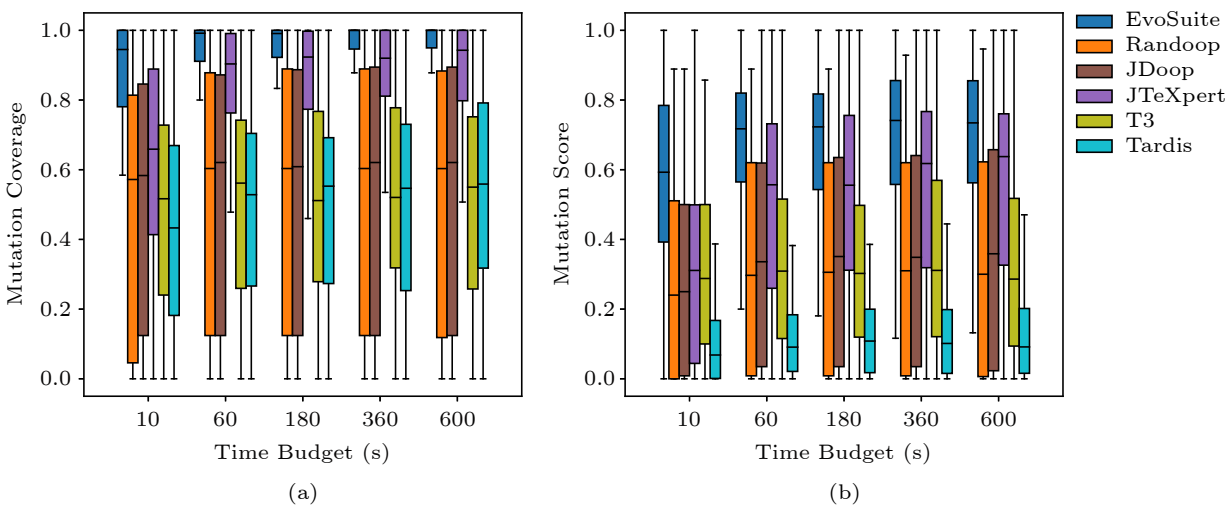

Fig.4. Average (a) mutation coverage and (b) mutation score for six automated test generation tools on different time budgets.

Mutation coverage is the proportion of *SURVIVED* and *KILLED* to all results as *NO_COVERAGE* implies no tests covered the mutation. Mutation score reflects the ability of killing seeded mutants (i.e., the ratio of *KILLED* to all results).

EvoSuite beats the other tools in terms of mutation coverage and mutation score. The main reason is that EvoSuite applies mutation testing itself and removes the assertion that cannot detect any of the remaining seeded mutants. In terms of average muta-

tion score, JTeXpert performs better than the remaining four automated tools under all time budgets. An oracle builder implemented in JTeXpert is supposed to generate meaningful assert statements. It uses the value returned from a method call as an oracle and leads to much stronger ability of killing mutants. Tardis still performs the worst among these six automated tools. Its mutation score is much lower than those of the other tools.

As for random testing tools, when the time budget grows, they show a trivial improvement in mutation coverage and mutation score. Random testing tools, like Randoop, create assertions with intelligent guessing. Randoop generates assertions guided by user-defined contracts and the pertaining logic. The results indicate that random testing tools cannot generate more effective unit tests under a larger time budget. Many of the generated method sequences prove to be useless and redundant, which are of no help to the improvement of test quality. In general, Randoop is more suitable for testing small-scale programs with short test generation time on the basis of good code coverage and mutation score.

5.1.3 Results of Test Suite Size

Test suite size means the number of test cases in a test suite. Given an automated tool T and a time budget B, we use the following formula to compute average test suite size (ATSS) for each CUT j on each execution i, where the total number of CUTs (named as N) is 75 and the total number of executions (named as R) is 3:

$$ATSS_{<T,\ B>} = \frac{\sum_{i=1}^{R}\sum_{j=1}^{N} size_{<T,\ B,\ i,\ j>}}{N \times R}.$$

Table 6 shows the ATSS results of each automated test generation tool on five different time budgets. Except for T3, other tools' test suite size always increases as the time budget increases. The size of test suites generated by random testing tools is the largest, and proportional to the growth of the time budget. Randoop and JDoop generate much more test cases for a given CUT than the other tools. As the time budget is larger and larger, more time should be allocated to concolic testing, which may limit the number of unit tests generated by JDoop. In general, the number of test cases generated by T3 is less related to the time budget, but the code coverage and mutation score increase as the test suite size increases.

As for the search-based test generation tools, the results of the test suite size are all lower than 60. EvoSuite uses a generic algorithm to produce and minimizes the test suite. The number of test cases generated by JTeXpert is even less than that of EvoSuite, but the length of a test case (i.e., the number of lines of code contained in a test case) is much longer than that of EvoSuite. Tardis gets the lowest test suite size, and meanwhile, its code coverage and mutation score are also the worst. However, the low size of test suite does not guarantee the quality of tests.

In general, too large test suites decrease test efficiency, increase test cost, and may affect readability. Excessive test suite is one of the bottlenecks of random testing tools.

5.1.4 Results of Readability

The readability of automatically-generated test cases does not relate to the time budget; hence, we set time budget to 10 s and compute the readability of the tests produced by six tools. T3's test suite is a binary file, which is not readable. In this experiment, we take out T3 and only compare the remaining five automated test generation tools.

Fig.5 represents the average score of B&W, CR and SRES. As for the B&W score, search-based test generation tools perform better than the other tools. JTeXpert ranks first. As the numbers of characters, identifiers and keywords are all regarded as metrics to train models, the long but clear sequences generated by JTeXpert get a pretty good score. EvoSuite applies various optimisation strategies (e.g., test minimization) to improve the readability.

Table 6. Average Test Suite Size for Six Automated Test Generation Tools on Different Time Budgets

Time Budget (s)	EvoSuite	Randoop	JDoop	JTeXpert	T3	Tardis
10	38.3	255.3	286.8	11.4	288.5	5.6
60	46.9	1 649.7	1 666.6	18.5	309.5	9.1
180	51.4	4 898.3	3 790.8	19.8	305.6	9.7
360	56.4	9 685.4	6 445.2	20.6	444.3	10.0
600	57.6	16 057.6	10 747.1	21.7	409.0	10.2

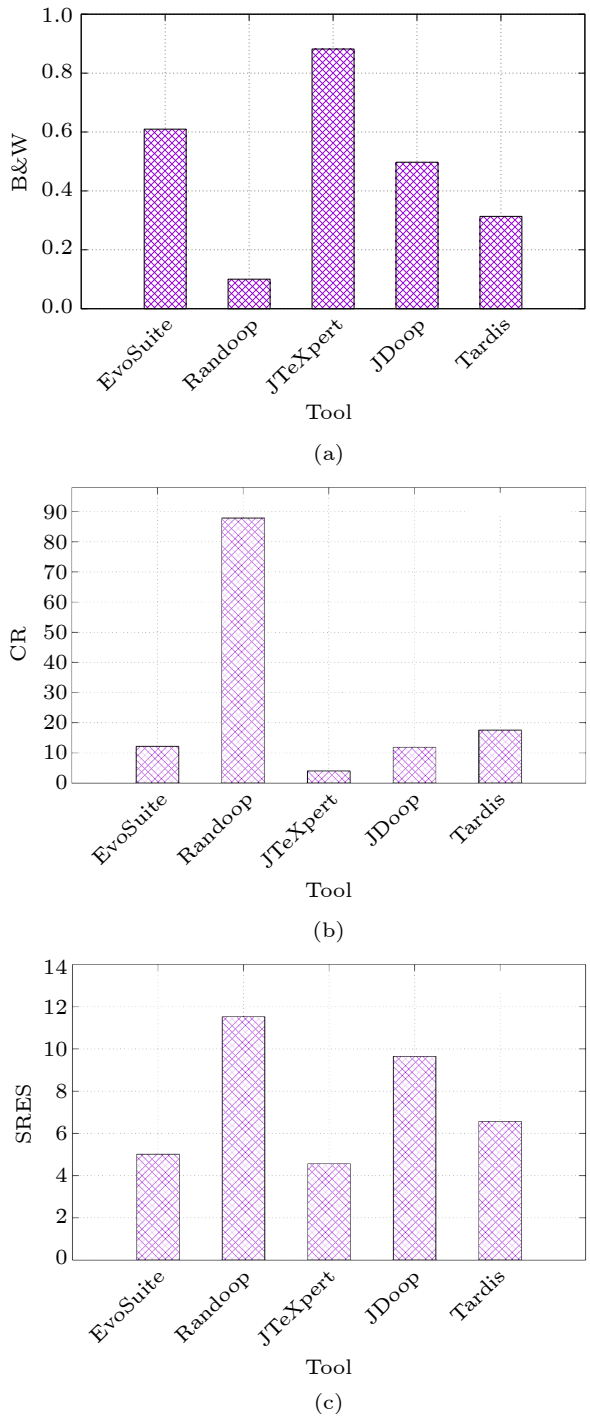

Fig.5. Average (a) B&W, (b) CR, and (c) SRES of different automated test generation tools.

As for the CR score, we remove the results whose number of comments is zero because the denominator cannot be zero. JTeXpert is still the distinguished tool. JTeXpert produces comments before each test method to explain its main content and covered paths.

As for the SRES score, three tools that implement search-based test generators are outstanding. JDoop and Randoop often generate very long sentences for a test case, which affect the overall readability.

5.2 Answering RQ2: Effects of Combination with Symbolic Execution

We compare JDoop with Randoop, and Tardis with EvoSuite to check the effects of the combination of symbolic execution.

5.2.1 JDoop vs Randoop

The experimental results of RQ1 show that JDoop is better than Randoop in code coverage when generating tests for one CUT at a time. Through our verification, JDoop can generate new test inputs which do not exist in the test suites generated by Randoop for given CUTs. The correct test inputs for test methods may explain that JDoop has better quality of tests constructed for one CUT.

In our additional experiment, JDoop performs poorly when it comes to generating tests for an entire project. Fig.6 shows the comparison results between JDoop and Randoop under the time budget of 10 s, 360 s and 600 s, respectively. Randoop outperforms JDoop in most cases, and JDoop produces more uncompilable unit tests than Randoop during the test generation process. Randoop outperforms JDoop in most cases. We can see that the code coverages of test suites are related to different benchmarks and test suite sizes; however, we cannot find a clear relationship between the benchmark and coverage ratio. But in general, larger test suite size always leads to higher code coverage.

We find that JDoop's performance on one CUT and an entire project has certain differences. Concolic testing is a hybrid technique that interleaves concrete execution with symbolic execution. Under a given time budget, there exist lots of paths in complex programs that concolic testing tools (e.g., JDoop) cannot cover, because no constraint solver can support these tools to cover all reachable branches[32]. Apart from that, JDoop selects test cases at random, and calls JDART to execute symbolized unit tests. First, JDART can only generate a small number of test inputs in primitive types (e.g., Int, Char, Boolean), but cannot symbolize complex types such as Array and Object, which influences its effect and

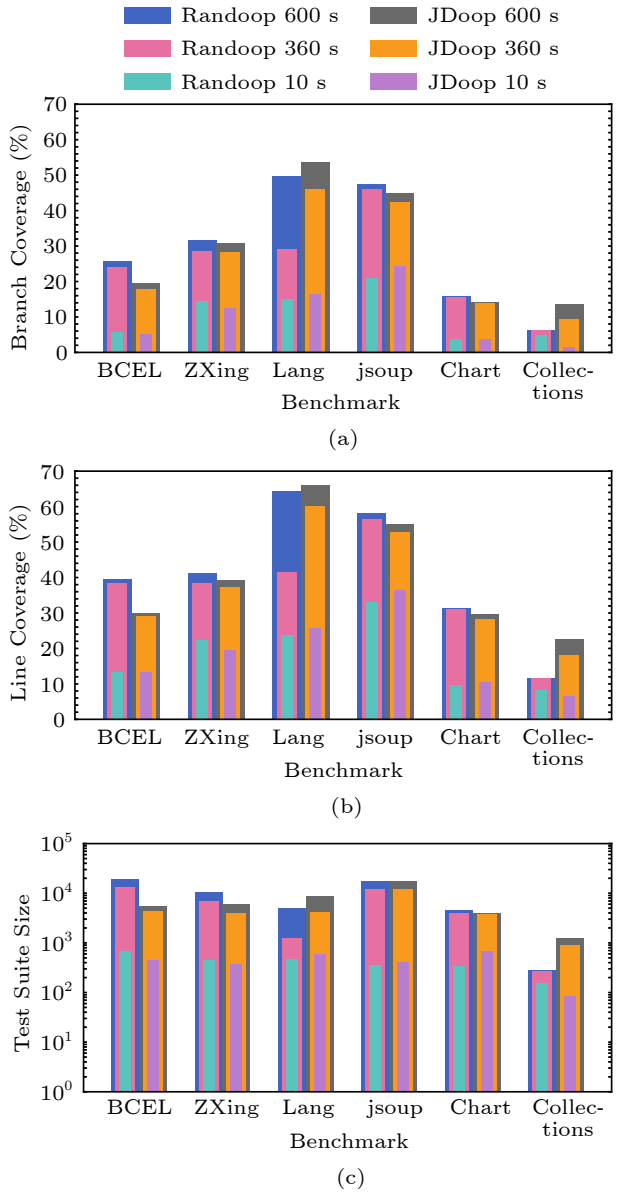

Fig.6. Average (a) branch coverage, (b) line coverage and (c) test suite size on the whole projects from JDoop and Randoop.

applications. Second, JDoop shuffles test cases and gets scrambled indexes. Then it applies the test cases to concolic execution by order until JDART runs out of time limit. However, this process often chooses test cases that do not need more comprehensive test inputs, e.g., the method under test contains no reachable branch. Third, JDoop uses a customized version Randoop as the test generator. To sum up, the effectiveness of the constraint solver and the limitations of JDoop can explain why JDoop's optimization performance is not so significant after incorporating symbolic execution.

Moreover, concolic testing works well when the programs can be tested as a single unit[33]. The experiments of RQ1 focus on one CUT at a time, which can be regarded as a unit because a class in one CUT is small and all methods in the class just implement a single functionality cohesively[32]. However, in our additional experiment, an entire project is large and usually composed of multiple independent functionalities, leading to a very complex situation. Due to time budget, concolic testing consumes much time to generate some concrete but limited test inputs, and the final improvement is trivial. Furthermore, if we want to test a large industrial project with JDoop, we should consider a tradeoff between costs of partitioning and setting up the units for testing[33].

5.2.2 *Tardis vs EvoSuite*

According to the results of RQ1, Tardis attempts to improve test generation by incorporating the path condition information created by JBSE and applies the search-based testing tool EvoSuite to instantiate complex data structures satisfying the path conditions, but it performs worse than EvoSuite on all metrics. We summarize the following four main problems of Tardis. First, from the logs during the test generation process, we find that Tardis often fails to apply EvoSuite to produce test cases for the given path conditions and prints out the exception message "Failed to generate the test case for path condition". Second, the time limit of test generation used for other automated tools is between 0 and $2T$ (where T is the time budget). However, it takes more than $2T$ for Tardis to complete test generation. Third, we have to change the default setting of -Dassertions from false to true in order to generate assertions. But most of the types of assert statements are "fail". Fourth, Tardis is fragile and its documentation is not detailed enough for us to solve problems.

Furthermore, we discuss about the reasons of the experimental results. On the one hand, Tardis is a variant of SUSHI, which aims at generating concrete test inputs for programs with complex heap inputs. The authors of SUSHI evaluated its performance on Java classes with paths that involve complex interprocedural dependencies and complex data structures[8]. However, the CUTs used in our experiment are not specially selected like this and not very suitable to explore in-depth of the programs for complex heap inputs by means of JBSE. Moreover, it takes half of the time budget for symbolic execution, which may affect the overall test generation. It im-

plies that Tardis and SUSHI would encounter such problems when generating tests for various kinds of industrial programs, which discourages their popularity. On the other hand, Tardis cannot instantiate the path conditions that are spurious due to unsound computation of JUnit test cases. For example, the CUTs have many infeasible branches or loop iterations. And this may explain why the logs show a lot of failure-related messages. Therefore, the symbolic execution process of JBSE needs to be improved with the goal of enhancing the effectiveness of the explored program paths, and avoiding problems such as path explosion.

Note that EvoSuite itself involves a dynamic symbolic execution (DSE) module, called EvoSuiteDSE. Galeotti *et al.*[34] incorporated DSE into EvoSuite and proposed a novel adaptive approach combining the genetic algorithm with DSE. DSE can be used for handling dynamic data structures and optimizing primitive values. In this subsection, we analyze the differences of these two approaches. First, when it comes to EvoSuiteDSE, during the process of search-based exploration, the generic algorithm can determine whether to apply DSE or not. For example, if a CUT has many branches that depend on numerical constraints on test inputs, DSE is suitable in this situation. EvoSuiteDSE aims at selecting individuals of the population to perform symbolic execution. Applying symbolic execution on meaningful cases during the search process can save the cost. However, Tardis just allocates a fixed time limit to symbolic execution. Second, Tardis converts path conditions into optimization problems. It leverages EvoSuite to generate test cases by using the evaluator programs of these path conditions as fitness functions[8]. This may affect EvoSuite's normal process of generating test cases because of some existing infeasible path conditions. Nevertheless, the combination approach of DSE module and EvoSuite is based on the theory that DSE can be applied as a local search step on primitive values in the sequences of method invocations[34]. Besides, EvoSuite applies global search to explore the whole population, and individuals can be improved by means of DSE during the search process. This way of combination better integrates the strengths of each one at a lower cost, and as a result, it would find more uncovered branches. While Tardis works better in terms of creating complex objects than Galeotti *et al.*'s method[34], it is not very suitable for improving EvoSuite on code coverage of CUTs used in our experiment. In summary, more effective and executable improvements of combining symbolic execution with search-based test generation need to be studied in the future.

5.3 Answering RQ3: Real Fault Detection

We compute the bug detection ratio by aggregating the bugs found by the 10 test suites automatically generated by each tool in order to avoid the randomness. Table 7 shows the fault detection results in Defects4J benchmarks.

Table 7. Fault Detection Results in Defects4J

Time Budget (s)	Tool	jsoup	Lang	Chart	Collections	Total
120	EvoSuite	23	22	16	0	61
	Randoop	25	13	16	0	54
	JTeXpert	24	20	10	1	55
360	EvoSuite	25	22	16	0	63
	Randoop	26	15	16	0	57
	JTeXpert	20	18	11	1	50

5.3.1 *Fault Detection Results*

Considering tools individually, EvoSuite, Randoop and JTeXpert can detect 63, 57, and 55 bugs at best in our experiment, respectively. Among the three automated test generation tools, EvoSuite is still the distinguished. And test suites with the highest fault detection rate for Lang are generated by EvoSuite. The time budget has little effect on improving the fault detection rate. However, a more thorough study on various time budgets would be a meaningful future work.

5.3.2 *How Faults Found by Different Automated Tools?*

We analyze the faults detected or not detected by automated tools from the following three perspectives, and attempt to answer these questions.

1) *How Many Times Can Faults Be Found?* Under a given time budget, the faults covered by test suites generated by automated tools on different executions are not the same. We count up the number of faults that can be found at least 90% of times in 10 executions (marked as Always), and the number of faults that are detected only once in 10 executions

(marked as Once), and the number of faults in other cases (marked as General). Fig.7 shows the percentages of the three types of faults detected by different automated tools.

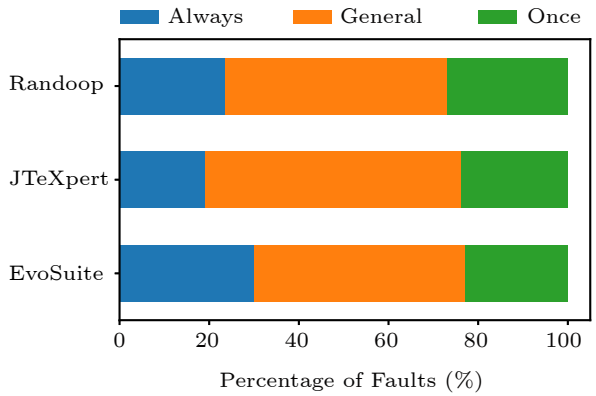

Fig.7. Percentage of faults that can be Always or Once detected by different automated tools.

In EvoSuite, the faults that can be found by all generated test suites accounts for the largest proportion. Randoop has the largest number of faults that are found by only one test suite. As for JTeXpert, the faults with the type of Always have the least proportion. This means that JTeXpert produces the lowest overlap of faults covered by different test suites. In summary, the 10 test suites generated by the three automated tools have similar coverage of faults. However, in terms of fault detection, tests generated by EvoSuite are the most stable.

2) *Analysis of Faults Detected or Not Detected.* The ability of different tools to find bugs is various. From the results, we classify the faults into the following three cases. Some faults (jsoup-8b, Lang-60b, etc.) can be detected by all tools during 10 executions. These faults do not require complex conditions or concrete input values. For example, Lang-60b is shown in Fig.8, where the reference of *thisBuf.length* would trigger errors.

```
    public boolean contains (char ch) {
        char[] thisBuf = buffer;
-       for (int i = 0; i < thisBuf.length; i++) {
+       for (int i = 0; i < thisBuf.size; i++) {
            if (thisBuf[i] == ch) {
                return true;
            }
        }
        return false;
    }
```

Fig.8. Example of faults that can be detected by all tools during 10 executions, Lang-60b.

Some cases (Chart-12b, Lang-65b, Collections-27b, etc.) can only be found out by one of the tools during 10 executions. With regard to search-based tools, EvoSuite is possibly better at generating more complex input strings, as it uses a genetic algorithm to fully explore the program state space, leverage runtime values, and then produce meaningful input strings and assert statements. Test cases generated by JTeXpert perform better in dealing with complex conditions, as its search strategy can help explore more unexpected branches. Besides, the test cases generated by JTeXpert usually contain more assertions than the other tools. For example, Lang-24b can only be detected by JTeXpert. The following code snippet shown in Fig.9 is a description of Lang-24b, which requires the generated test suite to meet complex conditions.

```
    if (chars[i] == '1' || chars[i] == 'L') {
        // not allowing L with an exponent
-       return foundDigit && !hasExp;
+       return foundDigit && !hasExp && !hasDecPoint;
    }
```

Fig.9. Example of faults that can only be found out by one of the tools during 10 executions, Lang-24b.

However, there are also cases (Chart-3b, Lang-31b, jsoup-57b, etc.) where none of the tools can detect the fault during 10 executions. Collections-26b is a fault that cannot be found by the three tools. None of these three tools generate tests for private classes or methods, and thus this type of faults is much more difficult to be found. Collections-26b is shown in Fig.10.

```
-   private Object readResolve(){
+   protected Object readResolve(){
        calculateHashCode(keys);
        return this;
    }
```

Fig.10. Example of faults that cannot be found out by any of the tools during 10 executions, Collections-26b.

3) *Which Category of Faults Is Easier to Be Found?* The number of bugs and the category of bugs found by different tools are comparable. As Almasi *et al.*[35] suggested, we classify the faults into two categories: assertion-based and exception-based.

- *Assertion-Based Faults.* These faults are detected by assertions in the test case. Assertions include utility methods that support the asserting conditions in tests, including *assertEquals*, *assertTrue*, *assertNull*, *fail*, etc. Only failing assertions are recorded. For example, Chart-8b can be found by manual tests through assertions. Fig.11(a) shows the information of Chart-8b by using Defects4J. The fault is detected by a JUnit assertion (i.e., $assertEquals(35, w.getWeek());$) at line 11 in Fig.11(b), which is expected to be 35 but 34 obtained. And it triggers *AssertionFailedError* to reveal this error.
- *Exception-Based Faults.* An unhandled excep-

```
Summary for Bug: Chart-8
---------------------------------------------------------------
Revision ID (fixed version) :
1085
---------------------------------------------------------------
Revision date (fixed version) :
2008-06-10 02:19:11 -0700
---------------------------------------------------------------
Bug report id:
UNKNOWN
---------------------------------------------------------------
Bug report url:
UNKNOWN
---------------------------------------------------------------
Root cause in triggering tests:
- org.jfree.data.time.junit.WeekTests::testConstructor
  --> junit.framework.AssertionFailedError: expected: <35> but was: <34>
  ------------------------------------------------
```

(a)

```
@Test
public void testConstructor() {
    // Other content…

    GregorianCalendar cal = (GregorianCalendar) Calendar.getInstance(
        TimeZone.getDefault(), Locale.getDefault ()) ;
    Date t = cal.getTime();
    Week w = new Week(t, TimeZone.getTimeZone("Europe/Copenhagen"));
    t = cal.getTime():
    w = new Week(t, TimeZone.getTimeZone( "Europe/Copenhagen"));
    assertEquals(35, w.getWeek());

    // Other content...
}
```

(b)

```
Summary for Bug: Jsoup-5
---------------------------------------------------------------------------------------------
Revision ID (fixed version):
62afbdf6a85fec75fde38d327928a180bc5bccb3
---------------------------------------------------------------------------------------------
Revision date (fixed version):
2011-07-02 21:11:58 +1000
---------------------------------------------------------------------------------------------
Bug report id:
32
---------------------------------------------------------------------------------------------
Bug report url:
https://github.com/jhy/jsoup/issues/32
---------------------------------------------------------------------------------------------
Root cause in triggering tests:
 - org.jsoup.parser.ParserTest::parsesQuiteRoughAttributes
   -> java.lang.StringIndexOutOfBoundsException: String index out of range: 14
---------------------------------------------------------------------------------------------
```

(c)

```
@Test
public void parsesQuiteRoughAttributes() {
    String html = "<p =a>One<a =a";
    Document doc = Jsoup.parse(html);
    assertEquals("<p>One<a></a></p>", doc.body().html());

    doc = Jsoup.parse("<p ... ");
    assertEquals("<p></p>", doc.body().html());

    doc = jsoup.parse("<p ... <p!!");
    assertEquals("<p></p>\n<p></p>", doc.body().html());
}
```

(d)

Fig.11. Information of (a) Chart-8b and (c) jsoup-5b by using Defects4J, and the test code that detects the faults (b) Chart-8b and (d) jsoup-5b.

tion is thrown in the test code, causing the fault to be detected. We collect several common exceptions, such as *NullPointerException*, *IOException*, *IllegalArgumentException*, *StringIndexOutOfBoundsException* and so on. For example, jsoup-5b can be found by manual tests through throwing exceptions. Fig.11(c) illustrates the information of jsoup-5b by using Defects4J. The test code shown in Fig.11(d) throws *StringIndexOutOfBoundsException* because the index of a String variable is out of range during the execution process.

Based on the classification, we calculate the proportion of different categories of faults detected by automated tools. Generally speaking, JTeXpert is better at detecting assertion-based faults than the other two tools. Especially for Lang, the number of assertion-based faults found by JTeXpert is much greater than that of exception-based faults. In terms of Chart, all tools have a little gap between assertion-based faults and exception-based faults. It can be seen from Fig.12 that it is easier for all the three automated tools to find out assertion-based faults than exception-based faults. In the future, we will attempt to improve the assertion generation process and strengthen the ability of detecting exception-based faults.

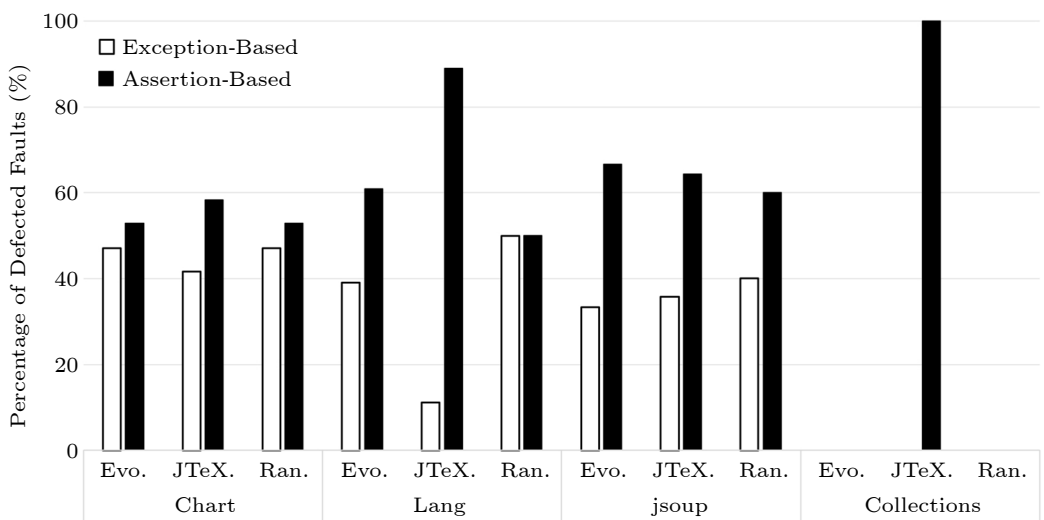

Fig.12. Percentage of detected faults with different categories: assertion-based and exception-based. Evo: EvoSuite; JTex.: JTeXpert; Ran.: Randoop.

5.4 Answering RQ4: Ease of Use

In fact, we face many challenges when setting up and executing these tools. It takes a certain amount of time for developers to successfully run all the tools, especially tools combined with symbolic execution. Among them, four tools provide executable jars directly, i.e., EvoSuite, Randoop, JTeXpert, and JDoop. As for T3, we have to build the jar from source code. The output of T3 is a binary trace file (.tr), and it should be loaded by the defined test files. As for JDoop, we need to prepare for the experiment environment (e.g., JPF-core, jConstraints, and Z3) in order to meet prerequisites and execute the tool. As for Tardis, we should build jars from source code, and install Z3 and other dependent packages locally.

5.4.1 *Encountered Problems*

First, we need detailed and updated manuals or documentations when using automated test generation tools. Developers may have to put more efforts into the tool setup process. It prevents the wide usage of automated tools in real industrial development. Second, we encounter some problems in executing Tardis and JDoop. We modify the source code of JDoop and add *--testclass* option, so that it can support the test generation for one CUT at a time. Apart from this, Tardis generates test files each containing one test case. It is not helpful in computing code coverage and mutation score. Hence, we write a script to merge all test cases together in one test suite class. Third, different versions of tools often have different ways of running them, such as Randoop. Therefore, the differences are required to be clearly written in the manual so as to avoid unnecessary barriers.

5.4.2 *Usability & Friendliness*

In terms of usability, EvoSuite and Randoop are the most practical tools among all the six tools and relatively more suitable for industrial projects, as they are easy-to-use, extensible, and stable. On the contrary, developers need to make more effort to successfully configure and run JTeXpert, T3, and JDoop. JTeXpert often throws exceptions when generating tests, but we cannot get its source code to solve this problem. Tardis is an academic tool, which focuses more on functionality than on usability, and thus we have spent the most time on it. Furthermore, EvoSuite has plugins for Maven, Eclipse and IntelliJ IDEA. Though its plugins are not suitable for any type of experiments, they help to improve its usability. Randoop is proven useful in practical contexts, and is supported by development environments (e.g., Eclipse). Besides, Tardis can also work under Eclipse if we successfully deploy it. Other tools do not have Maven plugin or IDE plugin.

In terms of user-friendliness, EvoSuite provides the most clear documentations and its runtime log feeds back the progress and status of test generation in real time. T3 produces binary test suites, and it is not user-friendly. The test generation process of Randoop and JDoop is relatively simple and smooth, but the automatically-generated test suites are too large, which increases the time required to compile and run tests and impairs user-friendliness. Tardis is relatively the worst. The time it takes to create tests is often uncontrollable and far exceeds the given time budget. In summary, automated tools still need perfect related documentations and functionalities to improve their usability and user-friendliness, with positive implications in generating unit tests automatically, thereby reducing the burden on developers and improving test efficiency.

6 Suggestions

According to the results of our experiment and related work, we propose the following suggestions for automated unit test generation tools.

6.1 Readability Improvements

From our experimental results, automated tools often generate complicated and obscure unit tests that are difficult to comprehend and maintain for humans. To overcome this problem, for example, Daka *et al.*[36] proposed a domain-specific machine learning model to predict the readability of tests based on human judgements, and used this model to augment the test generation process of EvoSuite to improve readability.

According to test cases produced by automated tools and their results in Fig.5, there are too few comments to provide some hints about scenarios. Improving the quality of comments in test cases is a feasible way to generate more readable test cases. Panichella *et al.*[37] proposed a template-based approach to automatically generate summaries as comments for each test case to improve understandability. DeepTC-En-

hancer[38] uses a template-based approach to create comments by summarizing the scenarios of test cases. It may be an effective way to use the mentioned approaches to create more meaningful comments for test methods, and then incorporate the comment generation module with automated test case generation process.

6.2 Meaningful Assertions

Automated tools can create assertions on return values, compare objects with each other or call inspector methods on objects[3]; however, only a finite number of things can be asserted. As Zhang and Mesbah[39] inferred, the number of assertions and the types of assertions in a test suite are strongly correlated with test suite effectiveness. To this end, generating meaningful assertions is one of the key challenges in test generation.

From the tests generated by automated tools in our experiments, we observe that assertions are usually incomplete, and lack contextual information and necessary complexity to capture a specified fault. Apart from traditional heuristic approaches, there are many studies making use of machine learning technologies. Watson *et al.*[40] employed an approach based on neural machine translation (NMT) to automatically generate meaningful assert statements. Tufano *et al.*[41] proposed a sequence-to-sequence transformer model to generate accurate assertions for methods under test. To sum up, NLP technologies for assertion generation are expected to further improve the ability of error detection.

Furthermore, we can attempt to generate assertions based on specifications in test automation. Cheon and Leavens[42] proposed an approach that automatically writes the unit test oracles by using a formal specification language's runtime assertion checker to decide whether the program's methods are working correctly. Parameterized unit testing[43] can be considered as a type of specification[44]. For example, Pex[43] can instantiate the parameterized unit tests by determining test inputs with systematic program analysis.

6.3 Test Suite Reduction

From our experimental results, we can find that random testing tools (e.g., Randoop) generate many redundant test cases for the systems under test. Most of the test cases are not helpful to increase test quality and efficiency, but cost much time to execute. Test suite reduction approaches are supposed to decrease test costs and improve test understandability. Jaygarl *et al.*[45] implemented a tool called GENRED that combines a sequence-based reduction approach and a coverage-based reduction approach for Randoop to remove redundant test cases. Cruciani *et al.*[46] proposed a novel test suite reduction approach for very large-scale systems. It is based on evaluating the similarity of test cases and can keep code coverage unchanged. Chetouane *et al.*[47] introduced a test suite reduction method that combines K-means clustering with binary search. They illustrated that the clustering algorithm could significantly reduce the number of test cases.

In general, it is worth studying the subject how to make use of runtime information (e.g., the branches reached by newly produced method sequences) during the test generation process and incorporate feasible test suite reduction approaches with random testing tools in a better way in the future.

6.4 Hybrids of Symbolic Execution and Other Test Generation Approaches

The experimental results of JDoop are slightly better than those of Randoop for either one CUT or the whole project. When it comes to suggestions for JDoop, we highlight the following aspects. First, as we analyze above, JDART has many limitations but it also provides some extensibility. Improving JDART may be an effective method to enhance the overall test quality of JDoop. For example, Mues and Howar[48] improved JDART by implementing four strategies: exploration strategies, bounded analysis, path-specific constraint solving strategies, and the SMT-Lib string approach. Second, we can make use of runtime information and execution trace to select test cases for concolic execution. Third, the constraint solver applied in concolic testing considerably affects the capability of tools and needs to be improved.

Both search-based software testing and the symbolic execution technique have been shown to be effective in many papers. Baluda[49] presented an evolutionary algorithm, EvoSE, which searches and traverses for symbolic paths that do not satisfy the minimized branch conditions in the program. Besides this, fuzz testing is also a dynamic analysis technique. Ol-

sthoorn *et al.*[50] combined grammar-based fuzzing with search-based testing tool EvoSuite to generate highly-structured input data and maximize the code coverage for Java JSON projects. In general, further clarification and improvements are still needed in the next step.

7 Threats of Validity

As for a threat to external validity, our experiments only study on six automated test case generation tools, mainly based on search-based testing, random testing, and symbolic execution. It is worth noting that many other techniques, such as combinatorial testing, fuzzing testing, specification-based testing and so on, are not discussed in our paper. However, these approaches might be more appropriate in other complex cases, not in our study.

We conduct our experiments on Java open source programs[19]. It may not be generalized for other programming languages. Besides, we only select 75 CUTs from six projects to study on RQ1, and ensure that the selected CUTs contain more branches and statements. As a reference, the SBST workshop also uses selected CUTs from open source projects as the benchmark. Generally speaking, the benchmark of our study is reasonable. It can also be reused in other empirical studies.

With regard to internal validity, we use JaCoCo, PITEST and Readability Checker to compute code coverage, mutation score and readability, respectively. The instrumentation may bias our results. But all the results can be reproduced with the same settings. We use three objective indicators to measure the relatively subjective metric, i.e., readability. Because most people have little concept of automated tools, their judgment on the readability of tests may deviate from the actual situation. Therefore, the indicators of readability are more scientific and reliable in our experiment.

The threat to construction validity is that the experiment of RQ1 depends on the test workflow with scripts designed by us to automatically generate test suites and compute evaluation metrics. Besides, due to the number of projects and the test suites generated, we could not perform an overall manual inspection to verify whether the tests are stable, and we use the Defects4J test framework to remove flaky tests when comparing the real fault detection ability.

8 Conclusions

This paper presented systematic and comprehensive experiments on automated unit test generation tools in terms of code coverage, mutation score, test suite size and readability. According to the results, search-based tools perform the best. While random testing tools produce the largest test suite size that affects test efficiency. Tests generated by EvoSuite are the most stable in fault detection. We also found out that assertion-based faults are easier to be found than exception-based faults. Furthermore, there is much room for improvement on integrating symbolic execution with test generation approaches. In the next step, we will discuss about commercial tools and their applications in large-scale industrial software.

Conflict of Interest The authors declare that they have no conflict of interest.

References

[1] Anand S, Burke E K, Chen T Y, Clark J, Cohen M B, Grieskamp W, Harman M, Harrold M J, McMinn P. An orchestrated survey of methodologies for automated software test case generation. *Journal of Systems and Software*, 2013, 86(8): 1978–2001. DOI: 10.1016/j.jss.2013.02.061.

[2] Chen J J, Bai Y W, Hao D, Zhang L M, Zhang L, Xie B. How do assertions impact coverage-based test-suite reduction? In *Proc. the 2017 IEEE International Conference on Software Testing, Verification and Validation* (*ICST*), Mar. 2017, pp.418–423. DOI: 10.1109/ICST.2017.45.

[3] Fraser G, Arcuri A. EvoSuite: Automatic test suite generation for object-oriented software. In *Proc. the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, Sept. 2011, pp.416–419. DOI: 10.1145/2025113.2025179.

[4] Pacheco C, Lahiri S K, Ernst M D, Ball T. Feedback-directed random test generation. In *Proc. the 29th International Conference on Software Engineering* (*ICSE'07*), May 2007, pp.75–84. DOI: 10.1109/ICSE.2007.37.

[5] Dimjašević M, Rakamarić Z. JPF-Doop: Combining concolic and random testing for Java. *Collections*, 2013, 422(3894): 58470. https://dimjasevic.net/marko/2013/11/17/presented-jpf-doop-at-java-pathfinder-workshop-2013/jpf-workshop-2013.pdf, Mar. 2024.

[6] Sakti A, Pesant G, Guéhéneuc Y G. Instance generator and problem representation to improve object oriented code coverage. *IEEE Trans. Software Engineering*, 2015,

[19] https://github.com/rjust/defects4j, Mar. 2024.

41(3): 294–313. DOI: 10.1109/TSE.2014.2363479.

[7] Prasetya I S W B. T3, a combinator-based random testing tool for Java: Benchmarking. In *Proc. the 1st International Workshop on Future Internet Testing*, Nov. 2013, pp.101–110. DOI: 10.1007/978-3-319-07785-7_7.

[8] Braione P, Denaro G, Mattavelli A, Pezzè M. Combining symbolic execution and search-based testing for programs with complex heap inputs. In *Proc. the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis*, Jul. 2017, pp.90–101. DOI: 10.1145/3092703.3092715.

[9] Panichella A, Kifetew F M, Tonella P. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information and Software Technology*, 2018, 104: 236–256. DOI: 10.1016/j.infsof.2018.08.009.

[10] Baresi L, Lanzi P L, Miraz M. TestFul: An evolutionary test approach for Java. In *Proc. the 3rd International Conference on Software Testing, Verification and Validation*, Apr. 2010, pp.185–194. DOI: 10.1109/ICST.2010.54.

[11] Pacheco C, Ernst M D. Randoop: Feedback-directed random testing for Java. In *Proc. the Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, Oct. 2007, pp.815–816. DOI: 10.1145/1297846.1297902.

[12] Csallner C, Smaragdakis Y. JCrasher: An automatic robustness tester for Java. *Software: Practice and Experience*, 2004, 34(11): 1025–1050. DOI: 10.1002/spe.602.

[13] King J C. Symbolic execution and program testing. *Communications of the ACM*, 1976, 19(7): 385–394. DOI: 10.1145/360248.360252.

[14] Păsăreanu C S, Rungta N. Symbolic PathFinder: Symbolic execution of Java bytecode. In *Proc. the 25th IEEE/ACM International Conference on Automated Software Engineering*, Sept. 2010, pp.179–180. DOI: 10.1145/1858996.1859035.

[15] Mues M, Howar F. JDart: Dynamic symbolic execution for Java bytecode (competition contribution). In *Proc. the 26th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Apr. 2020, pp.398–402. DOI: 10.1007/978-3-030-45237-7_28.

[16] Li W B, Le Gall F, Spaseski N. A survey on model-based testing tools for test case generation. In *Proc. the 4th International Conference on Tools and Methods for Program Analysis*, Mar. 2017, pp.77–89. DOI: 10.1007/978-3-319-71734-0_7.

[17] Dranidis D, Bratanis K, Ipate F. JSXM: A tool for automated test generation. In *Proc. the 10th International Conference on Software Engineering and Formal Methods*, Oct. 2012, pp.352–366. DOI: 10.1007/978-3-642-33826-7_25.

[18] Lakhotia K, Harman M, McMinn P. Handling dynamic data structures in search based testing. In *Proc. the 10th Annual Conference on Genetic and Evolutionary Computation*, Jul. 2008, pp.1759–1766. DOI: 10.1145/1389095.1389435.

[19] Sen K. Concolic testing. In *Proc. the 22nd IEEE/ACM International Conference on Automated Software Engineering*, Nov. 2007, pp.571–572. DOI: 10.1145/1321631.1321746.

[20] Braione P, Denaro G. SUSHI and TARDIS at the SBST2019 tool competition. In *Proc. the 12th IEEE/ACM International Workshop on Search-Based Software Testing* (*SBST*), May 2019, pp.25–28. DOI: 10.1109/SBST.2019.00016.

[21] Chitirala S C R. Comparing the effectiveness of automated test generation tools "EVOSUITE" and "Tpalus" [Master's Thesis]. University of Minnesota, Minnesota, 2015.

[22] Ma L, Artho C, Zhang C, Sato H, Gmeiner J, Ramler R. GRT: Program-analysis-guided random testing (T). In *Proc. the 30th IEEE/ACM International Conference on Automated Software Engineering* (*ASE*), Nov. 2015, pp.212–223. DOI: 10.1109/ASE.2015.49.

[23] Zafar M N, Afzal W, Enoiu E, Stratis A, Arrieta A, Sagardui G. Model-based testing in practice: An industrial case study using graphWalker. In *Proc. the 14th Innovations in Software Engineering Conference* (*Formerly Known as India Software Engineering Conference*), Feb. 2021, Article No. 5. DOI: 10.1145/3452383.3452388.

[24] Braione P, Denaro G, Pezzè M. JBSE: A symbolic executor for Java programs with complex heap inputs. In *Proc. the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, Nov. 2016, pp.1018–1022. DOI: 10.1145/2950290.2983940.

[25] Grano G, De Iaco C, Palomba F, Gall H C. Pizza versus Pinsa: On the perception and measurability of unit test code quality. In *Proc. the 2020 IEEE International Conference on Software Maintenance and Evolution* (*ICSME*), Sept. 28–Oct. 2, 2020, pp.336–347. DOI: 10.1109/ICSME46990.2020.00040.

[26] Hemmati H. How effective are code coverage criteria? In *Proc. the 2015 IEEE International Conference on Software Quality, Reliability and Security*, Aug. 2015, pp.151–156. DOI: 10.1109/QRS.2015.30.

[27] Papadakis M, Kintis M, Zhang J, Jia Y, Le Traon Y, Harman M. Mutation testing advances: An analysis and survey. *Advances in Computers*, 2019, 112: 275–378. DOI: 10.1016/bs.adcom.2018.03.015.

[28] Winkler D, Urbanke P, Ramler R. What do we know about readability of test code?—A systematic mapping study. In *Proc. the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering* (*SANER*), Mar. 2022, pp.1167–1174. DOI: 10.1109/SANER53432.2022.00135.

[29] Buse R P L, Weimer W R. Learning a metric for code readability. *IEEE Trans. Software Engineering*, 2010, 36(4): 546–558. DOI: 10.1109/TSE.2009.70.

[30] Aggarwal K K, Singh Y, Chhabra J K. An integrated measure of software maintainability. In *Proc. the Annual Reliability and Maintainability Symposium* (*Cat. No. 02CH37318*), Jan. 2002, pp.235–241. DOI: 10.1109/RAMS.2002.981648.

[31] Börstler J, Caspersen M E, Nordström M. Beauty and the

beast: On the readability of object-oriented example programs. *Software Quality Journal*, 2016, 24(2): 231–246. DOI: 10.1007/s11219-015-9267-5.

[32] Kannavara R, Havlicek C J, Chen B, Tuttle M R, Cong K, Ray S, Xie F. Challenges and opportunities with concolic testing. In *Proc. the 2015 National Aerospace and Electronics Conference* (*NAECON*), Jun. 2015, pp.374–378. DOI: 10.1109/NAECON.2015.7443099.

[33] Qu X, Robinson B. A case study of concolic testing tools and their limitations. In *Proc. the 2011 International Symposium on Empirical Software Engineering and Measurement*, Sept. 2011, pp.117–126. DOI: 10.1109/ESEM.2011.20.

[34] Galeotti J P, Fraser G, Arcuri A. Improving search-based test suite generation with dynamic symbolic execution. In *Proc. the 24th IEEE International Symposium on Software Reliability Engineering* (*ISSRE*), Nov. 2013, pp.360–369. DOI: 10.1109/ISSRE.2013.6698889.

[35] Almasi M M, Hemmati H, Fraser G, Arcuri A, Benefelds J. An industrial evaluation of unit test generation: Finding real faults in a financial application. In *Proc. the 39th IEEE/ACM International Conference on Software Engineering: Software Engineering in Practice Track* (*ICSE-SEIP*), May 2017, pp.263–272. DOI: 10.1109/ICSE-SEIP.2017.27.

[36] Daka E, Campos J, Fraser G, Dorn J, Weimer W. Modeling readability to improve unit tests. In *Proc. the 10th Joint Meeting on Foundations of Software Engineering*, Aug. 2015, pp.107–118. DOI: 10.1145/2786805.2786838.

[37] Panichella S, Panichella A, Beller M, Zaidman A, Gall H C. The impact of test case summaries on bug fixing performance: An empirical investigation. In *Proc. the 38th International Conference on Software Engineering*, May 2016, pp.547–558. DOI: 10.1145/2884781.2884847.

[38] Roy D, Zhang Z Y, Ma M, Arnaoudova V, Panichella A, Panichella S, Gonzalez D, Mirakhorli M. DeepTC-Enhancer: Improving the readability of automatically generated tests. In *Proc. the 35th IEEE/ACM International Conference on Automated Software Engineering* (*ASE*), Dec. 2020, pp.287–298. DOI: 10.1145/3324884.3416622.

[39] Zhang Y C, Mesbah A. Assertions are strongly correlated with test suite effectiveness. In *Proc. the 10th Joint Meeting on Foundations of Software Engineering*, Aug. 2015, pp.214–224. DOI: 10.1145/2786805.2786858.

[40] Watson C, Tufano M, Moran K, Bavota G, Poshyvanyk D. On learning meaningful assert statements for unit test cases. In *Proc. the 42nd ACM/IEEE International Conference on Software Engineering*, Jun. 2020, pp.1398–1409. DOI: 10.1145/3377811.3380429.

[41] Tufano M, Drain D, Svyatkovskiy A, Sundaresan N. Generating accurate assert statements for unit test cases using pretrained transformers. In *Proc. the 3rd ACM/IEEE International Conference on Automation of Software Test*, May 2022, pp.54–64. DOI: 10.1145/3524481.3527220.

[42] Cheon Y, Leavens G T. A simple and practical approach to unit testing: The JML and JUnit way. In *Proc. the 16th European Conference on Object-Oriented Programming*, Jun. 2002, pp.231–255. DOI: 10.1007/3-540-47993-7_10.

[43] Tillmann N, De Halleux J. Pex—White box test generation for .NET. In *Proc. the 2nd International Conference on Tests and Proofs*, Apr. 2008, pp.134–153. DOI: 10.1007/978-3-540-79124-9_10.

[44] Daka E, Fraser G. A survey on unit testing practices and problems. In *Proc. the 25th IEEE International Symposium on Software Reliability Engineering*, Nov. 2014, pp.201–211. DOI: 10.1109/ISSRE.2014.11.

[45] Jaygarl H, Lu K S, Chang C K. GenRed: A tool for generating and reducing object-oriented test cases. In *Proc. the 34th IEEE Annual Computer Software and Applications Conference*, Jul. 2010, pp.127–136. DOI: 10.1109/COMPSAC.2010.19.

[46] Cruciani E, Miranda B, Verdecchia R, Bertolino A. Scalable approaches for test suite reduction. In *Proc. the 41st IEEE/ACM International Conference on Software Engineering* (*ICSE*), May 2019, pp.419–429. DOI: 10.1109/ICSE.2019.00055.

[47] Chetouane N, Wotawa F, Felbinger H, Nica M. On using *k*-means clustering for test suite reduction. In *Proc. the 2020 IEEE International Conference on Software Testing, Verification and Validation Workshops* (*ICSTW*), Oct. 2020, pp.380–385. DOI: 10.1109/ICSTW50294.2020.00068.

[48] Mues M, Howar F. JDart: Portfolio solving, breadth-first search and SMT-Lib strings (competition contribution). In *Proc. the 27th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, Mar. 27–Apr. 1, 2021, pp.448–452. DOI: 10.1007/978-3-030-72013-1_30.

[49] Baluda M. EvoSE: Evolutionary symbolic execution. In *Proc. the 6th International Workshop on Automating Test Case Design, Selection and Evaluation*, Aug. 2015, pp.16–19. DOI: 10.1145/2804322.2804325.

[50] Olsthoorn M, Van Deursen A, Panichella A. Generating highly-structured input data by combining search-based testing and grammar-based fuzzing. In *Proc. the 35th IEEE/ACM International Conference on Automated Software Engineering*, Dec. 2020, pp.1224–1228. DOI: 10.1145/3324884.3418930.

Xiang-Jun Liu is currently pursuing her Master's degree with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nanjing University, Nanjing. Her research interests include software testing, cloud computing, and big data technology.

Ping Yu received her Ph.D. degree in computer science and technology in 2008 from Nanjing University, Nanjing. She is an associate professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nanjing University, Nanjing. Her research interests include intelligent software engineering, cloud computing, and big data technology.

Xiao-Xing Ma is currently a full professor with the State Key Laboratory for Novel Software Technology and Department of Computer Science and Technology at Nanjing University, Nanjing. From the same university, he received his Ph.D. degree in computer science in 2003. His research interests include self-adaptive software systems, software architectures, and quality assurance for machine learning models used as software components. He co-authored over 100 peer-reviewed papers and served in technical program committees of various international software engineering conferences.