Automated Testing for Java Programs using JPF-based Test Case Generation

Supasit Monpratarnchai, Shoichiro Fujiwara, Asako Katayama, Tadahiro Uehara
Software Innovation Laboratory, Software Technologies Laboratories, FUJITSU LABORATORIES LTD.
4-1-1 Kamikodanaka, Nakahara-ku, Kawasaki-shi, Kanagawa, 211-8588, Japan
Tel: +81-44-754-2675, Fax: +81-44-754-2570

Email: {m.supasit, fujiwara.sho, katayama.asako, uehara.tadahiro}@jp.fujitsu.com

ABSTRACT

Program testing requires a series of tasks such as preparing drivers and stubs, creating test cases, and executing unit tests. To reduce manual effort of performing such tasks for testing Java programs, we developed a tool that fully integrates and automates all of these processes, by using JPF with extensions as a symbolic execution engine for automatically generating unit test cases. In this paper, we present this tool and its application to real projects to evaluate its efficacy. The evaluation results demonstrate that the tool performs well in terms of the test time reduction compared with manual test as it eliminates the total amount of manual effort, while largely preserving a high coverage of greater than 90 % as our expected borderline.

Categories and Subject Descriptors

D.2.4 [**Software Engineering**]: Software/Program Verification;
D.2.5 [**Software Engineering**]: Testing and Debugging – *Testing tools (e.g. data generators, coverage testing)*

General Terms

Reliability, Experimentation, Standardization, Verification

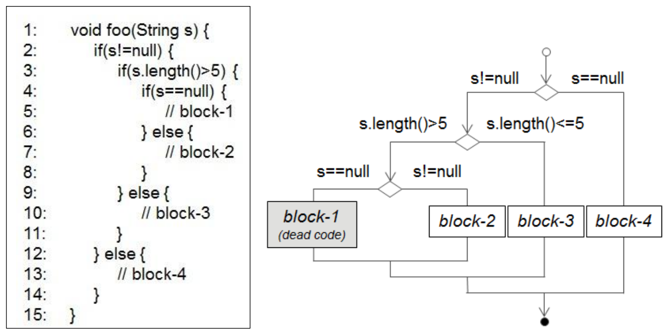
Keywords

Java, Program testing, JUnit, Symbolic execution, JPF, Driver, Stub, Test case, Generation, Automation, Coverage, Eclipse

1. INTRODUCTION

Among various techniques for software verification, this paper focuses on *program testing*, a technique of examining a program's code to check whether its behavior conforms to its specification. There also currently exist several approaches and technologies for program testing. A typical and practical technique is to actually execute the program under test (target program) with a set of unit test cases (test suite) using a unit testing tool or framework, like JUnit for Java programs [11].

For relatively small target programs, unit test cases can simply be written by hand. However, as the target program gets more complicated, manually preparing a high quality test suite becomes a hard, time-consuming, and infeasible task in practice. Recent years have seen the development of many techniques and tools for automatically generating unit test cases, from both researchers and industrial practitioners in the software testing community, such as in [9] and [10]. With a similar objective, we developed an automatic unit test case generation tool based on symbolic execution [1], by using JPF [12] with some extensions to support more complex data types such as Strings, often used in real industrial business applications [8]. To execute the tool we need to input some other programs called *drivers* and *stubs* as assistive devices of symbolic execution in addition to the target program. Currently we have to prepare drivers and stubs by hand, which is costly, as it requires knowledge and experience of the symbolic execution. To reduce the overall cost, we further extended our test case generation tool by automating all of its surrounding processes, such as driver and stub preparation, JUnit test execution, etc.



```
void foo(String s) {
    if(s!=null) {
        if(s.length()>5) {
            if(s==null) {
                // block-1
            } else {
                // block-2
            }
        } else {
            // block-3
        }
    } else {
        // block-4
    }
}
```

(a) (b)

Figure 1. Sample program code (a) and its control structure (b).

This paper presents the implementation details of the tool which include the following contributions, other than the automated tool we developed.

- Techniques for automatically generating drivers and stubs (e.g. a criteria for selecting methods to be stubbed, a procedure for generating stub variations),
- Techniques for automating all of test processes using our existing automatic unit test case generation tool as a core engine, and
- A report of the tool's application experiment on real projects for evaluating its effectiveness.

The techniques of deploying and extending JPF for automatically generating unit test cases are omitted in this paper. The details can be found in related publications [10, 8]. We expect that the proposed tool will automatically perform program testing for the testers without any human input. The evaluation results show that the tool eliminates the total amount of manual effort hence reduces the test time compared with the manual test from several minutes to just a few seconds, while preserving high test coverage of greater than 90 % for most cases.

2. SYMBOLIC EXECUTION

Symbolic execution is a formal semantic technique for program analysis in which symbols are used as inputs instead of concrete values. It exercises all executable paths (possible combinations of all conditional branches in a program), while comprehensively exploring the complete range of each of the symbolic variables using *constraint solvers*. In this section, we briefly explain the underlying mechanism of symbolic execution by using the sample program in Figure 1.

Here we symbolize parameter *s* as a symbolic variable. When this program is symbolically executed, a special *symbolic value* α is created and assigned to *s*. This variable and its expressions are tracked in a symbolic state with a *path constraint (pc)*, which is updated each time a branch condition is executed. Assuming the case that the condition s!=null is true and s.length()>5 is false, the path constraint of the executable path containing *block-3* will be updated in the following sequence.

1: initial state : *pc* = { }
2: when s!=null is executed : *pc* = { α!=null }
3: when s.length()>5 is executed : *pc* = { α!=null && α.length()≤5 }

Symbolic execution gradually explores all executable paths by tracking along the program control structure. Although there are 4 paths in this program, only 3 of them are executable; the paths containing *block-2*, *block-3*, and *block-4*. Note that *block-1* will never be reached due to a conflict between the condition s!=null (line 2) and s==null (line 4), thus the left–most path that contains *block-1* is non-executable. We denote a logically non-executable program segment like *block-1* as *dead code*.

3. JPF-BASED TEST CASE GENERATION

For relatively small target-programs, consisting of just a few simple conditional statements, we can manually create unit test cases by analyzing each condition in the program to explore all executable paths and choosing some valid values that satisfy the conditions on each path as test inputs. But real-world programs containing complicated business logic are usually large in both size and complexity. Manually preparing a high quality test suite that covers all invalid, unexpected inputs or call sequences for such programs becomes a difficult and time-consuming task with high workload.

In order to reduce manual effort of preparing unit test cases, we developed an automatic unit test case generation tool based on symbolic execution technique, by extending *Java PathFinder*[1] (JPF) [10]. Our tool extends and strengthens JPF to support more complex data types such as Strings [8], often used in real industrial business applications. The generation starts with performing symbolic execution to extract all program paths and obtain their corresponding path constraints. For each constraint, constraint solvers then check the constraint validity. If valid, the solvers generate valid values as a test input data for all variables which satisfy that constraint. For the sample program in Figure 1, symbolic execution extracts 4 paths with their corresponding path constraint, and generates test input data as shown in Table 1. Notice that no test input data is generated for the first path as it is non-executable path containing dead code of *block-1*.

Table 1. Path constraints obtained by symbolic execution.

	Path Constraint	Test Case	
		Test Input	**Expected Results**
1	α!=null && α.length()>5 && α==null	–	–
2	α!=null && α.length()>5	*s*=" " (6 whitespaces)	Execution results of *block-2*
3	α!=null && α.length()≤5	*s*=" " (5 whitespaces)	Execution results of *block-3*
4	α==null	*s*=null	Execution results of *block-4*

Generally we measure the quality of the generated test cases and the thoroughness of testing by the *test coverage*; the degree (percentage) to which the source code of the target program has been exercised (tested) by the test cases. Our unit test case generation tool reports two basic coverage criteria; *line coverage* (has each line or statement executed at least once?) and *branch coverage* (has each branch, such as in *if* statements or loops, executed at least once?). Basically we expect 100 % of test coverage, but for real applications the coverage varies due to various reasons such as dead code as in our example, for instance. Preliminary experiments indicate that our unit test case generation tool generates test cases with higher test coverage than other similar tools for most cases [8].

[1] JPF [7, 12] is an open-source JVM for verifying executable Java byte-code programs by means of symbolic execution, developed by the NASA Ames Research Center.

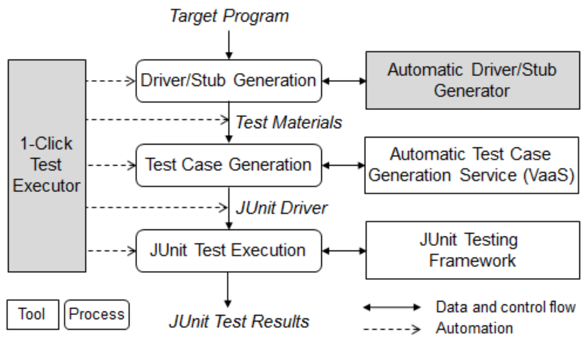


Figure 2. Implementation architecture of 1-click test executor.

To support more effective use and applicability of the tool and speed-up the tool execution, we further hosted it on the cloud as a web service for automatic test case generation. As such a delivery model is typically referred to as a *Testing-as-a-Service* (TaaS) [3], we gave a similar name *Validation-as-a-Service* (VaaS) [5, 6] to our service, to convey the meaning of on-demand validation. Users can remotely access to the VaaS service and simply perform test case generation via web browsers.

4. TEST AUTOMATION

4.1 Motivation and Approach

To run the automatic test case generation tool/service, we require some other programs called *drivers* and *stubs* as an execution input in addition to the target programs. A driver is a Java class that specifies symbolic variables and invokes the target method in the target program. A stub is a simplified implementation of other methods invoked by the target method which cannot be directly used due to any reasons, e.g. they are under implementation, under test, or under bug-fix. Although our tool/service automates the test case generation process, however, preparing the drivers and stubs still has to be done manually, which is time-consuming, labor-intensive, skill-required, and sometime error-prone task, so that it becomes a bottleneck in using the tool/service.

In order to reduce overall costs in the test cycle, we further extended our test case generation tool/service as another new tool, to cover the automation of all other remaining processes including driver/stub preparation and unit test execution. Next subsection gives the implementation details of this tool. We expect that the tool will reduce test cost by eliminating human input and decreasing manual operation time, while preserving a high level of test coverage. Since all of the processes are performed by the tool, users do not need to know the content of the target programs, drivers, and stubs. Thus, we also expect that testers, developers, or any other users, with no prior knowledge of symbolic execution or JPF, can use our tool to simply perform program testing in a black-box manner.

4.2 Implementation

We first implemented the *automatic driver/stub generator*, an engine that automatically creates corresponding drivers and stubs for a particular target program. Then, we developed a tool that automates all test processes, by 1) using this automatic driver/stub generator to generate driver and stubs, 2) remotely executing the automatic test case generation service (VaaS) on cloud to generate JUnit test cases, and 3) invoking JUnit testing framework [11] to run the unit test with the generated test cases. These processes are connected altogether in sequence, so that we can activate and run the tool with just a single click. We gave our tool a specific name of *1-click test executor*, to represent its means of execution which requires minimum human input. Figure 2 depicts the implementation architecture of the 1-click test executor. Notice that it does not automate only the above three main processes, but also the connection and data/control flow between each of them. This subsection presents the implementation details of the tools, which are divided into two parts; the implementation of the automatic

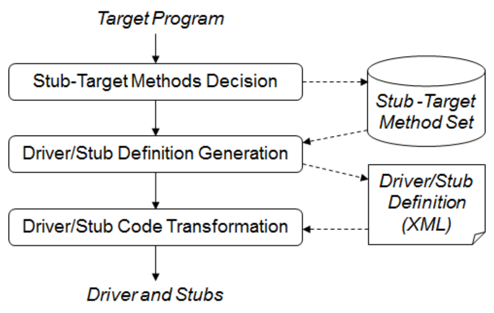

Target Program
Stub-Target Methods Decision
Stub-Target Method Set
Driver/Stub Definition Generation
Driver/Stub Definition (XML)
Driver/Stub Code Transformation
Driver and Stubs

Figure 3. Driver and stub generation processes.

```
class A {
   protected void ma() { ... }
}
class B extends A {
   public void mb1(int i, String s) {
      ma( );    mb2( );
      Object obj = C.mc(i);
      s = s.concat("A");
   }
   private mb2( ) { ... }
}
class C {
   public static Object mc(int j) {  D.md( );  }
}
class D {
   public static void md( ) { ... }
}
```

Figure 4. Sample Java class structure.

driver/stub generator and the implementation of the 1-click test executor, as shown with two gray boxes in Figure 2.

4.2.1 *Automatic Driver/Stub Generator*

We implemented the automatic driver/stub generator with three sequential processes as shown in Figure 3. Firstly, the generator decides which methods should be stubbed and collects them into a *stub-target method set* (a set of methods to be stubbed). The generator then analyzes the target method, extracts driver information, such as which variables should be symbolized, and stores them as an XML document called a *driver/stub definition*. Similarly, the generator analyzes each stub-target method in the stub-target method set and records its signature by appending it to the driver/stub definition. And lastly, the generator transforms the definition into an executable Java code.

4.2.1.1 *Stub-target methods decision*

Although there is no standard technique for deciding which methods should be stubbed to obtain the best coverage, we simply stipulate the stub-target methods to be a certain set of all non-Java-library-class methods directly invoked from the target method. Since a class usually represents a single concern and all methods within the class collectively implement its associated functions, these methods should be tested as well as the target method. Hence, we stub only methods that *do not* belong to the target class (class of the target-method), while retaining those of the target class (or its parent classes) un-stubbed. Consider the Java program in Figure 4 for example. Given method *mb1* as the target method, among method *ma*, *mb2*, *mc*, and *concat* which are directly invoked from *mb1*, we stub only method *mc* according to the above criteria. Notice that because these criteria consider only methods that are *directly* invoked from the target method, method *md* is out of the scope of the decision and is ignored when method *mc* is stubbed. However, in general, depending on the program design, the business logic to be tested maybe placed outside the target class (it should be tested but it is stubbed), or some other methods invoked from the target method may be irrelevant to the logic under test (it should be stubbed but it is not). Due to these reasons, the above selection criteria does not guarantee the best coverage.

```
<Driver>
  <TargetClass pkg="sample" name="B">
    <TargetMethod name="mb1" sig="(int,String)void">
      <ParamList>
        <Param name="i" type="int" status="SYMBOLIC"/>
        <Param name="s" type="String" status="SYMBOLIC"/>
      </ParamList>
    </TargetMethod></TargetClass></Driver>
<Stub>
  <StubClass pkg="sample" name="C">
    <StubMethod name="mc" signature="(int)Object">
      <ParamList>
        <Param name="j" type="int"/>
      </ParamList>
    </StubMethod></StubClass></Stub>
```

Figure 5. Sample driver/stub definition in XML.

```
1:   public class DriverForB {    // driver
2:      @Symbolic("true")
3:      public static int i0 = 0;
4:      @Symbolic("true")
5:      public static String s0 = "";
6:      @Symbolic("true")
7:      public static boolean obj_NULL = false;
8:      public static void main(String[ ] args) {
9:         C.setObj_NULL(obj_NULL);
10:        B b = new B( );
11:        b.mb1(i0, s0);
12:     }
13:  }
14:  public class C {    // stub
15:     public static boolean obj_NULL = false;
16:     public static setObj_NULL(boolean b) { obj_NULL=b; }
17:     public static Object mc(int j) {
18:        if(obj_NULL) return null;
19:        else return new Object( );
20:     }
21:  }
```

Figure 6. Sample driver and stub code.

4.2.1.2 *Driver/stub definition generation*

Given the target method, the generator analyzes its containing class's structure, lists all variables, specifies symbolic variables, and records all this information as the driver definition. Generally we symbolize two kinds of variables as symbolic: a) the target method parameters, and b) fields of the target class which are used in conditional statements in the target method. Similarly, the generator simply extracts and records the signature of each stub-target method as the stub definition. XML code in Figure 5 is a sample of (simplified) driver/stub definition for the target method *mb1* and the stub-target method *mc*.

4.2.1.3 *Driver/stub code transformation*

From the driver definition, the generator generates driver code as a Java class. The driver class contains symbolic variables declared as class members annotated with *@symbolic*, such as those shown in line 2–5, and a *main* method that invokes the target method, with symbolic variables passed as arguments, as shown in line 11, of the sample driver and stub code in Figure 6.

By specifying a particular variable as a symbolic variable, JPF generates its variation (a set of its values) as test input data. Since JPF supports only primitive types, for non-primitive types we have to add some other primitive *symbolic* variables as dummy variables to assist generating the variation of the original (non-primitive) variables. In this paper, we give the details and examples for two often used non-primitive types: object and list.

<u>Object type</u>: The generator adds a dummy variable of type boolean and a logic for generating its variation of null and an instance whose members are initialized with their own symbolic variables. Figure 7 (a) shows a sample driver code that symbolizes a variable *p* of type *Person* containing a field *name* of type String.

<u>List type</u>: In addition to a dummy variable of type boolean for generating a variation of null and list instance, the generator adds another dummy variable of type *int* to specify the size of the list. Figure 7 (b) shows an example of a driver code that symbolizes variable *list* of

```
public static Person p;
@Symbolic("true")
public static boolean p_NULL = false;
@Symbolic("true")
public static String name0 = "";
...

public static void main(String[ ] args) {
   if(p_NULL) {
        p = null;
   } else {
        p = new Person(name0);
   }
   ...
   // target method invoked here
   ...
}
```

(a)

```
public static ArrayList<String> list;
@Symbolic("true")
public static boolean list_NULL = false;
@Symbolic("true")
public static int list_NUM = 0;
@Symbolic("true")
public static String s0 = "";
...
public static void main(String[ ] args) {
   if(list_NULL) { list = null; }
   else {
      list = new ArrayList<String>( );
      switch(list_NUM)
         case 0: break;
         case 1: list.add(s0); break;
   }
   // target method invoked here
   ...
}
```

(b)

Figure 7. Sample driver for (a) object and (b) list type.

type ArrayList of String. We can expand the size of the list by adding more *case* statements to the variation generation logic. In our current implementation, the size of the list is fixed to 3.

For each stub-target class (class of stub-target methods), the generator creates stub code as a Java class that contains its stub-target methods with dummy *non-symbolic* variables and their setter method such as those shown in line 15–16 of the sample stub code in Figure 6. Since these dummy variables are used to generate a variation of the methods' return value, the generator puts their logic for generating the variation in the stub-target method body as shown in line 18–19. The rules for generating these dummy variables and their variation generation logic are the same as those for the driver as described above. To make these dummy variables able to be exercised by JPF, the generator symbolizes them in the driver as additional symbolic variables, such as those shown in line 6–7, and initializes them through their setter method, as shown in line 9.

4.2.2 1-Click Test Executor

In order to automate all processes of program testing that uses the automatic test case generation service (VaaS) as a core engine, we developed a new automated test tool called *1-click test executor* as an Eclipse plug-in. With this executor, users select a particular method as the target method, a particular class to specify all methods in the class as the target methods, or a particular package to specify all methods in all classes in the package as the target methods, and then start the test by just clicking the execution command. When a target item other than a method is selected, all methods in the target item are treated as target methods, and the executor performs the test on each method in sequence. This eliminates the manual work of executing the tool on each method in separate.

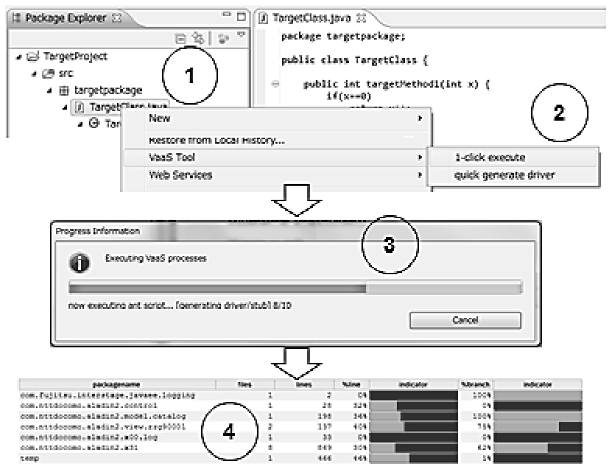
The 1-click test executor consists of 5 main processes, each one corresponding to each dotted arrow in the implementation architecture in Figure 2. We implemented each process as a single or a set of sequential independent *Apache Ant* task(s) [13]. When the execution command is clicked, the executor instantly activates and runs all Ant tasks one after another consecutively in the following order.

- **Process1**: *Driver/stub generation* – consists of 3 Ant tasks.

1) Compile the target project (i.e. the project that contains the specified target method/class/package) into binary, since JPF analyzes the target program in byte-code.
2) Run the automatic driver/stub generator to generate drivers and stubs into the target project.
3) Re-compile the target project to compile the generated drivers and stubs.

- **Process2**: *Test-materials packing* – consists of a single Ant task.

4) Group and pack the target programs (target classes and relevant library classes) with theirs corresponding drivers and stubs as a *test-materials* (.zip), by referring to the driver/stub definition (driver and all stubs contained in the same definition are grouped together).



packagename	files	lines	%line	indicator	%branch	indicator
com.fujitsu.interstage.javaee.logging	1	2	0%		100%	
com.nttdocomo.aladin2.control	1	28	32%		0%	
com.nttdocomo.aladin2.model.catalog	1	198	34%		100%	
com.nttdocomo.aladin2.view.[illegible]	2	137	40%		75%	
com.nttdocomo.aladin2.a00.log	1	33	0%		0%	
com.nttdocomo.aladin2.a31	8	449	30%		62%	
temp	1	444	44%		1%	

Figure 8. Sample usage snapshot of the 1-click test executor.

- **Process3**: *Test case generation* – consists of 3 Ant tasks.

5) Remotely connect to the automatic test case generation service (VaaS) with user-specified access data.
6) Upload the test-materials and run the service.
7) Monitor the service execution state. Once the execution has completed, download the results (.zip), which contain JUnit drivers, coverage reports, etc., into the target project.

- **Process4**: *JUnit pre-processing* – consists of 2 Ant tasks.

8) Unzip the execution results and extract JUnit drivers into the source folder of the target project.
9) Compile the JUnit drivers.

- **Process5**: *JUnit test execution* – consists of only one Ant task.

10) Execute the JUnit drivers under the JUnit testing framework.

5. USAGE AND EVALUATION

Figure 8 demonstrates a sample usage scenario of our 1-click test executor. In this scenario, a user selects a class as the target program by right-clicking its Java file in the package explorer (1), and starts the test by clicking the *1-click execute* command in the popped-up context menu (2). The execution status is reported on a progress bar (3). The majority of the execution time is spent on test case generation (the VaaS service execution), which possibly varies from a few seconds to many hours depending on the size and the complexity of the target program. Once the execution has finished, JUnit test results and coverage report are output to the console (4).

We evaluated the effectiveness of our 1-click test executor by showing that it reduces test time compared with manual test, while preserving high test coverage. For this purpose, we conducted an experiment to compare the time required for performing the test by hand with using the test executor, and also to measure the coverage. We use a real Java project, implementing an information management support system, as the target. The project consists of several library classes and 25 target methods of the size range from 10 to 150 LoC. The experimental results show that the required time is significantly shortened from several minutes (15.12 minutes in average) in case of manual test, to just a few seconds (11.68 seconds in average) with the 1-click test executor.

The histogram in Figure 9 shows a distribution of the test coverage. Among 25 target methods, 15 methods achieve the coverage of 100 %, 6 methods achieve the coverage of more than 90 %, while the remaining falls below 90 % of coverage. This indicates that with the executor, a certain degree of high coverage (greater than 90 % as our expected borderline) is guaranteed for most cases (84 % of the target methods). By analyzing the source code, we found that the major reason why another 16 % of the target methods could not achieve 100 % test

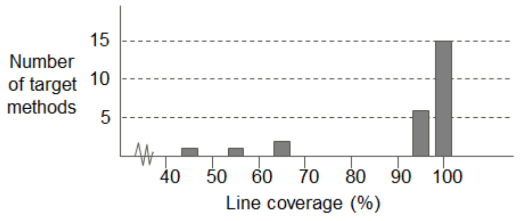


Figure 9. Distribution of the test coverage.

coverage is that they contained dead code. In general, even when the target program does not contain dead code, its coverage can be lower than 100 %. For this case, we can improve the coverage by optimizing the generated stubs or selecting other stub-target methods more suitable than those fixed by the executor.

6. RELATED WORK

Currently there are a variety of automated test tools for Java programs, available as both open-source and commercial software. Jtest [2] and AgitarOne [4] are in our scope of interest as they are similar to our tool in that they are implemented as an Eclipse plug-in, and they automate the unit tests of Java programs in any level (individual methods, classes, or larger units). AgitarOne uses an *agitation technique* to automatically generate JUnit test cases with the test coverage of more than 80 % [4]. Although its test processes is not fully automated in that testers have to run the tests manually after the test cases are generated, it provides several useful support features such as dashboard reports of the test feedback, advice for improving the test coverage, etc. While AgitarOne is semi-automated, Jtest fully automates all of the test processes like our 1-click test executor does. It generates JUnit test cases, executes JUnit test with the coverage analysis, and outputs custom reports in HTML/XML. Unlike our tool which has to be run only on Eclipse IDE via a plug-in (Eclipse-dependent), Jtest is available as both Eclipse plug-in and as a cloud service. In the future work, we intend to make our tool more flexible to be used as an isolated independent software product, and also as a *Cloud-based Testing-as-a-Service* (*CTaaS*) [3] similar to Jtest, in order to expand the usage scope and applicability of the tool.

7. CONCLUSION

In this paper, we present the implementation techniques of a tool called *1-click test executor*, which fully automates all processes necessary for Java program testing, which includes driver/stub generation, test case generation, and unit test execution. This tool uses JPF-extended symbolic execution engine for unit test case generation. With the tool, any user can simply perform testing by running the tool with only one click, on the selected target programs which can be either individual method, class, or package. The experimental results show that our tool performs well in terms of the time reduction compared with manual test as it significantly reduces the manual test time for high-coverage testing, hence improves the real-world applicability of JPF-based test case generation. As future work, we plan to extend the tool to support the automatic testing of the repositories on revision control systems such as SVN, so that the tests are automatically triggered and performed every time the source code in the repository is updated.

8. ACKNOWLEDGMENTS

Our thanks to Mukul R. Prasad, a senior researcher of the Trusted Systems Innovation Group at Fujitsu Laboratories of America (FLA), for thorough content-check and proof-reading of this paper.

9. REFERENCES

[1] King, J. C. 1976. Symbolic execution and program testing. In *Communications of the ACM* 19, 7 (Jul. 1976), 385–394.

[2] Parasoft Corporation 2007. Automatic Java software and component testing: using Jtest to automate unit testing and coding standard enforcement. From http://www.parasoft.com/jsp/products/article.jsp?articleId=839 (Jan. 2007).

[3] Gao, J., Manjula, K., Roopa, P., Sumalatha, E., Bai, X., Tsai, W. T., and Uehara, T. 2012. A cloud-based TaaS infrastructure with tools for SaaS validation, performance and scalability evaluation. In *Proceedings of the 4th International Conference on Cloud Computing Technology and Science* (Taipei, Taiwan, December 03-06, 2012). CloudCom'12. IEEE, Washington, DC, 464–471.

[4] Agitar Technologies, Inc. 2009. *ArgitarOne junit generator – prevent regressions and cut the maintenance cost of your Java applications*. From http://www.agitar.com/pdf/AgitarOneJUnitGeneratorDatasheet.pdf (2009).

[5] Fujitsu Laboratories Ltd. 2008. *Fujitsu develops software verification technology for practical-use web applications*. From http://www.fujitsu.com/global/news/pr/archives/month/2008/20080404-02.html (Apr. 2008).

[6] Fujitsu Laboratories of Europe Ltd. 2011. Validation-as-a-service – advancing Java testing. In *Brochure of 2011 Fujitsu Technology Forum (Fall)* (Oct. 2011).

[7] Anand, S., Păsăreanu, C. S., and Visser, W. 2007. Jpf–se: a symbolic execution extension to Java PathFinder. In *Proceedings of the 13th International Conference on Tools and Algorithms for the Construction and Analysis of Systems* (Braga, Portugal, March 24 – April 1, 2007). TACAS'07. ACM, New York, NY, 134–138.

[8] Ghosh, I., Shafiei, N., Li, G., and Chiang, W. 2013. JST: an automatic test generation tool for industrial Java applications with strings. In *Proceedings of the 35th International Conference on Software Engineering* (San Francisco, CA, May 18-26, 2013). ICSE'13. ACM, New York, NY, 992–1001.

[9] Visser, W., Păsăreanu, C. S., and Khurshid, S. 2004. Test input generation with Java PathFinder. In *Proceedings of the International Symposium on Software Testing and Analysis* (Boston, Massachusetts, July 11-14, 2004). ISSTA'04, ACM, New York, NY, 97–107.

[10] Ginbayashi, J., Uehara, T., Munakata, K., and Yabuta, K. 2010. New approach to application software quality verification. In *FUJITSU Sciences and Technology Journal* 46, 2 (Apr. 2010), 158–167.

[11] JUnit: Programmer–Oriented Testing Framework for Java, http://junit.org/.

[12] Java PathFinder, http://babelfish.arc.nasa.gov/trac/jpf.

[13] Apache Ant™, http://ant.apache.org/.