ExecutionFlow: a tool to compute test paths of Java methods and constructors

William Niemiec, Keslley Silva, Érika Cota
PPGC - Informatics Institute - Federal University of Rio Grande do Sul
Porto Alegre, Brazil
williamniemiec@hotmail.com,{keslley.silva,erika}@inf.ufrgs.br

ABSTRACT

Calculating the code coverage of a test suite depends on the identification of the executed test paths. Available code coverage tools are limited to a few basic tracing strategies and do not provide this information as proper test paths, precluding its use in further analyses. We present ExecutionFlow, a solution to trace executed test paths in Java code. The tool is based on the use of a debugger, aspect-oriented programming, and a testing framework. Through empirical validation we show that it is possible to obtain the test path from several methods and constructors, including high complexity ones. Video demonstration: https://youtu.be/2W5BjVuc6fw

CCS CONCEPTS

• **Software and its engineering** → **Software creation and management**; **Software maintenance tools**.

KEYWORDS

Software testing, Testing tool, Coverage criteria, Test path generation, Aspect-oriented programming, Tracing tool

ACM Reference Format:
William Niemiec, Keslley Silva, Érika Cota. 2021. ExecutionFlow: a tool to compute test paths of Java methods and constructors. In *Brazilian Symposium on Software Engineering (SBES '21), September 27-October 1, 2021, Joinville, Brazil.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3474624.3476014

1 INTRODUCTION

A good test suite is the core of an effective test and a strong-power suite can detect more bugs than a weak-power one [7]. The first requirement for a strong-power suite is its capacity to reach and exercise the faulty code. Code coverage criteria have been proposed to be an estimator of this capacity.

Test code coverage criteria are based on the notion of test requirements (TR) and test paths. A TR is a particular software element that a test case must satisfy or cover. A test path (TP) is an intricate use that a test case has exercised in a source code. During unit testing, both TR and TP are extracted from the control flow graph (CFG) of a program and are expressed as sub-paths in the graph. TRs are extracted from a static analysis of the CFG whereas TPs must be extracted from the actual execution of the software for a given test suite. TRs are extracted from the CFG according to a given test criterion such as node, branch, or path coverage. The criterion indicates which elements of the CFG should be explored during test. The number of TRs that are sub-paths of the TPs extracted during test execution, i.e., the number of TRs covered by the TPs, defines the test coverage achieved by that suite for a given criterion.

Currently, test paths are identified as part of the code coverage analysis performed by some tools such as Cobertura and JaCoCo. However, those tools provide the coverage analysis for few basic test criteria (typically node and branch) and visually show, in the code, the statements that are executed or not, without providing this information as proper test paths (sequence of nodes in the corresponding CFG). Therefore, one cannot use the results of a test campaign to explore additional criteria or look for redundant tests.

We propose a tool called ExecutionFlow[1] that aims to generate the test paths of a project at the method level in a suitable format to be manipulated and used for different analysis. Our solution relies on the use of aspect-oriented programming (AOP) to collect information concerning the methods and constructors covered by one or more test methods, a debugger for inspecting the code and a test framework for executing the test methods. In this paper, we detail the solution tailored for Java projects and present the results of the empirical validation performed for the proposed tool.

This paper presents the following contributions:

- We propose a strategy to compute the test paths at the method level for a given test suite and provide this information in a suitable format for further use;
- we present a tool based on the proposed strategy that can generate test paths for Java projects;
- we perform an empirical study involving 12 open-source projects to validate our tool in practice.

The paper is structure as follows: Section 2 presents the overall strategy for test path generation using AOP, a debugger and a unit testing framework. Section 3 details the tool developed for Java platform, giving essential knowledge concerning the implementation and use of ExecutionFlow. Sections 4, 5 and 6 describe, respectively, the validation strategy, the experimental results and threats to validity. Section 7 discusses the related work and Section 8 concludes the paper.

2 TOOL DESIGN

The proposed approach focus on unit tests and aims at identifying the paths executed by application methods during the execution

[1]https://github.com/williamniemiec/ExecutionFlow

ACM ISBN 978-1-4503-9061-3/21/09…$15.00
https://doi.org/10.1145/3474624.3476014

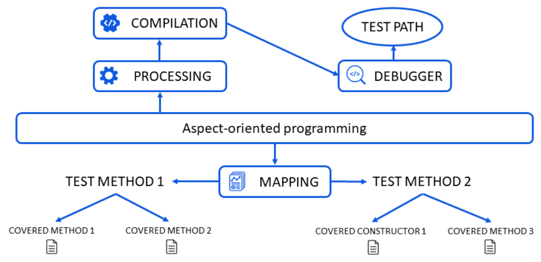
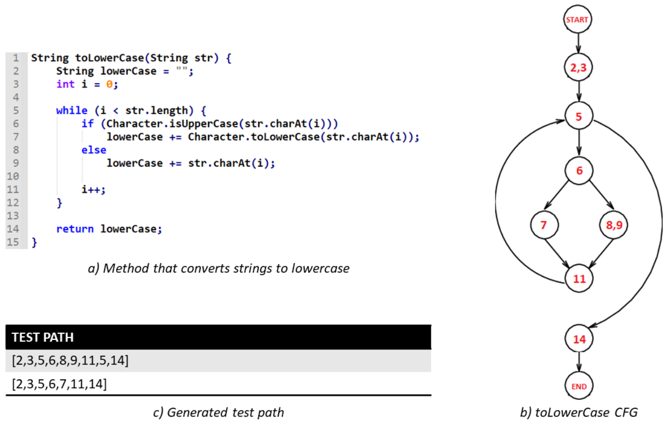


Figure 1: Tool architecture



```
1   String toLowerCase(String str) {
2       String lowerCase = "";
3       int i = 0;
4
5       while (i < str.length) {
6           if (Character.isUpperCase(str.charAt(i)))
7               lowerCase += Character.toLowerCase(str.charAt(i));
8           else
9               lowerCase += str.charAt(i);
10
11          i++;
12      }
13
14      return lowerCase;
15  }
```

a) Method that converts strings to lowercase

TEST PATH
[2,3,5,6,8,9,11,5,14]
[2,3,5,6,7,11,14]

c) Generated test path

b) toLowerCase CFG

Figure 2: Example of a code (a), its corresponding CFG (b) and some test paths (c)

of a test suite. Each time an application method is called by a test method, the path executed within that method is collected.

For the test paths to be used for different purposes, it is important they are expressed not only in terms of what the developer sees in the source code, but also in accordance to the corresponding CFG. We consider the CFG of an application method is built using each executable statement as a node and the test paths must be extracted accordingly, as shown in Figure 2. Thus, the goal of our tool is to generate the test paths as described in Figure 2(c), receiving as input the source code of the test and application methods.

For a given test suite, test methods are processed one by one according to the following strategy: 1) identification of which application methods and constructors - called methods/constructors under test (M/CUT) hereafter - are executed by one test; 2) collection of information and preparation of the M/CUTs for the correct identification of the paths executed during test; and 3) test execution and test path extraction. We use AOP to achieve Steps 1 and 2. For Step 3, we use a unit testing framework and a debugger. This strategy is detailed below.

2.1 Mapping and processing of M/CUTs using AOP

Aspect refers to the modularization of a concern that cuts across multiple classes [4]. The identification and processing (collection of information and preparation) of the M/CUTs is the aspect modeled

```
1   public aspect MethodHook {
2
3     private pointcut insideTestedMethod():
4      && withincode(@org.junit.Test * *.*())
5      && !get(* *.*)
6      && !set(* *.*);
7
8     before(): insideTestedMethod() {
9      identifyMethodSignature(thisJoinPoint);
10     findMethodFile(thisJoinPoint);
11     registerMethodInfo(thisJoinPoint);
12    }
13    [...]
14  }
```

Figure 3: Extract from the source code that defines the AOP-based metric collection

in the proposed approach. Join point is a point of a program, such as the execution of a method, that will be monitored during runtime and connected to the instrumentation code implemented as the aspect. The definition of a join point is implemented using the pointcut concept. Pointcut is a predicate that defines join points (for example, the name of a method or a specific command). In the proposed approach, every method call made from within test methods must be monitored. Thus, a pointcut must be defined wherever there is a test method in the source files. For instance, this is implemented in lines 4-6 of Figure 3 by referring to any method that is not a getter or a setter and the is called from a test method. Advice is an action taken by an aspect at a particular join point. For each declared pointcut, one can define advices that are bounded to the monitored code during execution, i.e., runs at any join point matched by the pointcut. Different types of advice include "before" (action is taken just before the execution of a joint point), "after" (action taken immediately after the execution of a join point), and "around" (action is taken before and after a join point) advice. In the example of Figure 3, we have one advice that is executed before the join point. The code between lines 9-11 is executed just before the execution of each method that is defined in the pointcut "insideTestedMethod()", i.e., before each method called by a test method. In this example, we first get the method signature (line 9) and then we search for the location of the source code of the application method (line 10) and finally we register this information for later use (line 11). By using AOP, if additional tests are added to the suite they will be processed automatically.

In the proposed approach, we use the advices to perform most of the computation required to collect the test paths. This computation involves collecting information about the M/CUTs, adapting the code of the test methods and of the M/CUTs to allow the correct tracing, and controlling part of the path extraction process.

The first advice collects the following information for each M/CUT called from a given test:

- location of source and executable files
- method signature
- call site (the number of the line in the test method where the call to M/CUT appears)

A second advice performs the following actions when the execution reaches a test method:

(1) we need to modify the original test class to ensure the advices are executed over the correct test method. For the first advice to be processed and M/CUTs can be identified in the test method, one must run the testing framework for a given test class. Then, the information collected is used by a second advice to prepare this same test code (before the actual execution of the test) for the test path extraction. This means the testing framework has to be called a second time for the second advice to be executed and we must ensure that the second call to the testing framework process the same test method. Thus, after collecting the information about the M/CUTS, the first advice comments out all other test methods in that class, leaving only the current test method available for execution, and then calls the testing framework again.
(2) one needs to ensure that all arguments in method calls are on the same line, to allow the correct identification of the executed path in terms of line numbers in a later step of the process (Section 2.2).
(3) preparation of each M/CUT for the path extraction step that will be executed later. During compilation the basic structure of the code changes and some lines and/or structures may be omitted in the compiled code. For example, the if-then-else structure will be converted to if's and goto's. There is no else statement. Consequently, the line containing the 'else' will be omitted. The same applies to the statements like 'try-catch-finally', 'continue', 'break', 'do-while', 'switch' and declarations of uninitialized variables. The test path will be obtained using the compiled code, but it must be relatable to the original source code. Thus, we must adjust the original code by adding dummy instructions so as to keep the original behavior as well as the number of the lines associated to each statement that represents a node in the CFG. By doing this we make sure that the code executed in the path extraction phase (Section 2.2) corresponds exactly to the code seen by the developer. The added instructions can be any one that does not change the code behavior and is not suppressed by the compiler.
(4) compilation of the files with the M/CUTs instrumented as described above.
(5) execution of the test method and extraction of the test paths using a debugger, as explained in Section 2.2.

2.2 Test path extraction using a Debugger

The last action defined as an advice for when a test method is executed is responsible for the actual test path extraction and this is performed through the use of a debugger. Debuggers are programs that allow us to follow the execution of another program one line at a time and inspect its state during this process. An useful functionality of a debugger is the possibility of stopping at certain points of the source code (called breakpoints) and, from that point on, take specific actions such as step into a sentence (to carefully inspect its code), step over a sentence (execute without inspecting its contents), or simply continue (resume execution until the next breakpoint). To extract the test paths, we add a breakpoint in each M/CUT called from a test method and, when the debugger passes through these breakpoints, we step into those methods and constructors and register the number of all lines that are executed. In more detail, the following actions are performed in this step of the process:

(1) Place a breakpoint on the call to the M/CUT identified in the test method. This breakpoint is placed on the line where the method or constructor is invoked.
(2) Run the debugger.
(3) Upon reaching the breakpoint, run the 'Step into' command and, while in the method, run the 'Step over' command. For each 'Step over' command, the line being executed is registered. When returning from the method, the 'Continue' command is run. If it reaches the same breakpoint again (method is being called in an iteration), repeat the same procedure. Otherwise, execution is finished and another test method can be processed.

2.3 Test execution

Having defined the strategy to extract the test paths, all we need to do is to define how to put it in motion. Since our AOP-based approach is implemented around the test methods, we need to execute the test suite. By doing so, all the actions described in the previous sections will be executed whenever a test method is invoked. Thus, we simply use a unit testing framework to start and control the whole process.

3 TOOL IMPLEMENTATION

In this section we present ExecutionFlow, the implementation of the approach described in Section 2 for the Java platform. Besides the large use and tool support, we chose this platform because Java community is used to unit testing practices and our tool can provide additional support to help developers in implementing effective tests.

As mentioned, our approach is based on the use of AOP, a debugger and a unit testing framework. In our implementation, we use, respectively, AspectJ[2], JDB[3], and JUnit[4]. Next we detail the main implementation decisions of ExecutionFlow.

3.1 Implementation of the aspects

According to the proposed approach, the bulk of the implementation is the definition of the pointcuts (identification of test methods in the source files) and advices (codification of the actions that must be taken at each join point at runtime). In ExecutionFlow, we take advantage of JUnit notation and define a pointcut wherever there is the annotation *@Test* in the source files, as the example shown in Figure 3. The aspects are implemented in three classes, each one implementing the pointcuts and advices related to, respectively, application methods, constructors, and test methods. For M/CUTs, "before" advices are used to collect information that is later used to set up the debugger. For test methods, "around" advices are used to collect information before and after the test execution.

[2]https://www.eclipse.org/aspectj
[3]https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jdb.html
[4]https://junit.org/

Through the advices we perform some adaptations in the test and application methods. Besides the generic adaptations mentioned in Section 2 to keep the test path information relatable to the original code, implementation-specific modifications are required, as detailed next.

3.1.1 Adaptation of test methods. First, we have to convert any JUnit5 tests to JUnit4 notation due to incompatibilities between AspectJ and the JUnit5 API. In addition, to automate the whole process, we have to ensure that the processing is not stopped even if a test fails. Although not recommended, in practice we find many test methods with multiple asserts. In this case, when one assert fails during execution, the execution of that test method is interrupted, precluding the extraction of the remaining test paths. One can deal with this situation in two ways: either assuming that all tests are successful, or making sure that the processing continues even in case of a failed assertion. To avoid limiting the usage scope of ExecutionFlow we choose to leave this decision to the user. Unless explicitly required by the user, the path corresponding to a failing assertion is not collected. When the assertion fails, an exception is thrown and the information collected so far is lost. When the user indicates otherwise (test paths of failing assertions must be kept), the advice includes try-catch structures around all assertions of the test method being processed, allowing the processing of the information collected so far (in the catch block).

3.1.2 Adaptation of M/CUTs. As for the dummy instructions that must be added to the code as explained before, in this implementation we add a dummy assignment statement (like 'int d41d8c = 0') that does not affect the system behavior. To avoid any conflict with identifiers of the original code, we use the MD5 encoding [6] of the timestamp in the added statement.

After all modifications, the file is compiled and the debugger is executed over this modified file. Once the execution of the debugger is finished, the original files of the test and M/CUTs are restored.

3.2 Execution environment and flow

ExecutionFlow is built to be executed from Eclipse[5] v2020-03, due to its large dissemination, native support to JUnit, and support to AspectJ.

In summary, to use ExecutionFlow the developer must:

(1) Import the project to Eclipse and convert it to an AspectJ project;
(2) Add the file 'aspectjtools.jar'[6] to the build path;
(3) Add the jar of ExecutionFlow to the AspectJ build path;
(4) Run a test (or a test package or all tests) using JUnit.

The extracted test paths can be presented in the console or stored in a folder called 'results' that is created in the project´s root folder.

When JUnit is started, the AOP advice related to test methods is executed and the test method is processed, modified and compiled as explained. Then, JUnit is automatically called again. At this point, advices related to M/CUTs are executed and the corresponding application files are processed, modified, compiled and submitted to the debugger. During the execution of the debugger, executed paths are registered in memory. After all M/CUTs of a given test method are processed, all original files are restored, the test paths are exported to a file, and the test method is marked as processed. JUnit can then resume control and start processing the next test method.

[5]https://www.eclipse.org
[6]https://mvnrepository.com/artifact/org.aspectj/aspectjtools

4 VALIDATION METHODOLOGY

Our goal is to provide a tool that is compatible with any Java project. To achieve this purpose, the processing performed by the application in the source files (detailed in Section 2.1) must work with any valid source-code (successfully compiled). This is one of the central challenges in this work, essentially because of two factors: 1) the processing has to be compatible with code written in several different ways and 2) most of the time, discrepancies are detected only at runtime. Another challenge for this type of tool is the processing of high complex code, where the number of paths can be extremely high. Thus, besides the typical verification strategies implemented during implementation, we performed an empirical validation experiment to evaluate our tool and ensure its reliability. To this end, we evaluate the following research questions (RQs):

> ***RQ1:*** *Can the implemented tool extract the correct paths for Java codes implemented using distinct implementation styles?*
>
> ***RQ2:*** *Is it possible to obtain the test paths of methods and constructors with high cyclomatic complexity?*

To carry out the experiments, we selected 12 open-source Java projects of different sizes and domains: Apache Commons Lang[7], JFreeChart[8], Apache Commons Text[9], Apache Commons IO[10], Urban Airship Java Library[11], Biojava[12], Checkstyle, Apache Commons Collection[13], Apache Maven[14], Jscribejava[15], Dubbo[16] and Apache Commons Math[17]. Regarding each selected project, Apache Commons Lang extends the Java Util framework. JFreeChart allows creating a wide variety of interactive and non-interactive graphics. Apache Commons Text is a library focused on string manipulation algorithms. Apache Commons IO is a file utilities library. Urban Airship Java Library is a framework for messaging, content personalization and audience management. Biojava is an open-source project dedicated to providing a Java library for processing biological data. Checkstyle is a development tool to help programmers write Java code that adheres to a coding standard. Apache Commons Collections is a library that extends the Java Collections framework. Apache Maven is a software project management and comprehension tool. Jscribejava is an OAuth library for Java. Dubbo is a high-performance framework for remote procedure calls. Finally, Apache Commons Math is a library containing mathematics and statistics components.

[7]https://github.com/apache/commons-lang
[8]https://github.com/jfree/jfreechart
[9]https://github.com/apache/commons-text
[10]https://github.com/apache/commons-io
[11]https://github.com/urbanairship/java-library
[12]https://github.com/biojava/biojava
[13]https://github.com/apache/commons-collections
[14]https://github.com/apache/maven
[15]https://github.com/scribejava/scribejava
[16]https://github.com/apache/dubbo
[17]https://github.com/apache/commons-math

Finding a set of open-source operational projects with test suites is not an easy task. We selected these projects mainly because they are widely known, have JUnit tests, and have distinct coding styles, an important issue for our proposal as detailed in 5.1. Since the set of selected projects is very diverse with respect to functionalities, development teams and age, we believe they are adequate to answer the RQ1. To answer RQ2, we select the methods with high cyclomatic complexity value to be evaluated since they are more challenging to be tested and maintained.

5 EMPIRICAL RESULTS

In this section, we present and analyze the experimental results with respect to the established research questions. The analyses are presented in two parts: 1) the results of the experiments conducted to answer RQ1 and 2) the results of the experiments conducted to answer RQ2.

5.1 Analysing RQ1

To answer RQ1, we utilized the 12 previously mentioned projects. Figure 4 shows the number of M/CUTs for each project (strong color bar) and the number of methods and constructors with test paths computed by ExecutionFlow (light color bar). Throughout the experiment, we noticed that ExecutionFlow was able to process an average of 75.48% of the M/CUTs. This difference is related to i)some JUnit5 features (test methods that are not public for instance); and ii) very specific coding styles that ExecutionFlow cannot handle yet. ExecutionFlow is currently capable of processing the most common JUnit5 features and coding styles that are referred by the Java community, such as Google Coding Style[18] and Java Code Conventions[19]. We are working to expand the tool´s capabilities in order to process the remaining M/CUTs and improve our results.

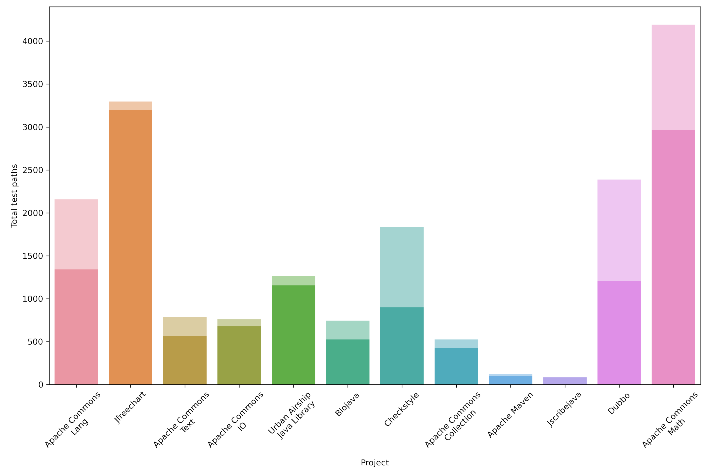


Figure 4: Number of M/CUTs for each project (strong color bar) and number of M/CUTs computed by ExecutionFlow (light color bar)

[18]https://google.github.io/styleguide/javaguide.html
[19]https://www.oracle.com/technetwork/java/codeconventions-150003.pdf

Project	Num Methods	Cyclomatic Average	Count Path	Count Stmt
Apache Collection	12	17.33	29.17	41.83
Apache Lang	5	10.20	14.00	24.80
Apache Maven	10	14.30	1161.41	49.80
Apache Math	9	13.78	5362.33	57.33
Apache Text	5	9.00	12.20	20.40
Dubbo	5	12.20	74.20	32.60
Jfreechart	14	22.43	68251233.07	54.43

Table 1: Characteristics of M/CUTs with high cyclomatic complexity.

5.2 Analysing RQ2

A source code fragment (function, method or class) is considered to have high cyclomatic complexity when its value is greater than or equal to 8 [5]. Based on this, we extracted the cyclomatic complexity metric for all methods and constructors using the Understand Tool[20]. Then, to answer the RQ2, we manually checked the computed test path for a subset of the methods with cyclomatic complexity greater or equal to 8. To this end, a hand-operated examination was performed between the computed test path and the actual (manually generated) test path. This validation process was done by executing the debugger in the covered methods.

To answer this RQ, we selected a set of 60 application methods obtained from the defined projects. Table 1 presents, for each project, the number of selected methods (NumMethods), the average cyclomatic complexity (Cyclomatic), the number of possible paths (CountPath), and the average number of declarative and executable statements (CountStmt) of those methods.

Our results in this experiment were notably satisfying since the application was able to compute the test paths of methods with cyclomatic complexity values over 50. Moreover, the generated test paths presented errors for only two out of the 60 selected M/CUTs. One of the methods is from the Apache Lang project, and the second one from the Apache Math project. In both cases, there was a problem in the processing of the source file that interrupted the test path extraction and aborted the execution.

Based on the experiments carried out, we have evidence that ExecutionFlow can accurately identify the test paths of methods with high cyclomatic complexity value. On the other hand, our experiments also showed that ExecutionFlow may not generate all possible test paths for some specific coding styles. As explained earlier, this is due to the many ways that code can be written, and it is a challenge to make the application compatible with any valid Java source-code. We believe that these specific errors can be fixed and improved as we use the application with different projects. It is worth highlighting that these errors are improvements that must be made in the application's processing, not an intrinsic limitation of the tool.

6 THREATS TO VALIDITY

An important threat to validity is related to the accuracy of the results, mainly the accuracy of the test paths, due to the use of

[20]https://www.scitools.com/

third party tools such as AJDT and AspectJ. A few error messages from those tools indeed appeared during the development and usage of ExecutionFlow. However, in all cases, the error was such that the execution was stopped and no result was generated. To deal with this threat we implemented a peer review process for both the development of ExecutionFlow and the preparation of all experiments. Furthermore, execution logs were generated and carefully inspected. The experiment was repeated whenever we detected any indication of problems with any tool.

A second possible threat is the generalization of the presented results, i.e., the claim that ExecutionFlow can be used with any Java project. As explained in Section 4, the 12 projects used in the experiments were not randomly chosen and present different sizes, complexities, coding styles and test coverage. Results have shown that the source code processing is the main limitation of the proposed tool. It is fair to claim that ExecutionFlow can handle Java codes that follow the main coding styles accepted by the Java community. However, dealing with very specific (sometimes unique) coding styles is indeed a hard task and we are continually improving the parser to incorporate new styles.

A third identified threat is related to the origin of the projects used in the experiments since 7 out of the 12 projects belong to Apache platform, which can introduce some bias in the generalization of the results. The reason for this configuration is the difficult to find appropriate projects to be used. We spent a great deal of effort looking for open-source projects with meaningful test suites that are operational and we continue this search. On the other hand, the selected Apache projects have many contributors that differ from one project to another, thus opening the window of different coding and test styles. In fact, we observe in the results that Apache projects present similar processing problems when compared to projects from other origins and we could not identify any trend specifically related to the Apache projects.

7 RELATED WORK

Several works use traces for different purposes. Bodhuim et al. [1] extract behavior models based on traces collected from a set of execution traces of Java applications to perform formal verification. They proposed a technique to construct appropriate models from the system's traces, preserving the behavior concerning the class of safety properties to be checked. Their approach consists of an iterative procedure that begins with a first raw primary process, including all system traces as alternative branches. In this work, they only present examples of their approach applied to single traces. Furthermore, it is not possible to use this approach to obtain, precisely, information about which lines of code are actually executed since the compilers perform optimizations in the source code before converting them to bytecode.

Jeevarathinam [3] proposes to generate test cases automatically from software specification.The model is represented as a symbolic execution tree, and test coverage criteria, representing each path condition based on the input specification — their model guides in checking for the correct path to be covered. With a similar purpose, Cartaxo, Neto and Machado [2] have proposed a model-based testing technique to generate test cases from UML sequence diagrams that are translated into labeled transition systems. From the labeled transition system model, a path can be obtained using the depth-first search method. In both approaches, they do not have easy access to the generated test paths. Besides, to obtain the test paths, the line number of a source code line is not considered, using other metrics.

We propose the generation of test paths based on the line number of a source code, allowing the tester to use this information for debugging or code coverage analysis. Our work introduces a tool based on the execution traces of methods and constructors used in JUnit test methods. The execution trace process is performed using the source code of these methods or constructors, using the line number corresponding to each executed statement.

8 CONCLUSION AND FUTURE WORKS

This paper proposes an approach to obtain the test paths of methods and constructors and provide them in a suitable format for further use for different purposes. Our approach relies on the use of AOP, an unit testing framework and a debugger to monitor the test execution and extract the test paths. We also present ExecutionFlow, the implementation of the proposed approach for the Java platform that uses AspectJ, JUnit and the JDB debugger and is publicly available under a MIT license. ExecutionFlow is validated through an empirical experiment performed with 12 Java open-source projects. Our results show that the application can accurately obtain the test paths of methods and constructors, even those with a high complexity cyclomatic value. In its current version, ExecutionFlow can deal with most programming styles of the java language and could process an average of 75% of the target M/CUTs.

Current work include the support for specific Java programming statements and practices that are not yet handled by the proposed approach, such as the support to additional visibility levels of JUnit 5.

ACKNOWLEDGMENTS

This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior - Brasil (CAPES) - Finance Code 001. This work is also partially supported by Propesq-UFRGS, FAPERGS and CNPq, Brazil.

REFERENCES

[1] Thierry Bodhuin, Federico Pagnozzi, Antonella Santone, Maria Tortorella, and Maria Luisa Villani. 2009. Abstracting Models from Execution Traces for Performing Formal Verification. *Communications in Computer and Information Science* 59, 11 (2009), 143–150.
[2] Emanuela G. Cartaxo, Francisco G. O. Neto, and Patricia D. L. Machado. 2007. Test Case Generation by means of UML Sequence Diagrams and Labeled Transition Systems. In *36th IEEE International Conference on Systems, Man and Cybernetics*. 1292–1297.
[3] Jeevarathinam and Antony Thanamani. 2010. Towards Test Case Generation from Software Specification. *International Journal of Engineering Science and Technology* 2, 11 (2010), 6578–6584.
[4] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. M. Loingtier, and J. Irwin. 1997. Aspect-oriented programming. In *11th European Conference on Object-Oriented Programming (ECOOP)*. 220–242.
[5] T.J. McCabe. 1976. A Complexity Measure. *IEEE Transactions on Software Engineering* 2, 4 (1976), 308–320.
[6] Ronald Rivest. 1992. The MD5 Message-Digest Algorithm. *RFC* 1321, 1 (1992), 1–21.
[7] Jie Zhang, Ziyi Wang, Lingming Zhang, Dan Hao, Lei Zang, Shiyang Cheng, and Lu Zhang. 2016. Predictive Mutation Testing. In *25th International Symposium on Software Testing and Analysis*. 342–353.