International Journal of Recent Technology and Engineering (IJRTE)
ISSN: 2277-3878 (Online), Volume-7 Issue-4, September 2018

AOP Approach for Testing Program Invariants

Manish Jain, Dinesh Gopalani

Abstract*: Mankind is becoming increasingly dependent on software technology which intensifies the need for thorough software testing and development of automated software testing techniques. We have proposed Aspect Oriented Programming (AOP) as a potential methodology for performing various types of automated software testing. In this paper, we have particularly demonstrated the use of AOP for performing testing of invariants in software artifacts. Invariants are assertions about the state of a program that are expected to be true each time control reaches a particular point in the program. If an assertion is found not to be true, then such condition corresponds to a bug discovery. In this paper, adherence to run time as well as compile time invariants in Java applications has been tested using AspectJ, which has become the de-facto standard for AOP. We established that our AOP approach for testing invariants has got several advantages like faster test execution, no test-code scattering etc.

Index Terms: Aspect oriented programming, AspectJ, aspects, crosscutting, invariant testing, software testing.

I. INTRODUCTION

As software applications have become an inevitable part of human life, testing the software for quality assurance is of utmost importance in the software development life cycle. Testing amounts to observing the execution of a software system to validate whether it behaves as intended and further identify potential malfunctions. Software testing can be automated using testing tools or methodologies and such automation improves and speeds up the process of software testing. The ultimate goal of automated software testing is to free up people to do works that add value to the end software. In order to accelerate the process of software testing, researchers and developers have come up with different automated testing tools. For example, for the testing of Java applications, there exist several tools like JUnit, Selenium, TestNG, Arquillian, Mockito, JMeter etc. Among these testing tools, JUnit happens to be the most popular tool for testing Java applications [1], [2]. The paradigm of Aspect Oriented Programming (AOP) renders a well built mechanism for the separation of concerns. A functionality expected from a software system is called a concern. Thus a software application is actually a implementation of one or more concerns. There are two types of concerns:

Primary Concern: These are the business logic concerns also called the core concerns Secondary Concerns: These are the system level concerns which are called the crosscutting concerns Crosscutting represents a situation when the code for a secondary concern is scattered into code structures throughout the system although the functionality of such code doesn't directly relate to the functionality defined for the containing code structures. AOP introduces a new unit of modularisation, called aspect which obliterate the issue of crosscutting. The crosscutting concerns are implemented in this aspect instead of directly intermixing them with the core modules. In our antecedent papers [3], [4], [5], [6], we have proposed the use of AOP methodology as an automated testing tool and also established the use of aspects in AspectJ for carrying out different types of testing of Java applications. We also detected remarkable bugs into various widely used Java applications using our proposed approach.

The proposed AOP methodology automates the complete life cycle of software testing namely, the generation of test cases, write the test script, execute the test cases and further compare the results with the expected results and prepare a test report. There exists aspect oriented implementations for all popular programming languages using which the testing code can be written in the form of before, after or around advices within the aspects. The testing aspect code is weaved with the source code under test as shown in Fig. 1.

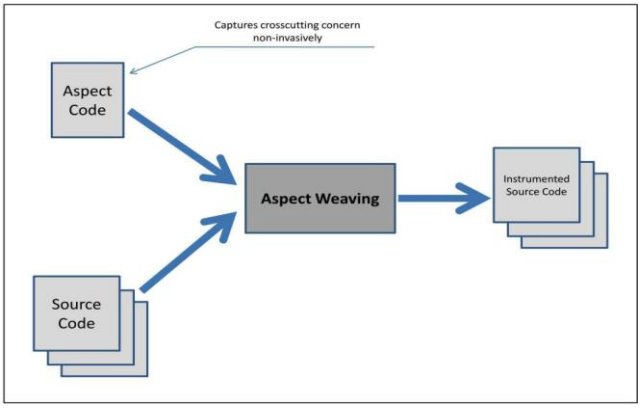

Fig. 1. Weaving of testing aspect code with source code under test

In this paper, we shall particularly discuss the use of AOP methodology for the purpose of carrying out invariant testing. Invariants are the conditions that are expected to hold true for a part or whole of the software program. There are two type of invariants: rum time invariants and compile time invariants. Invariants which are used to specify contracts regarding the run time behaviour of the program are called run time invariants. Conditions that are supposed to be enforced at the time of compilation are called compile time invariants.

Revised Manuscript Received on 30 September 2018.
***** Correspondence Author
Manish Jain*, Department of Computer Science, Malaviya National Institute of Technology, Jaipur, India
Dinesh Gopalani, Department of Computer Science, Malaviya National Institute of Technology, Jaipur, India

Using the conventional techniques, the testing code for testing the invariants shall be scattered at all places in the code structure where the invariant condition under test is expected to hold true. AOP obliterates such test-code scattering. Further the test execution time is also reduced as we shall discuss in the forthcoming sections.

The paper is organised as follows: in Section II, we provide a list of the related work done in this field. In Section III,

we have discussed how AOP can be utilised for carrying out invariant testing with suitable examples. Section IV enlists the various benefits of using AOP for invariant testing. At the end, Section V is used to specify the conclusion and future work. References are enumerated at last.

II. RELATED WORK

Duclos et al. [7] used AspectC++ for carrying out certain basic testings of C++ programs. Sioud [8] implemented the missing garbage collection in C++ using AspectC++. Java has its own garbage collection mechanism in place but still there are possibilities of memory leakages like buffer overflows and null pointer exceptions. Sokenou and Herrmann [9] used AOP to test programs written in AOP languages. They stated that aspects seem worthy for testing the aspect-oriented systems. Copty et. al [10] have used AspectJ to implement certain functionalities of the concurrency testing tool ConTest. We enhanced the ideas of these researchers and used aspects in AOP to perform various types of software testing of applications written in various languages. Moreover, in this paper we have established several benefits of using AOP for the purpose of invariant testing.

Daszczuk [11] has demonstrated how a large class of errors in the operating systems can be detected by invariant testing. He used an idle process to evaluate the invariants of an operating system. We, on the other hand, use AOP and test for invariant claims. Ernst et. al [12] developed Daikon, a tool that reports likely program invariants. Their methodology is based on machine learning techniques. Further, their methodology and tool were limited to only four programming languages, namely C, C++, Java and Perl. Our methodology can be used for almost all programming languages as there exists fully developed AOP versions of most popular programming languages. Hangal et. al [13] developed a tool called Dynamic Invariant Detection [Checking Engine (DIDUCE) (DIDUCE) that dynamically formulates hypotheses of invariants obeyed by the program. They detected bugs using the concept of dynamic invariant detection and checking. DIDUCE could detect anomalies in dynamic run Java programs only. We covered testing for static compile time invariants as well. Further, the run timer overhead of DIDUCE was high whereas our proposed methodology reduces overheads and renders lesser test execution time.

III. INVARIANT TESTING USING AOP

There are various types of software testing classified based on the knowledge of the system, phase at which testing is being performed, extent of automation, source code execution or non-execution, functional or non-functional behavior of the software etc. In our antecedent papers [3], [4], [5], [6], we have proposed the use of AOP methodology as an automated testing tool and established the use of aspects for carrying out different types of testing. AOP languages provide us with pointcuts which can be used to capture joinpoints from the program code. Using suitable pointcuts, we can capture joinpoints of interest from the program code which need to be tested. There are wildcard pointcuts available in AOP languages which can be used to capture multiple joinpoints that are to be tested simultaneously. Further, advice can be used to write the appropriate testing code which shall be executed before/after/around the captured joinpoints and attempt to discover bugs.

In this paper, we identify the applicability of AOP for performing invariant testing. An invariant can be defined as a condition or guideline that is mandated to hold true for a program component or may be even for the whole program structure [14]. We used pointcuts in AOP to capture all the execution points where the invariant condition is supposed to be true and further used suitable advice to check for the correctness of the invariant condition at all such points. This doesn't require any modifications to be made in the source code. Using aspects, invariant conditions imposed at both compile time as well as run time can be tested. For example, using a simple aspect, we tested the runtime invariant condition that one (or all) methods of a class and all it subclasses should never return a null value as shown in Fig.

2. In this code listing, the after returning () advice captures the return value of all the methods defined by the pointcut and if it is null, an error message is shown. The method getSignature() of the thisJoinPoint reference variable returns the signature for the executing join point. Likewise, the method getSourceLocation() allows access to the source location information of the joinpoint. We have used the getSignature() and getSourceLocation() methods in this example to get the signature of methods which return null and the corresponding source code line number where null is returned. Compile time invariant conditions can also be checked using AOP. There could be compile time invariant conditions which should hold true for a part of or complete source code like a particular API should never be called or a particular optimised method should be considered for objects of an class or that a private member should not be set outside a setter function. AspectJ provides us with declare warning and declare error which are static crosscutting instructions that we used to generate compile time warnings or errors respectively when a particular usage pattern is detected in the source code of the application. We used the code snippet shown in Fig. 3 to issue a compile time warning whenever the value of any of the private members of a class, say Classname, is set outside the setter function.

Similarly, in order to ascertain another runtime invariant condition that a non-static field inside a class in Java should be set only within a constructor and not outside constructor, we used the aspect as shown in Fig. 4.

```
public aspect runtimeinvariant
{
      after () returning (Object obj) : call(Object Classname+.method(..))
      {
            if (null == obj)
            {
                  String error = "Null value returned at " + thisJoinPoint.getSignature() + " from "
                  + thisJoinPoint.getSourceLocation();
                  System.out.println(error);
            }
      }
}
```

Fig. 2. Runtime invariant testing using AspectJ-Example I

```
public aspect compiletimeinvariant
{
declare warning : within(Classname) && set(!public * *) && !withincode(* set*(..)) :
    "private field should be accessed only through a setter function" ;
}
```

Fig. 3.Compile time invariant testing using AspectJ

```
public aspect runtimeinvariant
{
      pointcut staticFieldAccess() : set(!static * Classname.*);
      pointcut creation() : execution(Classname.new(..));
      before() : staticFieldAccess() && !cflow(creation())
      {
            throw new Error("non static field should be accessed only within a constructor");
      }
}
```

Fig. 4. Runtime invariant testing using AspectJ-Example II

IV. COMPARISON AND ANALYSIS

Daikon tool for the dynamic detection of invariants uses instrumenters which instrument the source code and produces a new version of the source code to check for the invariants[12]. With our technique, source code is instrumented with aspects externally. Further, Daikon can test for invariants in C, C++, Java and Perl programs but this framework can be used only for primitive and string types and is not suitable for complex types. Dynamic Invariant Detection

Checking Engine (DIDUCE) is another invariant detection tool which is meant for Java programs. Alike AOP, it instruments a program's bytecode. DIDUCE dynamically formulates hypotheses of invariants obeyed by the program and considers an anomaly only when there is a large deviation from the past value for a variable at a program point [13]. Thus, it can be used to provide dynamic invariant violations when a software error occurs but not for testing invariant conditions like "the values of a variable should lie within desired constraints". Various advantages of using AOP for carrying out invariant testing are discussed hereunder:

A. Scattering of testing code

Most of the times, the invariant condition is required to be tested across the whole program itself and such testing is thus crosscutting in nature. For example, if we want to test

that the value of a program variable should always be within certain constraints (like y=ax+b or a x b or x2y) then it would be required to write testing code at all the places within the program code where the program variable is being (directly or indirectly) assigned a value. Thus the testing code shall be scattered at several places. Nevertheless using aspects, we can capture all accesses to the variable using the set pointcut and further write an after advice to check that the value assigned is within desired constraints. Comparing this with JUnit, there is construct called assert which can be used to check for internal invariants within the test class, but there exists no provision for testing the crosscutting invariant conditions.

B. Reduced lines of testing code

The number of lines of testing code are also reduced by the use of wildcard pointcuts which are available in AspectJ and other AOP languages as well. For example, the simple pointcut execution(* *(..)) shall capture the execution of any method regardless of return or parameter types. Thus if we want to test for the condition whether any of the methods in the whole program returns null, which can lead to a null pointer exception, this single pointcut would be sufficient. Similarly, wild card pointcuts can be used to capture joinpoints that share common characteristics and then can be tested all at once. However, there is no such mechanism in conventional testing tools like JUnit and hence for testing different methods even

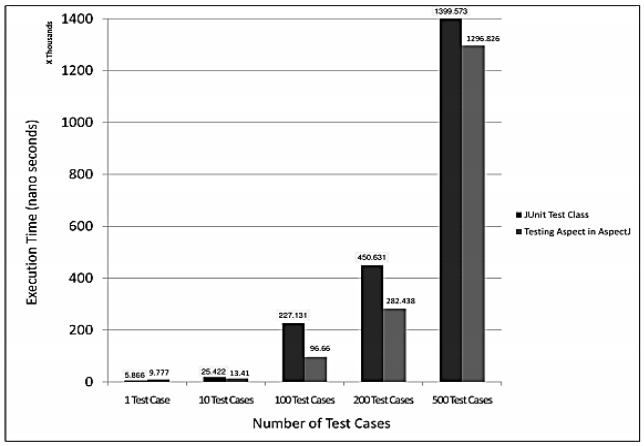

Fig. 5. Test execution times for testing a 2-argument function using JUnit and AspectJ, performed on a system with Windows XP SP3 having Intel T6670 Processor and 4GB RAM

with common attributes, separate testing code has to be written which increases the number of lines of code.

C. Reduced test execution time

The number of invariant test cases for testing bigger projects is too high [15] and practically it is quite time consuming to test the software with all such test cases using the conventional testing techniques. We observed that the execution time for running test cases is shorter with our proposed AOP testing technique. For example, we tested a simple two argument function using JUnit and AspectJ with multiple test cases for the two inputs. The function returned an integer value which was compared with the expected output. The observations depicting the test execution times have been shown in Fig. 5.

V. CONCLUSION AND FUTURE WORK

Invariant testing involves examining conditions that are expected to hold true for a program component or may be for the whole program implementation. Invariants conditions could be run time like a newly created object should not be null or the return value of a method should be within a specified range or a particular method should not be called from any class other than the allowed class. The context at a joinpoint can be captured with the help of constructs like args, after returning advice or within in AspectJ and such context can be used to ascertain the imposed run time conditions. Static invariant conditions that are required to be tested could be asserting that a particular API is never called or verifying that the private members are set within the setter functions only.

Compile-time declarations in AspectJ like declare warning or declare error can be used to ascertain such static conditions. In this paper, we have established the benefits of using AOP for invariant testing over the conventional techniques. No test-code scattering, reduced test execution time and reduced lines of testing code are the key benefits of using our approach. As far as the available automated testing tools are concerned, to use these tools, testers need skills like knowledge of test tools, general software, domain and system knowledge etc [16]. Aspects, on the other hand, are easier to be adopted into existing development projects.

There are several lines of experimentation which arise from our research work which can be carried out in future. Our AOP approach can be extended to cover other testing types like concurrency testing, regression testing, loop testing etc. Moreover, since AOP languages like AspectJ, happen to be a new programming paradigm and not all developers or testers are familiar with this technology, a Domain Specific Language (DSL) can be created whose syntax is natural language-like and the statements written thereof are automatically converted to testing aspects using the DSL parser so that even the testers who do not have the knowledge of AOP can still avail the benefits of testing using AOP. The preliminary results obtained in this direction are encouraging.

REFERENCES

1. A. Hussain, A. Razak, and E. Mkpojiogu, “The perceived usability of automated testing tools for mobile applications,” Journal of Engineering Science and Technology, vol. 12, pp. 89–97, 04 2017.
2. P. S. Kochhar, F. Thung, N. Nagappan, T. Zimmermann, and D. Lo, “Understanding the test automation culture of app developers,” in 2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST), April 2015, pp. 1–10.
3. M. Jain and D. Gopalani, “Use of aspects for testing software applications,” in 2015 IEEE International Advance Computing Conference, June 2015, pp. 282–285.
4. M. Jain and D. Gopalani, “Memory leakage testing using aspects,” in 2015 International Conference on Applied and Theoretical Computing and Communication Technology, Oct 2015, pp. 436–440.
5. M. Jain and D. Gopalani, “Aspect oriented programming and types of software testing,” in 2016 Second International Conference on Computational Intelligence Communication Technology, Feb 2016, pp. 64–69.
6. M. Jain and D. Gopalani, “Testing application security with aspects,” in 2016 International Conference on Electrical, Electronics, and Optimization Techniques (ICEEOT), March 2016, pp. 3161–3165.
7. E. Duclos, S. L. Digabel, Y. G. Gueheneuc, and B. Adams, “Acre: An automated aspect creator for testing C++ applications,” in IEEE 7th European Conference on Software Maintenance and Reengineering, 2013, pp. 121–130.
8. A. Sioud, “Gestion de cycle de vie des objets par aspects pour C++,” Master’s thesis, UQaC, 2006.
9. D. Sokenou and S. Herrmann, “Aspects for testing aspects?” in 1st Workshop on Testing Aspect-Oriented Programs, 2005.
10. S. Copty and S. Ur, “Multi-threaded testing with AOP is easy, and it finds bugs!” Lecture Notes in Computer Science, vol. 3648, pp. 740–749, 2005.
11. W. B. Daszczuk, “Invariant testing technique for debugging a structured operating system,” Microprocess. Microsyst., vol. 11, no. 4, pp. 205–208, May 1987. [Online]. Available: http://dx.doi.org/10.1016/0141-9331(87)90339 5
12. M. D. Ernst, J. H. Perkins, P. J. Guo, S. McCamant, C. Pacheco, M. S. Tschantz, and C. Xiao, “The Daikon system for dynamic detection of likely invariants,” Sci. Comput. Program., vol. 69, no. 1-3, pp. 35–45, December 2007.
13. S. Hangal and M. S. Lam, “Tracking down software bugs using automatic anomaly detection,” in Proceedings of the 24th International Conference on Software Engineering. ICSE 2002, May 2002, pp. 291–301.
14. M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin, “Dynamically discovering likely program invariants to support program evolution,” in IEEE transactions on software engineering, vol. 27, 2001, pp. 99–123.
15. D. R. Kuhn and V. Okun, “Pseudo exhaustive testing for software,” in 30th Annual IEEE Software Engineering Workshop, 2006, pp. 153–158.
16. D. Rafi, K. Moses, K. Petersen, and M. Mantyla, “Testing non-functional requirements with aspects,” in IEEE 7th International Workshop on Automation of Software Test AST, 2012, pp. 36–42.