On the Applicability of Random Testing for Aspect-Oriented Programs*

Reza Meimandi Parizi, Abdul Azim Abdul Ghani, Rusli Abdullah, and Rodziah Atan

Department of Information System, University Putra Malaysia, 43400 Serdang, Malaysia
{parizi,azim,rusli,rodziah}@fsktm.upm.edu.my

Abstract

Random Testing (RT) and its derivatives such as Adaptive Random Testing (ART) are active and important research topics in software testing, which have also a niche in practical settings due to the merits they offers, e.g. fault-detection capacities at low cost, ease of implementation, reliability estimation, facility for automation and so forth. Inspired by these advantages, we believe the idea behind random testing can be worthwhile and attractive for testing aspect-oriented programs since current research on testing of AOP, especially automated has not been adequately performed and is still in infancy. In this paper, we propose a preliminary approach to automated random testing of aspect-oriented programs, which are becoming an important part of software engineering theory and practice. This paper also includes a survey of applicable testing techniques and discussion of established testing methods in both area of Aspect-Oriented Programming (AOP) and Random Testing (RT).

Keywords: Aspect-Oriented Programming, Random Testing, Adaptive Random Testing, Aspect Testing

1. Introduction

Aspect-oriented programming, a relatively new programming paradigm, provides means for modularizing and separating crosscutting concerns in which it produces a system with higher degree of modularity than the other paradigms such as Object-Oriented Programs (OOPs). However, since it has new constructs and properties that other programming paradigms do not have it brings new challenges and aspect-related faults, not present when testing other types of programs, which in turn make them to be not addressed using traditional unit or integration testing approaches. For instance, given an aspect in a bank system that is supposed to implement authentication (as crosscutting concern) before calling to a set of methods. AOP can get this requirement done by simply capturing any call to the given methods (i.e. join points) in the core code by means of pointcut and then injects those identification functionalities or behaviors (i.e. advice) before the methods which were invoked by callers. If AOP, more specifically the pointcut, misses to capture some call to the given methods therefore the authentication will not properly be applied and it may cause a severe failure in the system (This is referred as one of the aspect fault called incorrect strength in pointcut patterns). Therefore, this new paradigm, although enhances the modularity, cannot provide correctness by itself and thus it also like any other programs is prone to errors (by

* A preliminary version of this paper was presented at the ISCA 18th International Conference on Software Engineering and Data Engineering (SEDE-2009) [1].
developer, programmer, etc.) and requires the use of testing strategy to produce validated and high quality AO software.

On the other hand, the current research in testing aspect-oriented programming has focused on systematic approaches for testing this relatively new paradigm (e.g. code-based structural testing, specification-based functional testing) so far. In reality, however, 

(1) Efficient testing and verification of this new paradigm in a thorough and quantitative manner is still neglected using the current proposed techniques. It means, although systematic methods are often intuitively preferred but their significance is in doubt due to they cannot be analyzed in any precise terms that are meaningful for software quality (e.g. confidence in MTTF- Mean Time to Failure) [2]. For instance, many test standards require that 80% of program paths (making some special provision for loops) be executed during testing. Yet there is no information on the effectiveness of 80% coverage at finding failures, nor on the significance of an 80%-coverage test that happens to succeed (with systematic methods we know what we are doing, but not what it means; only by giving up all systematization can the significance of testing be known [2]). 

(2) The systematic methods are subjective and incomplete, either in theory or practical, in the sense that any choices or elements of information used to drive testing may not be evaluated with respect to their impact on the entire test process or not fully consideration of the all features of AOP constructs. These can vastly influence the testing results and its quality in which it may stem from bias occurred in the selection of the generated test data.

Model-based and structural-based are two significant categories of the AOP systematic testing approaches proposed in the literature. For each of which several researches has been proposed. However, these approaches are not yet complete in the case of above issue (#2). For instance, in model-based testing of AOP, everything revolves around models in which once a failure is reported the aspect-oriented model will be inspected in order to determine whether the aspect-oriented program under test has been faulty or not. It is normally done by checking the model against the requirement itself (building time) and exercised test case for finding the violation. This needs developing of some automatic model-checking mechanism or tool which has not been adequately addressed at the current state of the art. Furthermore, test generation is only possible if a detailed aspect-oriented model of the intended behavior is available, e.g. based on UML or finite state machine or any other modeling notation. Moreover, it is even harder to guarantee, assuming a model was initially produced, that the project will keep it up to date as the software changes [3]. Therefore, the quality of these models can dramatically influence on the quality of the testing and its results. If the choices either in building the model, test case generation or checking the model are not wisely enough therefore, it may not produces a good testing results and possibly causes bias.

In particular, the AOP structural testing approaches practically are not complete form the idea of coverage. For instance, although at the current state of the art techniques for Control Flow Graph (CFG) of aspect-oriented programs have been possibly defined and developed, to be used in structural testing of these programs, but these techniques still are not fully completed to cover the entire features of an AO program. E.g. the inter-type declaration that introduces fields or methods, other types of join points such as method or constructor calls and around advice have not been addressed by these techniques yet. Therefore, even in structural testing of AOP using CFG the coverage it requires may not be very useful in exposing failures in which the related coverage (Aspectual branch coverage criteria, which characterizes aspectual behavior only- e.g. coverage of advice
or Interaction coverage, which characterizes aspectual composition regarding to the interaction between base code and aspect codes) to those neglected parts will never be occurred and executed during testing and consequently resulting in very low possibility to hit their faults either.

(3) Finally, the complex and systematic methods are difficult to be carried out especially in an automated manner. Because they often have restrictions imposed by the language constructs or scope, e.g. most of the proposed testing techniques in the literature are meant for AspectJ than the other AO languages (Table 1 clearly evident to this matter) as well as difficulty in executing a great possible combinations of the input values since there is problem of restriction in selecting a sub-set of input data from the program input domain to be used in testing. The latter can viewed as lack of optimized coverage in the space of possible values.

The aforementioned issues (1-3) associated with systematic methods motivates a need for developing an AOP testing method which is capable of alleviating the non-quantification of results and statistical estimation, the subjectivity, incompleteness, difficulty in implementation, need of human efforts and so on. Random testing (RT) can be viewed as potential approach on this regard, which has capability of replacing human choice with chance selection [2], so that it is unbiased and not influenced by wrong assumptions made by human tester (lack of human bias). Furthermore, random testing is quite simple (simplicity of implementation) and is not particularly costly in comparing with systematic approaches. In addition, it is efficient in terms of test case generation and implying reliability and statistical estimates, which means random testing the only strategy that can offer any statistical prediction of significance of the results. Hence, if such a measure of reliability is necessary, random testing is the only option. Moreover, random testing is one of the approaches to make the testing process fully automated [4] (provided that an automated or effective oracle is also available), where is almost difficult to be done in other testing strategies. Nevertheless, work on RT has so far been only considered for procedural and recently object-oriented paradigms but applying RT to the testing aspects remains challenging and still open research area to be conducted, where the proposed testing method of this research, is an attempt towards this direction.

The challenges regarding the random testing of aspect-oriented programs are as below where challenges 1-3 have been generally addressed by this work, whereas the challenge 4 is in the process of being research.

1. How to generate test cases randomly for the aspect type input domain?
2. How to select the best test cases from the input domain for Aspect Under Test (AUT)?
3. How to assess the correctness of each selected test?
4. Does applying random testing concept on AOP can enhance the effectiveness of testing these programs? If so, when it can outperform and when it cannot. In particular, how could this approach be best used?

The rest of the paper is organized as follows: Section 2 presents the background on AOP concepts; Section 3 gives the related work; Section 4 describes the random testing approach for AOP; Section 5 gives the experimental evaluation outline; Section 6 reports the conclusion and future work on this area of research.

2. Background on AOP
This section provides a bird’s-eye view of AOP concepts and AspectJ language, which is the most commonly used aspect-oriented programming language as well as target language used in this work.

The main idea behind the AOP is to separate the core concern, central functionality, of a module from its crosscutting concerns, system-level requirements that cross (span) multiple modules e.g. logging, authentication. The crosscutting concerns will be implemented in a separated unit of modularization called aspect (called aspect code) whereas the core concerns will be implemented using the conventional development methodology like object-oriented (called base code). Simply, AOP works in the way that whenever a crosscutting concern needs to be executed, the corresponding aspect is triggered, by means of the join points and pointcuts, to execute the actual code, by means of advice, which implements the given crosscutting concern. Then, it returns the control back to the base code to go on with the rest of the code. The resulting system is easy to maintain, understand and to evolve due to the omission of code tangling and scattering.

AspectJ is a language extension to Java, where it introduces some new concepts and constructs in support and implementation of aspect-orientation. However, although at the current state of art the terminology for AOP has not yet been standardized but the following AspectJ constructs are getting common among the community and the most of AOP languages. The joint points are well-defined points or locations in the program (base code) execution where the new behaviors (crosscutting) can be executed or injected, e.g., call to a method or constructor invocations in the execution of a program. The pointcuts are program constructs or means that select a set of join points and their associated context based upon matching a give pattern in their designator. The advice is a method-like construct that contains the actual code (or crosscutting concerns implementation) to be executed at a join point that has been reached by a pointcut. Advice can be executed at three different places when a join point is matched: before, after, or around. No matter which type, first of all a pointcut must be triggered before execution of the advice body in order to capture the desired join point. The introduction provides modification into the source code of the system, e.g. aspects, classes, interfaces. Finally, the aspects are much like of classes in the base code, e.g. object-oriented, that encompasses the all constructs for implementing the crosscutting concerns.

3. Related work

This section reviews the related literature in testing aspect-oriented programs and random testing since the proposed work is related to these areas.

3.1. Testing aspect-oriented programs

A range of topics related to the testing aspect-oriented programs has been proposed yet. Among them code-based testing, model-based testing, fault and risk models, regression testing, mutation testing, and infrastructure for, automated, test generation and execution are highly considerable.

Code-based is a white-box or structural testing strategy that emphasizes on designing test cases based on objectives derived from the aspect implementation. In other words, it uses the knowledge of AO program internal structure, by means of some structural representatives such as control flow graph or data flow models, in order to generate the
For instance, test that triggers specific programs paths or exercises a given control flow.

Model-based testing is a black-box testing technique in which test cases are derived partially or fully from an aspect-oriented model. Modeling methodology with aspect is regards to another research area called Aspect-Oriented Modeling (AOM). In context of AOM lots of research efforts have been devoted to address the characteristics of aspects. This includes aspect-oriented extensions to a wide range of modeling notation languages such as Unified Modeling Language (UML) [7-14], Finite State Machine (FSM) [15-19], Markov chain, Petri nets, grammars or formal methods [20], which describes the requirements of the system under test and its specified functionalities.

Fault models for aspect-oriented programs are a set of foreseeable fault types or source that is likely to be occurred in the context of aspect-oriented programming. In other words, fault models define some sort of failure patterns that can help the tester or designer to predict which parts of the system are prone to cause faults and in turn what parts must be tested (their corresponding testing criteria) in order to alleviate the consequences of particular faults. In the literature these candidate AO fault models ranging from solely pointcut related faults [21, 22], where they give the most likely faults that can only occur in pointcuts to combination of all fault types stemmed from advice, pointcut, and other unique features of aspect-oriented programs such as inter-type declaration or introduction [23-32].

As aspect-oriented software is modified, during development and maintenance, it is regression tested [33-35] to provide confidence that the changes did not introduce unexpected problems. Because the size of the regression test suite typically keeps growing, a regression-test-selection technique can be employed to reduce the cost of regression testing. A safe regression-test-selection algorithm selects every test case that may reveal a fault in the modified software. Various regression test selection techniques have been developed for procedural languages, object-oriented languages, and recently for aspect-oriented programs [36-38].

Mutation testing approaches for aspect-oriented programs have mostly focused on mutation testing of pointcuts (a type of fault-based pointcut testing) because faults in pointcuts can cause aspects to fail to satisfy their requirements and this in turn causes the system fails to perform its complete functionality [39]. Thus, testing pointcuts is necessary in order to ensure correctness of aspects and finally the final system success. Generally, The mutation testing techniques for aspect-oriented, in the literature, rang from automated generation of pointcut mutants [39, 40] specifically for mutation testing of PCDs (pointcut descriptors) [41, 42] to definition of a comprehensive set of mutation operators [43, 44] and testing these mutants against the designed test data.

For the brevity, the above presented works in the favor of different AOP testing techniques have been summarized in Table 1.

<table>
<thead>
<tr>
<th>Work</th>
<th>Title</th>
<th>Language</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>No.</td>
<td>Title</td>
<td>Method(s)</td>
<td>Format</td>
</tr>
<tr>
<td>-----</td>
<td>----------------------------------------------------------------------</td>
<td>---------------------------</td>
<td>----------</td>
</tr>
<tr>
<td>48</td>
<td>Unit-Testing Aspectual Behavior</td>
<td>JAML</td>
<td>Code-based</td>
</tr>
<tr>
<td>49</td>
<td>On Identifying Bug Patterns In Aspect-Oriented Programs</td>
<td>AspectJ</td>
<td>Code-based</td>
</tr>
<tr>
<td>51</td>
<td>Detecting Redundant Unit Tests For AspectJ Programs</td>
<td>AspectJ</td>
<td>Code-based</td>
</tr>
<tr>
<td>52</td>
<td>Interference Analysis For AspectJ</td>
<td>AspectJ</td>
<td>Code-based</td>
</tr>
<tr>
<td>53</td>
<td>Testing Aspect-Oriented Programs As Object-Oriented Programs</td>
<td>AspectJ</td>
<td>Code-based</td>
</tr>
<tr>
<td>54</td>
<td>Testing Aspect Oriented Programming Pointcut Descriptors</td>
<td>AspectJ</td>
<td>Code-based</td>
</tr>
<tr>
<td>55</td>
<td>Generalizing AOP For Aspect-Oriented Testing</td>
<td>AspectJ</td>
<td>Code-based</td>
</tr>
<tr>
<td>56</td>
<td>A Systematic Aspect-Oriented Refactoring And Testing Strategy, And Its Application To Jhotdraw</td>
<td>AspectJ</td>
<td>Code-based, Fault model</td>
</tr>
<tr>
<td>58</td>
<td>A Unit Testing Framework for Aspects without Weaving</td>
<td>AspectJ</td>
<td>Code-based</td>
</tr>
<tr>
<td>7</td>
<td>Testing Aspect-Oriented Programs with UML Design Models</td>
<td>UML, AspectJ</td>
<td>Model-based</td>
</tr>
<tr>
<td>6</td>
<td>A Model-Based Approach to Test Generation for Aspect-Oriented Programs</td>
<td>UML, AspectJ</td>
<td>Model-based</td>
</tr>
<tr>
<td>9</td>
<td>Automated Test Code Generation from UML Protocol State Machines</td>
<td>UML, AspectJ</td>
<td>Model-based</td>
</tr>
<tr>
<td>10</td>
<td>Generation of Test Requirements from Aspectual Use Cases</td>
<td>UML, AspectJ</td>
<td>Model-based</td>
</tr>
<tr>
<td>11</td>
<td>State-Based Testing Of Integration Aspects</td>
<td>FSM, AspectJ</td>
<td>Model-based</td>
</tr>
<tr>
<td>12</td>
<td>State-Based Incremental Testing Of Aspect-Oriented Programs</td>
<td>FSM, AspectJ</td>
<td>Model-based</td>
</tr>
<tr>
<td>13</td>
<td>Model Checking Aspect-Oriented Design Specification</td>
<td>FSM, AspectJ</td>
<td>Model-based</td>
</tr>
<tr>
<td>14</td>
<td>A State-Based Approach To Testing Aspect-Oriented Programs</td>
<td>FSM, AspectJ</td>
<td>Model-based</td>
</tr>
<tr>
<td>15</td>
<td>Test Generation from Aspect-Oriented State Models</td>
<td>FSM, AspectJ</td>
<td>Model-based</td>
</tr>
<tr>
<td>16</td>
<td>Aspect flow graph for testing aspect-oriented programs</td>
<td>UML, AFG, AspectJ</td>
<td>Model-based</td>
</tr>
<tr>
<td>17</td>
<td>A Model-Based Detection Of Conflicts Between Crosscutting Concerns: Towards A Formal Approach</td>
<td>AspectJ</td>
<td>Model-based</td>
</tr>
<tr>
<td>20</td>
<td>Generating Unit Test Sequences For Aspect-Oriented Programs: Towards A Formal Approach Using UML State Diagrams</td>
<td>UML, AspectJ</td>
<td>Model-based</td>
</tr>
<tr>
<td>22</td>
<td>Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs</td>
<td>AspectJ</td>
<td>Mutation</td>
</tr>
<tr>
<td>23</td>
<td>Efficient Mutant Generation for Mutation Testing of Pointcuts in Aspect-Oriented Programs</td>
<td>AspectJ</td>
<td>Mutation</td>
</tr>
<tr>
<td>24</td>
<td>Mutation Testing for Aspect-Oriented Programs</td>
<td>AspectJ</td>
<td>Mutation, Fault model</td>
</tr>
<tr>
<td>25</td>
<td>AjMutator: A Tool for the Mutation Analysis of AspectJ Pointcut Descriptors</td>
<td>AspectJ</td>
<td>Mutation</td>
</tr>
<tr>
<td>26</td>
<td>A Test-Driven Approach to Developing Pointcut Descriptors in AspectJ</td>
<td>AspectJ</td>
<td>Mutation</td>
</tr>
<tr>
<td>27</td>
<td>Towards The Systematic Testing Of Aspect-Oriented Programs</td>
<td>AspectJ</td>
<td>Fault model</td>
</tr>
<tr>
<td>28</td>
<td>Towards A Fault Model For AspectJ Programs: Step 1 – Pointcut Faults</td>
<td>AspectJ</td>
<td>Fault model</td>
</tr>
<tr>
<td>29</td>
<td>Will Aspect-oriented Programming Improve Software Quality?</td>
<td>AspectJ</td>
<td>Fault model</td>
</tr>
<tr>
<td>30</td>
<td>Is AOP Code Easier or Harder to Test Than OOP Code?</td>
<td>AspectJ</td>
<td>Fault model</td>
</tr>
</tbody>
</table>
The first column provides the reference numbers of the work. The second column presents the title of given published work. The third, Language, column gives the design or modeling notation, e.g. Unified Modeling Language (UML) or Finite State Machine (FSM), and programming languages used in the corresponding work on which the testing approach is built or applied. The forth, Category, column lists the classification of the corresponding works related to aspect testing in which the primary focus of the testing approach is given. Different techniques are utilized in different level of abstraction and development. For instance, the techniques and tools used in aspect-oriented design, model, and level are different from those used in aspect-oriented coding or programming level. From Table 1 we can observe that these works all has focused on systematic code-based or model-based generation of unit or integration tests, whereas the proposed research in this work focus on an unsystematic approach in which the random-based test generation and execution in an automated conformance manner is taken into account for aspects. However, there is similarity between proposed work and those works in which AspectJ is used as target programming language.

3.2. Random testing techniques

Random Testing (RT) as one of the techniques which include automated test input generation and selection has been studied and applied for different paradigms and
application domains so far. Although, the first emergence of the random testing was meant for numerical input domain but with passage of time and emerging different paradigms the interest in random testing has been dramatically increased due to the merits it offers. This matter is clearly evident by various studies in the literature which apply RT to the area of their interest. Random testing techniques intuitively can be categorized into pure and enriched due to the strategies they use for test input generation and selection, see Figure 2.

By “enriched” it is meant to those strategies that have been equipped with some guidance to their normal random process (pure) to pick up test inputs which give higher efficiency in their results, whereas in “pure” random testing the test inputs are picked at random. In other words, both pure RT and enriched RT, e.g. ART, randomly generates test inputs from the input domain, but enriched RT uses additional guidance to help test case selection rather than randomly selection. For instance, in the case of ART the selection criteria are those to ensure that all test cases are evenly spread over the whole input domain.

In the following, a comprehensive investigation of existing random-directed testing strategies, either pure or enriched, for different application domains is surveyed and also summarized in Table 2.

In one of the preliminary studies [59] on random testing the robustness of Windows NT applications were assessed using RT techniques where it showed that it seems to be effective at locating bugs in real programs running on Windows NT operating systems. Similarly, in [60, 61] random testing was empirically used to test the reliability, assess the robustness, of the applications running on Mac OS X operating system. More specifically, in this study 135 command-line UNIX utilities and thirty graphical applications on Mac OS X were chosen and seeded using random inputs. The results showed that only 7% of the command-line utilities crashed, where the only eight out of thirty of GUI-based applications did not crash and twenty others crashed, and two hung. In another work [62] a tool called QuickCheck was proposed to test the Haskell programs based on the idea of random testing. The case studies presented in this work showed that the given tool can be successfully applied in order to aid the Haskell programmer. T. Yoshikawa et al. in [63] proposed a random test program generator and a Java JIT compiler test system. This test system automatically 1) generates Java class files which are random, executable and finite; 2) inserts codes that output execution-process-dependent results; 3) runs these on the target JIT compiler and other tested Java runtimes; 4) compares results. Besides, random testing also has been applied and used in testing interrupt-driven embedded software [64], PVS (Prototype Verification System) [65], formal software models and induced coverage [66], database systems [67], multithreaded concurrent programs [68, 69], security vulnerabilities [70], .NET architecture components [71], machine learning applications [72], where a sufficient real-world data set that is capable of exercising such applications is absence or quite difficult to be suitably created, and software model checking ranging from simple models obtaining of formal requirements specifications for a given system [66] to very large models, are considered as a set of communicating reactive modules, with presence of non-determinism in program inputs [73] or uniform sampling of traces [74, 75]. Furthermore, recently a great deal of works on random testing of object-oriented application has also been proposed. These works are ranging from complex tools for automated testing of Java such as T2 [76], JCrasher [77], Eclat [78], Jtest [79], Jartege [80], RUTE-J [81], or Korat [82] to profound concepts and definitions, where a great
contribution comes from the works done by I. Ciupa et al. e.g. [3, 4]. All the works and tools presented above employ somewhat pure random strategies for generating and selecting their test inputs.

From the view of enriched random testing, there have been some extensions to the pure random testing algorithm which attempt to maintain the benefits of classical random testing while increasing its efficiency and performance. These extensions mostly target on the random generated inputs to make them more efficient by means of some injected guidance to the selection process. However, after a thorough review of the current literature, domain analysis, we identified five general directions in proposed enriched RT to guide the test case generation or selection, which are **D1**: restricting the regions of the input space in order to avoid redundant and illegal inputs as well as finding useful inputs for more generating [83-89]. **D2**: combining the random and systematic approaches [90]. **D3**: Coverage-oriented approaches [91, 92]. **D4**: spreading evenly the input values over the input domain. **D5**: others, which cannot be located in any of the preceding four directions. The D4 direction represents the basic idea of the Adaptive Random Testing (ART) [93, 94] and quasi-random testing [95], and somewhat the Diversity Oriented Test Data Generation (DOTG) [88], which is the most dominant family of the enriched RT so far. Based on the idea of ART great deals of related algorithms have been proposed. Some of these algorithms are closely related to the ART, however, with slight changes e.g. Restricted Random Testing (RRT) [83, 84] or Ordinary Random Testing [96], while a plenty of them emphasis on the improvement to ART [93] itself.

Mirror ART [97, 98], Fuzzy ART [99], ART by restriction [100], ART by localization [101], ART through dynamic partitioning [102], and ART with CG constraints [103] alleviate the pitfalls of the classical ART algorithms. Further advancement to ART are also provided by lattice-based ART [104], Restricted ART by random partitioning [105], ART by bisection with restriction [106] and localization [107], ART through iterative partitioning revisited [108] and not revisited [109], ART with enlarged and high dimensional input domains [110-112], ART with randomly translated failure region [113], ART using Voronoi diagram [114], ART by balancing [115], Distribution Metric Driven ART [116, 117], ART through partitioning by edge and centre [118], and more recently a new family of ART [119] has been proposed which uses a new test profile called failure driven than uniform distribution or operational profiles used in the original ART algorithms to maximize the effectiveness of failure detection. These new algorithms show better failure detection capabilities in contrast with the corresponding original adaptive random testing and the pure random testing. Furthermore, ART for object-oriented programs also has recently been proposed in [120-122].

<table>
<thead>
<tr>
<th>Work</th>
<th>Application domain</th>
<th>Category of strategy</th>
</tr>
</thead>
<tbody>
<tr>
<td>[59]</td>
<td>Windows NT applications</td>
<td>✔️ Pure</td>
</tr>
<tr>
<td>[60, 61]</td>
<td>Mac OS applications</td>
<td>✔️ Enriched</td>
</tr>
<tr>
<td>[62]</td>
<td>Haskell Programs</td>
<td>✔️</td>
</tr>
<tr>
<td>[63]</td>
<td>Java JIT compiler</td>
<td>✔️</td>
</tr>
<tr>
<td>[64]</td>
<td>Embedded software</td>
<td>✔️</td>
</tr>
<tr>
<td>[65]</td>
<td>Prototype Verification System</td>
<td>✔️</td>
</tr>
</tbody>
</table>
The first column provides the reference numbers of the work. The second column represents the application domains in which random testing has been applied to. Specifically, the main concern of random testing tightly goes to the idea of random generation. It means in order to apply random testing to a given application domain firstly the issue of the random generation must be properly addressed. However, this generation can vary from random generation of simple data to the complex and higher level of abstraction such as rules (e.g. flex, a lexical analyzer generator), objects, and aspects. The ‘theory concepts’ notation indicates that the corresponding work give an improvement and/or extension on the random testing infrastructures. The third column categorizes the works based upon the pure and enriched strategies used in their test generation process. The ‘D1-D5’ notations indicate directions in proposed enriched RT category to guide the test case generation or selection. From Table 2, we can observe that although random testing has been applied to many application domains and paradigms yet but it had not been applied to the aspect-oriented paradigm.

Nevertheless, none of the aforementioned works presented above, either in aspect testing area or random testing area, targets testing of Aspect-Oriented Programs (AOP) with random strategies or any other enriched RT such as ART.

4. The approach

The approach taken in this work is based on AspectJ-like languages and manifested in the form of a framework. The proposed framework consists of five major stages, which outlines the abstract architecture of the approach and its involved components. It consists of five major stages that depict a general macro view of the aspect random testing. All these stages are the target components of automated testing. One of the strengths associated with the framework is, any new random input generation and selection strategy can be easily replaced or current strategy evolved without having influence on the other parts. Figure 1 illustrates this framework towards random testing of aspect-oriented programs.
Figure 1. The framework to random testing of aspect-oriented programs

In first stage, we identify aspects in the target system and determine their run-time coupling with other parts of the system by means of our previous developed tool called AJcFgraph Builder [123]. It helps finding the components or simply classes which are affected by the aspects to result in runtime traceability. Then, we specify the aspect expected behaviors to finally be chosen for testing. In other words, AUT is picked up in this stage.

The aspect language designs can dramatically effect or restrict the testing of aspects comparing to the objects. Due in part to most of AO programming languages such as AspectJ aspects do not have independent identity or existence in the system and cannot be instantiated [23]. More specifically, the base code has no references to the crosscutting concerns (aspect code). When a crosscutting concern needs to be executed, the corresponding aspect(s) will take the control flow and injects the code related to the crosscutting behavior (which is in advice part of a given aspect). Then, it returns the control back to the base code (join point). It means aspects are such static observers in the system that cannot stand on their own. Therefore, to generate and select random input for AUT in aspect-oriented programs (strategy) (1) it can be either started with base code or classes in which they depend directly or indirectly to the AUT (which are determined by means of AJcFgraph tool in first stage) in order to construct the execution contexts for the chosen AUT or using mocking concepts to mimic the behavior of these object and isolate the AUT. Note, these tests eventually exercise aspectual behavior and mostly focus on testing aspectual composition between base code and aspect code, which has been studied before based on the bytecode of both classes and aspects in [124] , this issue is alleviated by using random testing approach proposed in this work because chance selection are replaced by human choices. However, to this end, target objects are randomly generated, with help of randomly called constructor of the resultant classes, and retained in a object repository for reuse (Note: the objects are instance representation of their classes at run time and each of which are associated with some methods). Then, some methods or constructors of the objects are randomly chosen. Finally, for any chosen method or constructor their arguments (if any) based on their
data type, either primitive or reference, are randomly produced to be called on the created objects (random method calls). This is done by means a pseudo-random number generator (as with the classical RT) since the primary objective is to test the aspects (i.e. pointcuts and advice) than objects, we develop an infrastructure by which generated inputs are guided/selected by a join point model. The model helps driving the test cases to purposely exercise the aspect implementation or advice, called testing aspactual behavior, as well as the pointcut functionality which exercise the unique relationship between the aspect code and base code e.g. classes, called testing aspactual composition.

In third stage, the aspects are annotated with their expected behavior, which are going to be checked during runtime execution and asserts what conditions the aspect must meet at certain points of the execution. More specifically, the Pipa [125] specifications are written for aspect under test to provide a means for checking their specification against the test results once they were executed. This takes care of test oracle.

In forth stage, the test code generated for given AUT is automatically performed. In addition, we develop an infrastructure in which the testing process can be continued or recovered in despite of any failure occurring during the test running.

The fifth stage assesses the outcome to determine pass or fail of a given test run. We develop a contract violation checker for embedded Pipa specification to check any violations of contracts/assertions at runtime and consequently revealing the failure-causing inputs in a given test. Moreover, it reports and logs the testing results in which the input test cases are classified into failure-causing inputs (An input is said to be failure-causing if the program or system under test fails for this input) or none, which means passed.

5. Preliminary experimental evaluation

This section briefly sketches the way that the evaluation of the proposed approach may go about however; the actual results of the evaluation is being processed. Specifically, for evaluating and measuring the effectiveness of the proposed testing approach, we conduct some empirical studies, as follows: (1) using simulation and mutation analysis in which some candidate performance metrics such as, F-measure (the number of test cases required to detect the first failure), is the commonly used, P-measure (the probability of detecting at least one failure in a give test set ) or E-measure (the expected/average number of failures detected by the test set) are used to revel the testing effectiveness in terms of failure-detection capability. Moreover, for mutation analysis AspectJ benchmarks (programs) are selected as subject programs for analysis, as well as some tools such as MuJava [126, 127], of course must be extended to fit the properties of AspectJ language since the existence tool is only meant for Java, will be used to automatically generate mutants. For mutants analysis the fault seeding of the aspects is based on a set of AO mutation operators (which perform modifications to the original program to result in occurring a given fault type). (2) the code-based or program-based coverage of the given approach is evaluated, where mostly the control flow (such as: statement and condition criteria) and data flow coverage criteria (such as: c-uses and p-uses) are chosen to be exercised by the proposed approach in order to firstly discover any correlation between the failure-detection capability in first experimental study with the coverage percentage and also comparing the obtained coverage percentage by the other systematic (e.g. structural-based ) approaches.
6. Conclusion and future work

Gradually, the research on software testing has turned their attention to developing automated solutions to different stages of the testing process: input generation and selection, test execution, examining the output of the execution (the oracle), test result and test case classification, and optionally getting feedback from the result analyzing to estimate/improve the quality of the generated tests. This results in emerging of a wide range of testing strategies with different levels of automation for each stage of the testing process. Out of all these stages, possibly the most challenging to automate are the two major testing activities, *input generation and selection* and the *oracle*. On this regard, this paper proposed an automated random-based test generation and test execution approach for aspect-oriented programs. To the best of our knowledge, proposed approach is the first attempt in the community that applies random testing on AOP.

Besides, we are developing an AspectJ automated random testing tool based on the proposed approach in order to firstly put into practice the entire process of automated AOP random testing and secondly to be easily used in our experimental for evaluating the effectiveness of the proposed approach.

The future work for this work includes: 1) assessing the effectiveness and reliability of the proposed strategy 2) discovering the applicability of the ART notion to the aspects 3) designing the corresponding metric model (distance measure), comparing effectiveness of random testing and adaptive random testing for aspects if (2) is proven 4) and evolving the proposed strategy towards a more mature and effective one.

References


Authors

Reza Meimandi Parizi received the B.Sc. in Computer Science from the Shahid Bahonar University of Kerman, Iran, in 2005, the M.Sc. in Software Engineering from the University Putra Malaysia (UPM), Kuala Lumpur, in 2008. In 2008 he enrolled as a full-time Ph.D. candidate at the University Putra Malaysia in the field of Software Engineering. Since 2006, he has been with the Department of Information System at the University Putra Malaysia, Malaysia. His research interests include Aspect-Oriented Programming (AOP), Aspect control flow analysis, Aspect testing, Random testing, Automated software testing with more emphasis on Aspect-Oriented Software Development (AOSD).

Abdul Azim Abdul Ghani received the B.Sc. in Mathematics/Computer Science from Indiana State University in 1984 and M.Sc. in Computer Science from University of Miami in 1985. He joined University Putra Malaysia in 1985 as a lecturer in Computer Science. He received the Ph.D. in Software Engineering from University of Strathclyde in 1993. He is an Associate Professor and the Dean of Faculty of Computer Science and Information Technology, University Putra Malaysia. His research interests are Software engineering, Software measurement, Software quality, and Aspect-orientation.

Rusli Abdullah received the B.Sc. and M.Sc. degrees in Computer Science from University Putra Malaysia in 1988 and 1996 respectively, and Ph.D. in Software Engineering from Technological University of Malaysia in 2005. His research interests include information system, knowledge management and software engineering. He is now with the Faculty of Computer Science and Information Technology in University Putra Malaysia as a full-time lecturer.

Rodziah Atan received the B.Sc. degree in Computer Science in 1996 from Agricultural University of Malaysia and M.Sc. in 1998 from University Putra Malaysia. She completed her Ph.D. from the same university in 2005. She has been supported by the government of Malaysia and the University’s Young Lecturer Scheme (SLAB). Her field of interest is software process and business process modeling and pursuing for new knowledge in bioinformatics visualization tools.