Int J Softw Tools Technol Transfer (2017) 19:743–761
DOI 10.1007/s10009-016-0432-3

REGULAR PAPER

RAMBUTANS: automatic AOP-specific test generation tool

Reza Meimandi Parizi[1] · **Abdul Azim Abdul Ghani**[2] · **Sai Peck Lee**[3] · **Saif Ur Rehman Khan**[3]

Published online: 20 July 2016

Abstract Aspect-oriented programming (AOP) is a programmatic methodology to handle better modularized code by separating crosscutting concerns from the traditional abstraction boundaries. Automated testing, as one of the most demanding needs of the software development to reduce both human effort and costs, is a delicate issue in testing aspect-oriented programs. Prior studies in the automated test generation for aspect-oriented programs have been very limited with respect to the need for both adequate tool support and capability concerning effectiveness and efficiency. This paper describes a new AOP-specific tool for testing aspect-oriented programs, called *RAMBUTANS*. The *RAMBUTANS* tool uses a directed random testing technique that is especially well suited for generating tests for aspectual features in AspectJ. The directed random aspect of the tool is parameterized by associating weights to aspects, advice, methods, and classes by controlling object and joint point creations during the test generation process. We present a comprehensive empirical evaluation of our tool against the current AOP test generation approaches on three industrial aspect-oriented projects. The results of the experimental and statistical tests showed that *RAMBUTANS* tool produces test suites that have higher fault-detection capability and efficiency for AspectJ-like programs.

✉ Reza Meimandi Parizi
rparizi@nyit.edu; r.m.parizi@ieee.org

Abdul Azim Abdul Ghani
azim@upm.edu.my

Sai Peck Lee
saipeck@um.edu.my

Saif Ur Rehman Khan
saif_rehman@siswa.um.edu.my

[1] School of Engineering and Computing Sciences, New York Institue of Technolgoy (NYIT), Nanjing Campus, Nanjing, China

[2] University Putra Malaysia, Serdang, Malaysia

[3] University of Malaya, Kuala Lumpur, Malaysia

Keywords Software testing · Automated test generation · Testing tool · Aspect-oriented programming · Object-oriented programming · AspectJ

1 Introduction

Aspect-oriented programming (AOP) [1,2] has been proposed as a methodology for handling the modularization of source code by re-defining the abstraction level and reducing the scattering and tangling of crosscutting concerns [3]. To achieve this, AOP introduces a modular construct, the *aspect*, which is basically used to encompass crosscutting concerns (i.e., requirements that are spread across or tangled with other requirements, such as quality attributes) in applications.

An analysis of the literature uncovers that aspects have been used crosswise over different fields of exploration and spaces, going for the improvement of customary methods, for example, in reference architecture [4] and architectural invariants and decisions [5,6], embedded software testing and hardware verification [7–9], non-functional requirements' classification [10], traceability improvement [11], software product line implementation [12], and component-based development [13]. The outcomes accomplished from these studies have propelled the utilization of aspects and have demonstrated the handiness of AOP in software and applications engineering.

While the utilization of aspects is by all accounts helpful and offers advantages over traditional strategies in diverse contexts, unique characteristics [14] of AOP and its effect on testability and viability have made its testing more challenging. To understand the advantages of AOP and to build its reception, programs (i.e., aspects) created by this programming paradigm ought to be effectively tested, with consideration paid to all of its characteristics, i.e., augmentation of new programming constructs and properties [14] for the separation of concerns.

Automated testing has been a huge zone of enthusiasm in software testing research and has developed incredibly in recent years. Nonetheless, it must be called attention to the literature uncovers that there is relatively little work [15] on testing of AOP and not very many studies on automated testing of AOP, none of which has been to our knowledge generally accepted [16]. From a specialized perspective, the current approaches [17–20] on automated testing of AOP do not have an AOP-specific spotlight on the test generation process and it is more founded on the knowledge of base code (i.e., classes). This, as a result, can affect the effectiveness of tests, in which the crosscutting concerns actualized in aspects might not be effectively tested and expand the quantity of tests required. In addition, providing no automated test code generation, and above all, the absence of proper practical tools (as they are restricted in their functionality and for the most part not circulated beyond the researchers involved in the work) is significant issues connected with the current studies.

Random testing (RT) [21] is a dynamic research area, and has been a widely used technique in hands-on settings [22] due to its benefits, e.g., fault-detection at low cost, execution speed, absence of human bias, and above all the opportunity for automation that is sort of difficult in other techniques. The utilization of random-related testing techniques for test generation purposes has been a prospering enthusiasm for many researchers and programmers. Numerous studies and helpful approaches [23–25] have proposed the usage and execution of random testing as their center parts, which seem to beat efficient systematic test generation [25]. Notwithstanding the previously stated benefits of RT, the value of random testing is said to be expanded, as it is more likely to test the programs in novel routes (because of the randomized nature) and reveal faults that were missed during development [26]. In this admiration, the idea behind random testing can be beneficial, appealing, and offer much promise with respect to AOP test automation problems, since current research on testing of AOP, with respect to automated testing, has not been sufficiently performed and is still in its outset.

This paper describes a new AOP-specific tool for testing aspect-oriented programs, called *RAMBUTANS* tool. The level of testing that *RAMBUTANS* aims at is aspect level, which can be seen as a sort of unit testing that has a crosscutting impact. The *RAMBUTANS* tool uses a directed random testing technique [27] that is especially well suited for generating tests for aspectual features in AspectJ, which represents the most investigated AOP language. The directed random aspect of the tool is parameterized by associating weights (as listed in Table 1) to aspects, advice, methods, and classes by controlling object and joint point creations during the test generation process, rather than leaving everything to chance, i.e., pure random testing. This directed feature would be useful for stress testing, in case if one wants to more intensively test a given advice or aspect. The number of tests executed on a given advice can play an important role to reveal its faults as various join points (and their associated data from the execution context of the join points) can be picked out, resulting in behaviorally diverse tests.

Nevertheless, we present a comprehensive empirical evaluation of our tool against the current AOP test generation approaches on three industrial aspect-oriented projects. The results of the experimental and statistical tests showed that *RAMBUTANS* tool produces test suites that have higher fault-detection capabilities and are more efficient. Hence, the proposed tool could be worth utilizing as a compelling and effective AOP-specific test generation tool.

The rest of this paper is structured as follows: Sect. 2 presents related work and background. Section 3 gives the details of the tool, including its underlying technique, functional and architectural features, as well as inner workings details and the technologies that have been used. Section 4 discusses the empirical assessment and results, which is followed by implications and limitations in Sect. 5. Section 6 gives the conclusion and future work.

2 Background and related work

2.1 AOP and AspectJ

Aspect-oriented programming (AOP) [1] emerged as a solution to enhance software modularization. It was highly motivated by the fact that traditional paradigms, such as procedural and object-oriented programming, could not effectively handle *crosscutting concerns*. Non-crosscutting concerns, on the other hand, compose the *base code* of an application and comprise the set of functionalities that can be modularized within conventional implementation units (e.g., classes and data structures). AOP is primarily based on the idea of separation of concerns (SoC), which claims that computer systems are better developed if their several concerns are specified and implemented separately. Such separation in AOP is achieved by means of new conceptual modular units called *aspects*, which encapsulate code that usually appears either scattered over several modules in a system or tangled with code that realize other concerns [28].

In an AO program, the behavior (which technically represents a crosscutting concern) implemented within an aspect runs when specific events, the *joint points* (JPs), occur during the program execution. Typical examples of JPs are a method call, a method execution, or a field access. In AspectJ [29], which is the most popular Java-based and commonly used AOP language, a *pointcut* expression (or pointcut designator PCD) selects a set of JPs by means of declarative expressions. A PCD is generally formed by patterns (e.g., method signatures) and predicates. The JP models together with the PCD implement a quantification mechanism that enables crosscutting behavior to run at several JPs during the software execution (i.e., to inject the intended behavior). The PCDs are bound to *advices*, which consist of method-like portions of code that implement crosscutting behavior [28]. For a complete list of AspectJ features, the reader may refer to the AspectJ project website.

2.2 Related work

In the literature, very few numbers of different automated test generation approaches for AO programs have been proposed. To the best of our insight, there are as of now four noteworthy approaches for AOP testing. The approaches are: (1) Wrasp [17], (2) Aspectra [18], (3) Raspect [19], and (4) EAT [20]. The following sections briefly discuss the underlying working of these approaches.

2.3 Wrasp

Wrasp approach [17] supports unit and integration-level test generation for AspectJ programs by emphasizing on *aspectual behavior* testing. The Wrasp approach specifically focuses on adoption of unit-testing frameworks (i.e., JamlUnit [30] and AJTE [31]) and reusing existing Java supported random test generation tools (e.g., JCrasher [32] or Parasoft Jtest [33]). Both tools randomly generate method sequences based on the given class bytecode. The Wrasp approach applies two steps for random test generation: (1) to take the aspect class or woven class bytecode for supporting unit or integration-level testing respectively and (2) then forwards the Java bytecode (i.e., aspect class or woven class) to JCrasher [32] or Parasoft Jtest [33] tools for automatic generation of test cases. Furthermore, this approach employs a wrapper mechanism, which enables it to handle the leveraging issues across the Java supported random test generation tools.

In this approach, however, it is not clearly expressed which existing OO tool was utilized as a part of request to create the test inputs and especially which could give the best results in AO context, in terms of effectiveness (no empirical study was presented). In addition, the required effort and cost of the application of random testing of OO programs to AO programs were not talked about.

2.4 Aspectra

Aspectra approach [18] works similar to Wrasp and automatically generates tests that are able to exercise the *aspectual behavior* of a given AspectJ program. This approach also employs a wrapper mechanism to handle the aspect-weaving issues for test generation. Consequently, it enables a developer to provide base classes of an aspect by developing a wrapper for every single woven class. For the given woven class, this approach automatically generates a wrapper class. Next, Aspectra forwards the wrapper class to current test generation tool (i.e., Parasoft Jtest [33]), which automatically generates test cases for the given woven class and considers the wrapper class as the class under test. Finally, Parasoft Jtest tool automatically produces different data values of all parameters of a targeted method in a particular class.

Aspectra mainly employs two coverage measurements (i.e., aspectual-branch coverage and interaction coverage) for test selection purpose. The interested readers may consult reference [18] for more information about the employed measurements.

Parasoft Jtest utilized as a part of Aspectra is equipped for creating default values for primitive-type arguments and produces random method sequences (the sequence of method calls is referred to as *test sequence*) which incite uncaught runtime exceptions. At the same time, it also tries to maximize statement coverage or branch coverage, contingent upon the design by the tester.

Nonetheless, Parasoft Jtest regularly produces the same test arrangements and acquires no test groupings that accomplish full coverage. Moreover, Parasoft Jtest makes numerous repetitive test sequences, which may likewise exercise non-public methods. In our perspective, these disadvantages coming from Parasoft Jtest influence the produced test suite quality of Aspectra. This is for the most part because of some limitations forced on the test input generation as the tool only supports default or random data generation for targeted based code. Besides, the wrapper-synthesis mechanism (as contended by the authors) is useful more in test generation when advice has call pointcut type and there are no call sites of its advised methods in the code base. This might not be a typical case in AspectJ programs, as it is quite often than other kinds of pointcuts that are used in the program.

2.5 Raspect

Raspect [19] is a broadly similar approach to Wrasp and Aspectra to generate tests and additionally detect the redundancy from the generated test cases of AspectJ programs. In other words, Raspect is not a tool to only generate tests, but it also detects redundant tests in the process of testing. The motivation behind is that Wrasp or Aspectra generation tools normally generate an extensive amount for test inputs that might hold an incredible number for unimportant tests, which are not able to expose faults and hence unable to exercise the new behavior of the class under test.

The Rostra framework [34] reveals redundant unit test cases on object-oriented contexts. The Raspect approach extends the Rostra framework by determining and removing the generated redundant test cases, which ultimately helps in three types of unit testing for AspectJ programs.

The Raspect approach is capable of generating, exposing, and removing three types of unit tests for woven classes by employing current random test input generation tools. However, it does not focus on how to generate integration tests effectively for testing the integration of these three types of AspectJ program units.

2.6 EAT

EAT [20] approach depends on evolutionary testing for automated generation of test cases for AspectJ program. The EAT approach automatically generates test data for given aspects from the base classes, which is similar to Aspectra [18]. However, the generated tests might lack in exercising the aspectual composition behavior. This approach consists of four major components, to test aspectual behavior, including (1) aspectual-branch identifier to identify branches inside aspects of AspectJ programs, which are the coverage targets of the approach; (2) relevant-parameter identifier to identify only those relevant parameters of the methods of the base classes for covering a target aspectual branch; (3) evolutionary tester, which is the key component that forms the test generation technique; and (4) finally aspectual-branch-coverage measurer component to measure the coverage of aspectual branches by means of aspectual-branch coverage metric.

Despite the fact that this approach appeared to have a superior technique in test generation and to some degree outperformed the techniques used in Aspectra or Wrasp, it only spotlights on creating test information that accomplishes aspectual-branch coverage (i.e., control flow-based). In our perspective, considering alternate sorts of coverage measurements as selection criteria would be useful in exposing faults related to interactions of aspect and base code. Moreover, the fault-detection ability (with respect to AOP-specific faults presented by AO fault models, such as [35]) of EAT approach is not discussed and compared with existing ones.

In prior work, we have reported the results of a hypothetical assessment for these current AOP testing approaches described in [36]. In this past work, we have studied the automated AOP testing along three levels of automation, including test generation and selection, oracle, and test execution by identifying the current tools or methods in each level. As a supplement to this work, we directed an expansive scale experiment to look at these four existing automated AOP test generation approaches (as mentioned above). The results of comparing the approaches have been accounted for in [16].

These assessments (both theoretical [36] and empirical [16]) showed that there is an absence of AOP-specific spotlight on the test generation process of the existing approaches, as it is more founded on the knowledge of base code (i.e., classes which form object-oriented parts) rather than aspects themselves. In particular, it was found that the current approaches do not provide proper practical parts/tools (as they are restricted in their functionality and generally not dispersed beyond the researchers involved in the initial work). Inspired by these issues, we were motivated and gained methodological ideas towards proposing the new tool in this research.

3 *RAMBUTANS* tool

The main contribution of this paper is to show that it is possible to provide tool support for aspects using idea of directed random testing, thus making steps of test generation automatic. The result of this work is a tool, called *RAMBUTANS*, which gives support to all of test generation steps proposed in our own technique [27].

This section provides an overview of the tool by describing its underlying technique, functional and architectural features, as well as the technology and inner workings that have been used. The preliminary version of the tool and its detailed documentations are available at: http://fsktm.upm.edu.my/serg/RAMBUTANS.

3.1 Underlying technique

The technique consists of four major components (shown in green boxes in Fig. 1), considered as its essential ingredients, where their interactions and/or combinations form the basis of *RAMBUTANS*. Note, refer to Table 1 to get insight into terms, notations, and acronyms used in the technique.

As indicated by Fig. 1, *RAMBUTANS* begins producing test data by selecting a random aspect called *AUT* (which has the highest $P_{\text{selection}}(AP)$) from the aspect pool created by aspectual-class extractor component. Given this aspect, a random advice is gotten(with the highest $P_{\text{selection}}(AC)$) from advice collection associated with the AUT (i.e. AUT_{AC}). At that point, this random advice, *a*, is fed to the crosscutting dependency identifier component to recognize every one of the classes influenced by this advice, and correspondingly constructs its advised class collection (ACC). After this, a random class, *C*, and a random advised method, *m*, are acquired from ACC and AMC with thought of the probabilities of $P_{\text{selection}}(ACC)$ and $P_{\text{selection}}(AMC)$ respectively.

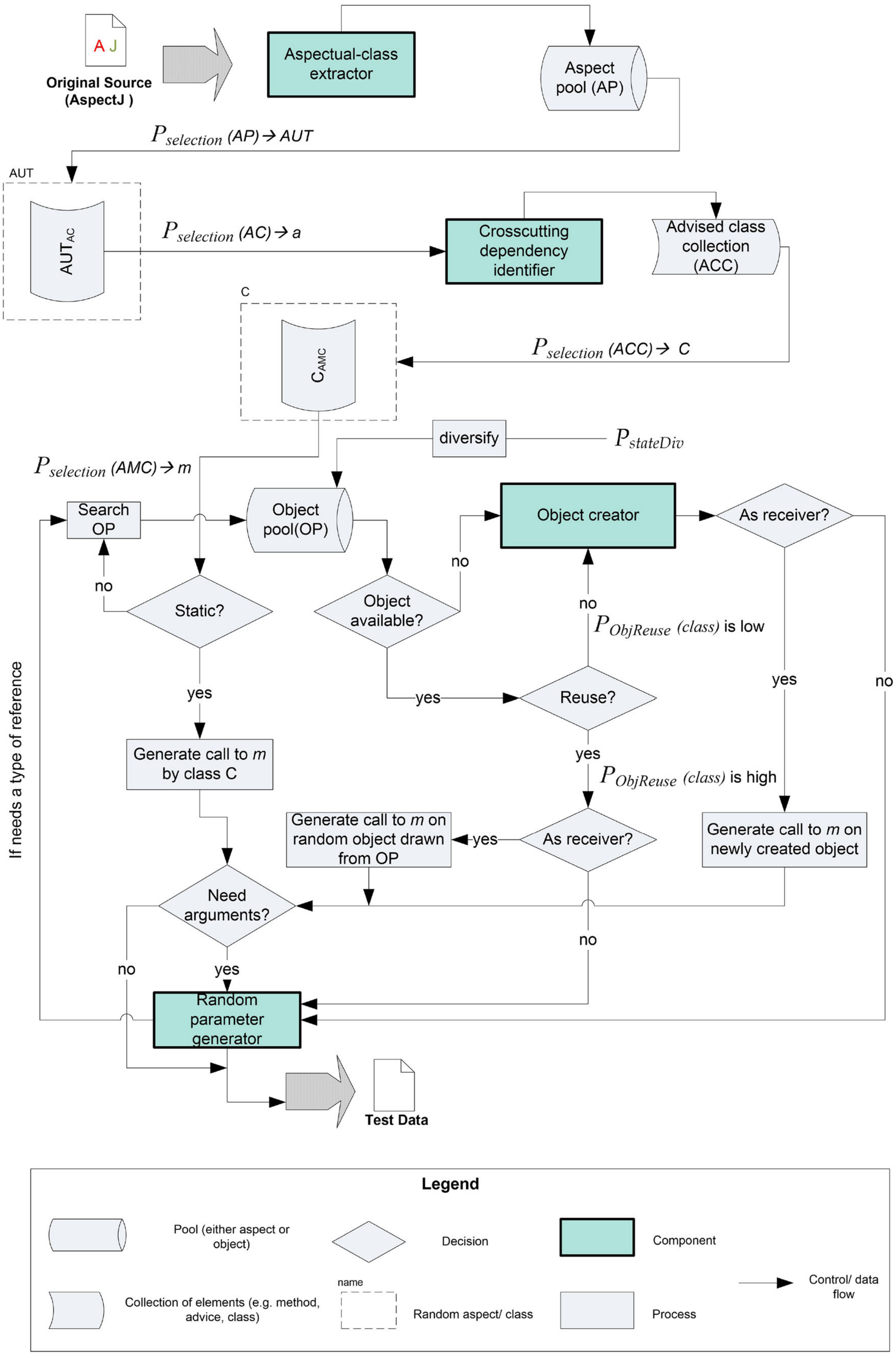

A J
Original Source (AspectJ)
Aspectual-class extractor
Aspect pool (AP)
$P_{selection}$ (AP) → AUT
AUT
AUT_{AC}
$P_{selection}$ (AC) → a
Crosscutting dependency identifier
Advised class collection (ACC)
C
C_{AMC}
$P_{selection}$ (ACC) → C
diversify
$P_{stateDiv}$
$P_{selection}$ (AMC) → m
Search OP
Object pool(OP)
Object creator
As receiver?
Static?
Object available?
no
yes
$P_{ObjReuse}$ (class) is low
Reuse?
$P_{ObjReuse}$ (class) is high
Generate call to m by class C
Generate call to m on random object drawn from OP
As receiver?
Generate call to m on newly created object
Need arguments?
Random parameter generator
If needs a type of reference
Test Data
Legend
Pool (either aspect or object)
Decision
Component
Collection of elements (e.g. method, advice, class)
name
Random aspect/ class
Process
Control/ data flow

Fig. 1 *RAMBUTANS* technique

Table 1 List of notations and acronyms used in *RAMBUTANS*

Acronym/notation	Description
AUT	An aspect under test
OP	Object pool (as repository): Keeps a pool of instances available for testing a given AO program. It contains classes instances related to an AO program during the testing session. All objects created as part of the test generation are gradually stored in this pool
AP	Aspect pool (as repository): Keeps all aspects existing in an AO program
AC	Advice collection
ACC	Advised class collection
AMC	Advised method collection
$\mathrm{AUT}_{\mathrm{AC}}$	Refer to advice collection of AUT aspect
C_{AMC}	Refer to advised method collection of class C
$P_{\mathrm{selection}}\,(\mathit{repository}) - > \mathit{element}$	Probability that a given element possesses to be selected for testing. Thus, the *element* will be selected for testing, if it possesses the highest weight among the other elements in the same *repository*
$P_{\mathrm{ObjReuse}}\,(\mathit{class})$	Probability of creating a new object and/or reusing in the presence of existing objects of a given type. It is defined with respect to the number of existing instances of a given class in the pool to the total number of instances of all types
P_{stateDiv}	Probability which is defined to indicate how frequent (determined by a user-settable timing value) the process of calling the methods of a object should be to modify its states during test generation process

In the event that m is a static method, then a call to m on its individual class C is produced, as the static methods can be called without making any object of their classes. Or else, if the method is not static (i.e., public non-static), the object pool (OP) is initially checked for any accessible object of the desired type to receive the method call m (i.e., known as receiver object). On the off chance that the object of the desired type is not available in the object pool or available, but rather the technique with thought of P_{ObjReuse} chooses to not reuse (in light of the fact that P_{ObjReuse} is resolved as low), a target object of the given type will be randomly developed to receive the call. This is performed with the assistance of the object-creator component. Besides, the recently created object that was rare in the pool would be additionally added to the OP. Despite what might be expected, if the object of the sought type is available in the object pool and the technique with thought of P_{ObjReuse} consents to reuse (in light of the fact that P_{ObjReuse} is resolved as high), a random object of the wanted type is browsed from the object pool so as to get the method call m.

At last, the created call to m is affixed (in both the cases of static or non-static) to the test data file with its input parameters/arguments (if m takes any) randomly produced by random parameter generator component. This implies that the input parameters of each method call prior to its addition are created randomly according to the parameter data types. In particular, the random parameter generator component may need to generate a reference data type for any input parameter (not a receiver object) of a given method call. In such manner, such as the receiver object generation, the same procedure is applied to reuse a current object (via searching the pool) making another one by calling the object-creator component. Be that as it may, for recognizing purposes, whenever an object is either created or browsed from the pool, *RAMBUTANS* checks its essential use to figure out if it is a receiver object for a method call or an input parameter to a method or constructor (i.e., the ‘As receiver?’ decision in the design introduced in Fig. 1).

At whatever point necessary, the object pool is diversified by the technique according to the frequency controlled by P_{stateDiv}. At that point, the same procedure is repeatedly carried out to consider producing tests for the remaining elements of the AspectJ program under test. In a key manner, the test data created by *RAMBUTANS* are that of random generation of join points to stimulate/trigger advice and pointcuts in a random way.

Note, the presentation of details and separate algorithms for the major components utilized in the technique are given in Appendix A.

3.2 Architecture

The architecture of the prototype tool developed for this research project is logically structured in three layers, including presentation, persistence, and most importantly the application layer that contains the core module comprised of four major components.

Figure 2 outlines the layered architecture of the tool. The architecture is somewhat adopted from a reference architecture for software testing tools called *RefTEST* [37], which

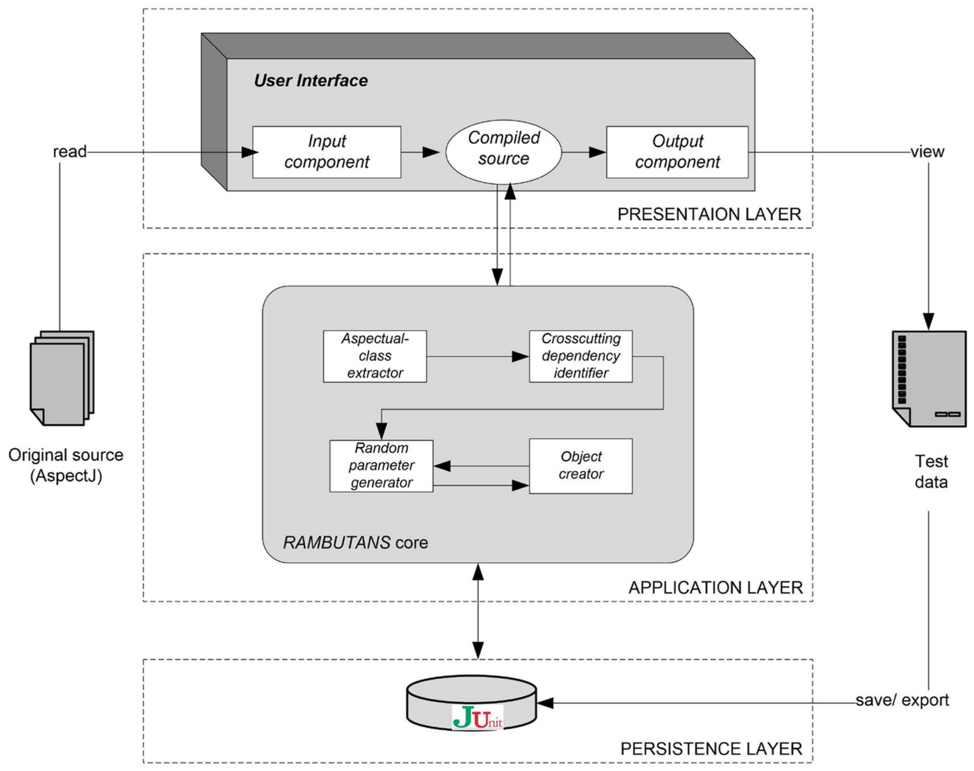


Fig. 2 *RAMBUTANS* tool architectur

helped us to structure the tool in terms of functionalities and component interactions.

According to the architecture, the system takes AspectJ source code as input and generates a set of test data as output. Concerning the presentation layer, the input component reads in and verifies the AspectJ sources. The output component allows the user to view, save, or export the generated tests to JUnit files. These two components are generic in the sense that they are independent of any specific AOP environment. The *RAMBUTANS* core module in application layer performs test generation, i.e., it hosts the main tool. The core module is comprised of aspectual-class extractor, crosscutting dependency identifier, random parameter generator, and object creator as its major components. Aspectual-class extractor component is a source file finder that differentiates all Java (i.e., classes) and AspectJ source files (i.e., aspects) in an AspectJ program. This component runs a preprocessing step, whose output is a list of separated aspects. The component of crosscutting dependency identifier recognizes the elements that have been advised by a given aspect, generally, all the advised join points, e.g., methods, captured by the associated pointcut(s). The random parameter generator component is responsible for generating input parameters' values for advised join points, such as methods and constructors. Object-creator component is responsible for constructing the new objects, generally reference types in the course of test generation.

Finally, persistence layer contains the file system and the database-related procedures and functionalities that can be invoked by the components in both application and presentation layers, whenever is needed.

The given tool has been designed in an object-oriented manner and has been implemented using Java programming language, based on the NetBeans platform. It tests every advice of the aspect under test independently. For each advice, it produces a collection of test data and exports the tests as JUnit test cases, thereby completely automating test generation of AspectJ aspects.

After identifying all the procedures and components that are needed by the systems and how they interact (as presented in the tool's architecture), the next step is to specify the internal algorithm of each component separately. Algorithms are presented in Appendix A.

3.3 Details of implementation (inner workings)

In this section, we briefly explain the implementation details of each component and how other tools/plug-ins were modified or put together to make the software performs test generation in the prototype tool.

Given the understanding of components' functionalities, to implement **Aspectual-class extractor**, we have used pattern-matching mechanism with regular expressions to identify/extract aspects and the advice blocks in AspectJ files through parsing the source code. This in turn can be seen as a kind of aspect mining.

The underlying technique used to implement the **Crosscutting dependency identifier** is based on the idea of providing metadata applied to advice of aspects in the form of annotations supplied with the input source. In this case, the annotations would represent the correct behaviors of pointcuts associated with advice in terms of advised elements.

Finding crosscutting dependency can be viewed as a kind of accessing and detecting the advised join point information at weaving time. For instance, finding the advised classes affected by a given aspect or when a Java method is affected by an advice. Obtaining such information is challenging and would not be trivial, as this information is not preserved in an easily accessible way in woven classes. In addition, the aspect itself does not originally store a list of places, where it applies.

On the other hand, there are Eclipse plug-in tools, such as AspectMaps, which is similar to the Visualiser, and Cross References (a.k.a as "XRef") provided by AspectJ Development Tools (AJDT) to provide a means to get the crosscutting dependency information at compile time. However, none of these tools produces the information in a more easily digestible way for use with other providers. Moreover, these tools' focus is more on visualization of the woven parts than giving absolute crosscutting dependency information and accessible through well-defined APIs. Thus, it can be said that there is currently no standalone tool to give this information in a convenient way. This motivated us to figure out an efficient technique based on the idea of providing annotation files in this version of implementation.

In this technique, to recognize the affected join points dynamically (i.e., advised methods/constructors) by advice of a given aspect, the following two steps are performed:

1. For each aspect, we create a .txt file that will attach crosscutting dependency annotations to the input AspectJ program to represent the name of affected classes and methods. This annotation is applied to any advice block that is to be made available for testing using *RAMBUTANS*. In this light, these annotations are seen as runtime processing types that are used by the underlying technique to help finding crosscutting information efficiently. The syntax for the annotation file takes the form as shown in the following:

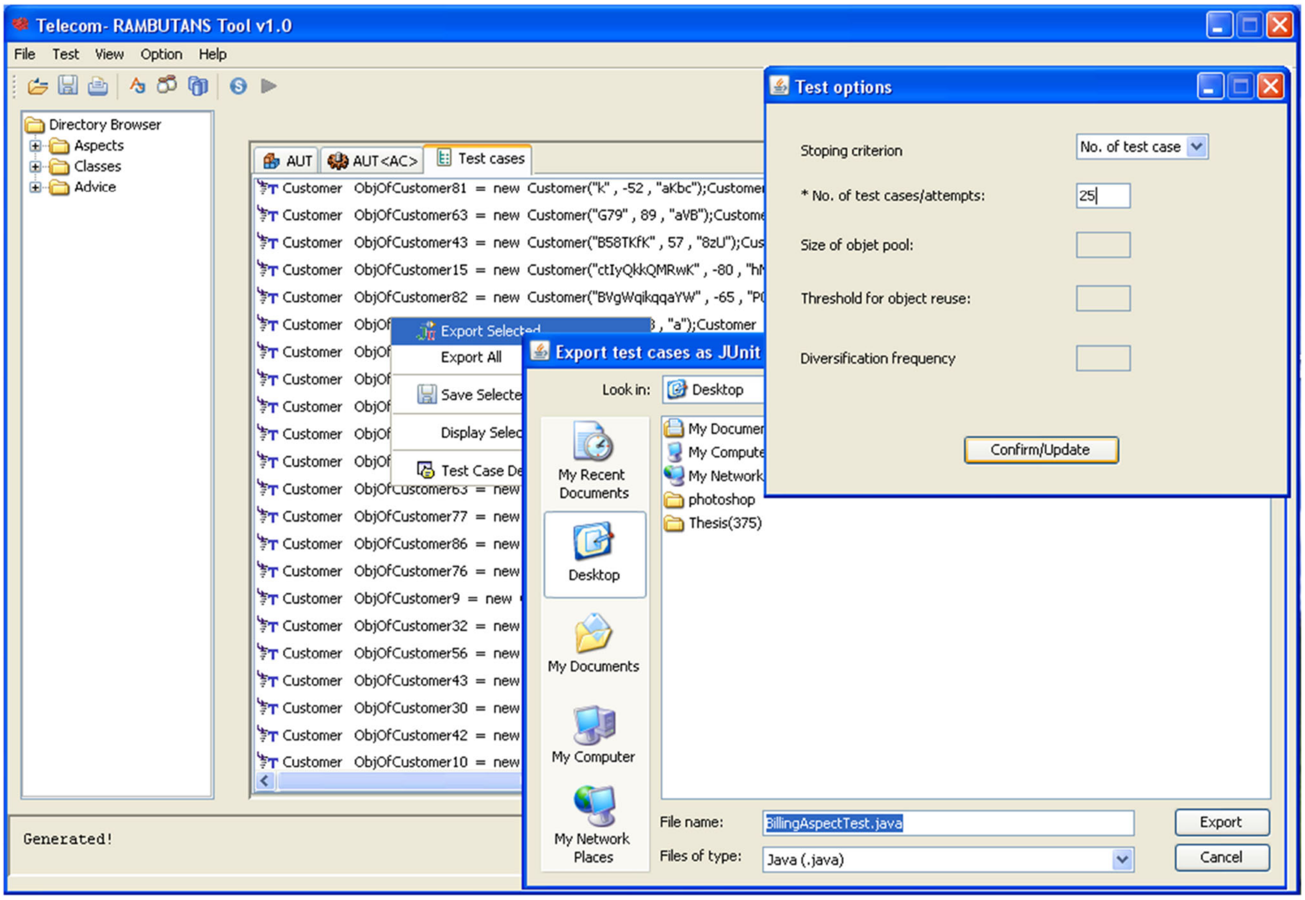
File name:
AspectName_Annotation.txt
Body:
If the join point were a method:
`@advice[advice_Number]:Class_Name(Method_Name);`
If the join point were a constructor:
`@advice[advice_Number]:Class_Name(Class_Name);`

advice_Number is a numerical number equal or greater than *0*, but less than *n*, where *n* is the number of advices of the aspect. From a technical point of view, the *advice_Number* represents a top–down position assignment, in which the ascending order of advice presented in the aspect file determines its value. It is important to note that the body of the file can consist of many lines if there is an advice that is associated with more than one advised class or method.

2. At this step, the aim is to turn this annotation into some kind of crosscutting dependency information that travels alongside the code and could then be loaded by the system later (to discover what was woven and where). To this end, the tool interprets the file by means of a getAnnotation() method of DependencyIdentifier.java class. In this method, similar to the previous component, regular expressions are used to construct the patterns to read the annotation file content, according to its syntax. Once the file is read, the tool stores the advised class collection and advised method collection of each aspect in a separate Java ArrayList.

It is the responsibility of the testers/users to create the annotation files for aspects. This in turn, however, might be the sort of a questioning matter that needs further explanation. As such, we discuss the pros and cons of providing such files. From a positive perspective, although this matter might impose relatively little manual effort to the users, it is worth it as it is very efficient to provide human-checked crosscutting dependency, given test engineer correctly written the crosscutting dependency information in the file.

As opposed to this perspective, there is a threat of "what will be the consequence in the rest of the process if the file is incorrectly written by the test engineer". Of course, this situation might rarely happen in testing high risk software systems, but obviously an inaccurate file will lead to undesirable tests. Now, the question is how to mitigate this situation and the risk of getting inaccurate information in the test generation process? As a measure to address this unwanted situation, we had devised a custom attribute in the tool that acts as an implicit traceability link to check the discrepancies in the annotation file and the crosscutting dependency information obtaining from an Eclipse (AspectMaps) plug-in running upon the pointcut's patterns associated with the advice. After an attribute is associated with an AO program entity, the attribute is implicitly run in the background as a plug-and-play tool. This means, it works out-of-the-box without need to understand any complexities behind it or to make any changes or adjustments to your system. The obtained information in the attribute is inspected in the embedded information in the body of the file using the static call graph analysis and Java reflection technique in relation with extracted production classes and aspects. The status of the attribute will be then used, *prior to test generation*, to notify users if the file's information is incorrect or misleading, in other words, if tool found discrepancies. Having said that utilizing the aforementioned custom attributes the risk of getting into undesirable tests derived from wrong information in the file could be nearly minimized.



```
Customer  ObjOfCustomer81  =  new  Customer("k" , -52 , "aKbc");Customer
Customer  ObjOfCustomer63  =  new  Customer("G79" , 89 , "aVB");Custome
Customer  ObjOfCustomer43  =  new  Customer("B58TKfK" , 57 , "8zU");Cus
Customer  ObjOfCustomer15  =  new  Customer("ctIyQkkQMRwK" , -80 , "hN
Customer  ObjOfCustomer82  =  new  Customer("BVgWqikqqaYW" , -65 , "P(
Customer  ObjOf                                   3 , "a");Customer
Customer  ObjOf
Customer  ObjOf
Customer  ObjOf
Customer  ObjOf
Customer  ObjOf
Customer  ObjOfCustomer63  =  new
Customer  ObjOfCustomer77  =  new
Customer  ObjOfCustomer86  =  new
Customer  ObjOfCustomer76  =  new
Customer  ObjOfCustomer9  =  new
Customer  ObjOfCustomer32  =  new
Customer  ObjOfCustomer56  =  new
Customer  ObjOfCustomer43  =  new
Customer  ObjOfCustomer30  =  new
Customer  ObjOfCustomer42  =  new
Customer  ObjOfCustomer10  =  new

Generated!
```

Look in: Desktop
My Documer
My Compute
My Network
photoshop
Thesis(375)

File name: BillingAspectTest.java
Files of type: Java (.java)

Stoping criterion: No. of test case
* No. of test cases/attempts: 25
Size of objet pool:
Threshold for object reuse:
Diversification frequency

Fig. 3 Sample screenshots

Random parameter generator component benefits from a rich toolkit provided by Java language, to be exact, by a class named as Random. This class is able to generate many kinds of random values of different types with flexibility of adjusting different seeds, scaling and translating the random number generator's sequences. Furthermore, to obtain the information regarding the methods of an advised class, the modifiers of a given advised method, and getting the advised method or constructor's parameters or list of the advised class constructors, we used Java reflection API to examine and manipulate internal properties of a given piece of program from within itself.

Similar to the previous component, for the purpose of random object generating, we used the same toolkit and Java reflective API to provide random-related information for object creation in **Object-creator component**.

3.4 Execution of the *RAMBUTANS* tool

The *RAMBUTANS* tool tests each advice of the aspect under test independently. For each advice, it produces a collection of test data and exports the tests as JUnit test cases. The tool additionally handles the generation of a test case for an advice woven in a method that calls a method in another class. For instance, if the advice is woven before method *m* of class A and that this method calls method *x* of class B, tool generates an instance of B to have a complete test case that tests the advice woven on *m*. In this manner, the tool analyses the methods in which advice are woven to consider these cases, thereby fully automating test generation of AspectJ aspects.

As mentioned earlier, to make the tool more available in broader term and potentially of use by developers and/or testers, generated test data are made to be applicable with the family of XUnit frameworks (most successful and related one to our work is Junit). However, such frameworks do not help with test generation, and thus, they require the configuration of the JUnit test classes to embed test data, which might be produced either by an automated strategy or manually by testers. *RAMBUTANS* tool provides an automated feature in which the generated tests can be selectively exported as JUnit test cases/classes. As an advantage, this feature would provide support for regression testing, as well as for incorporating manually created test data or test oracle with automated generated ones.

Figure 3 shows a sample screenshot of the tool. The aspect under test (i.e., AUT) is called Telecom. A typical

Table 2 Key properties of the case studies

Projects	# Classes	# Aspects	# LOC	Description
`Health Watcher (HW)`	100	13	4890	A typical Web-based information system, which allows people to collect and manage health issues and complaints [39]
`AJHotDraw`	279	31	18,450	It is a two-dimensional graphics framework for structured drawing editors
`iBATIS (iB)`	180	38	9974	A Java-based open source framework for object-relational data mapping [40]
Avg.	186.33	27.33	11,104.66	

test generation session begins with the user opening a Zip input file contains an AspectJ program. The tool first compiles the source code with the built-in compiler and then loads classes and aspects within the input file. The content of the file can be viewed from the view button on the toolbar of the user interface or optionally from the view menu if desired. To start generating tests, the user should simply click the select toolbar button. As a result, a random selected aspect and its respective advice collection are shown in *AUT* and *AUT* $< AC >$ tabbed panes, respectively. The AUT $<$ AC $>$ pane contains all advice of the selected aspect. Then, the user should click the start toolbar button. This brings up another window titled test options to prompt the user to key in or confirm the user-settable options, specifically the number of test cases to generate (in this example, the user has set the number of test cases to 25). As soon as the options are set, the tool starts generating the number of required test cases. *RAMBUTANS* tool generates tests for each aspect one at a time. After generating, the generated tests are shown in the *Test cases* tabbed pane. For each generated test case, the user can then choose to view display the selected test and test case description/history or to perform save selected or export selected by selecting an item from a pop-up menu triggered by right click mouse. To keep a permanent record of the entire generated test cases/data, user can click save toolbar button or the equivalent menu option. Tests are saved as a text file. Whenever needed, the user can utilize other functions of the tool, such as view aspect pool, view object pool, print tests, tutorial, and help.

4 Empirical assessment

The purpose of this section is to empirically assess the proposed tool in terms of fault-detection capability and its efficiency to get a full picture of how the performance of *RAMBUTANS* is correlated with other peer approaches. We, therefore, designed our study around the following research questions (RQs):

RQ1 How does the use of *RAMBUTANS* tool impact the ability to detect faults, compared to the other approaches, in the program under test? (regarding to effectiveness)

RQ2 How does the use of *RAMBUTANS* tool impact the size of tests, contrasted with alternate approaches, required during testing? (regarding to efficiency)

The following null hypotheses can be determined in accordance with the aforementioned research questions:

$H_{0\mathrm{RQ1}}$ There is no significant difference in terms of effectiveness of *RAMBUTANS* and the alternate four AOP testing approaches.

$H_{0\mathrm{RQ2}}$ There is no significant difference in terms of efficiency of *RAMBUTANS* and the alternate four AOP testing approaches.

4.1 Case studies (object programs)

We used three industrial aspect-oriented software development (AOSD) projects [38] written in AspectJ to increase the likelihood of observing statistically significant defects. The projects were taken from considerably diverse application domains and large sizes. Table 2 demonstrates some broad properties of these selected target systems.

The programs were gathered from various sources, iBATIS and AJHotDraw are accessible at both SourceForge.net and Apache.org repositories, while HW has been created at the University of Lancaster in the context of AOSD European Network of Excellence, and hence, it was taken from the respective website.

4.2 Implementation of the peer approaches

Unfortunately, the current approaches (as introduced in the Background section) did not initially provide tool support overall (or possibly not accessible to open), and along these lines, we needed to re-implement their core test generation and selection strategies' algorithms to handle automating the execution of the experiment. The implementations used in the experiment to apply the peer approaches were propelled in view of the data given in the original work of the approaches themselves and, when conceivable, the original material was reused and expanded. Keeping in mind the end goal to stay away from inadvertent experimental bias, the re-implementations of the peer approaches were ideally set up not to influence the accuracy of the original work's test generation and selection strategies by taking after the directions given in the implementation section of every work.

As a decent point, each of the approaches incorporated a section of *implementation* in its respective published work that guided us to imitate the implementations done by the creators and at the same time helped minimize inclination. The basic point between all the four existing approaches (with respect to implementation section) was that they had utilized different existing tools and/or bundles to automate the functionalities of their respective components and luckily, the necessary information in such manner was all given in implementation section. For example, to apply Aspectra, we utilized the same library that the authors had used to automate the wrapper-synthesis mechanism [a bundle in the Apache Avalon Framework and based on Byte Code Engineering Library (BCEL)] or for EAT approach, we also used EvoUnit [41] from Daimler Chrysler to actualize the evolutionary testing technique as with the original work.

Having realized approaches' best configuration based on original studies' estimation and accomplishment of results, the required parameters by every technique were tuned to original/default values. For example, the stopping criterion shared by all the approaches was the number of test cases or mutation and crossover ratios for EAT were set to 0.01 and 1 individually.

4.3 Experimental process

The technical setting for all trials was identical, and the approaches were run on a machine with infrastructure consisted of: Sun JDK 1.7.0-21 with a Core i5-2520M CPU @ 2.50GHZ and 2.00 GB of RAM, under Windows 7 Professional.

Our primary methodology to conduct evaluation was based on mutation analysis [42,43] in a comparative experimental manner. Mutation analysis offers an explicit fault injection process that generates artificial faults (i.e., mutants) by precisely defining a set of mutation operators. In recent years, mutation analysis [44] has been widely used by fellow researchers evaluating various testing techniques, thus making it as an acceptable and reliable methodology.

To generate aspect-level mutants, we adopted the set of aspect-oriented mutation operators reported in [45], as depicted in Table 3. The fault injection process of the programs was automatically conducted using a modified version of a mutation system called Adaptive AjMutator [46]. Thus, mutants were created and executed automatically (i.e., AjMutator was used to seed faults (mutants) into the programs and evaluate how many mutants each test suit kills).

The number of mutants for each project is shown in Table 4. With a specific end goal to guarantee a reasonable and fair-minded comparison, the mutants for the considered object

Table 3 Mutation operators used in the experimental studies

Operators	Description
Pointcut level	
PWIW	Inserts wildcards into pointcut expressions
PWAR	Removes annotation tags from type, field, method and constructor patterns
PSWR	Removes wildcards from pointcut expressions
POPL	Changes the parameter lists of primitive Pointcut Designators/ Descriptors (PCDs)
POEC	Adds, omits or alters exception throwing clauses
PCTT	Replaces a `this` PCD with a `target` one and vice versa
PCCE	Replaces a `call` PCD with an `execution/initialization/preinitialization` PCD and vice versa
PCGS	Replaces a `get` PCD with a set one and vice versa
PCLO	Changes the logical operators in PCDs compositions
PCCC	Replaces a `cflow` PCD with a `cflowbelow` one and vice versa
Advice level	
ABAR	Replaces a `before` clause with an `after(returning|throwing)` one and vice versa
APSR	Removes invocations to proceed statement
APER	Removes guard conditions which surround proceed statements
AJSC	Replaces a `thisJoinPointStaticPart` reference with a `thisEnclosingJoinPointStaticPart` one and vice versa
ABHA	Removes implemented advice
ABPR	Changes pointcut-advice binding by replacing pointcuts which are bound to advice

Table 4 Mutants

Projects	# Mutants
`Health Watcher (HW)`	457
`AJHotDraw`	443
`iBATIS (iB)`	572
Total	1472

programs were created and stored in a mutant's repository to be shared by all the tools. Finally, the created mutants were reused by all the testing approaches for evaluation purpose.

In this experiment, we initially generated 1600 mutants for all three projects. About 8 % of the resulting mutants could not be used due to mainly compilation errors or mutant's redundancy ("false positives") and, consequently, were not selected for analysis in the experiment. Hence, this reduction left us with 1472 mutants in all to use in the experiment. As shown above, the number of mutants of each project that were

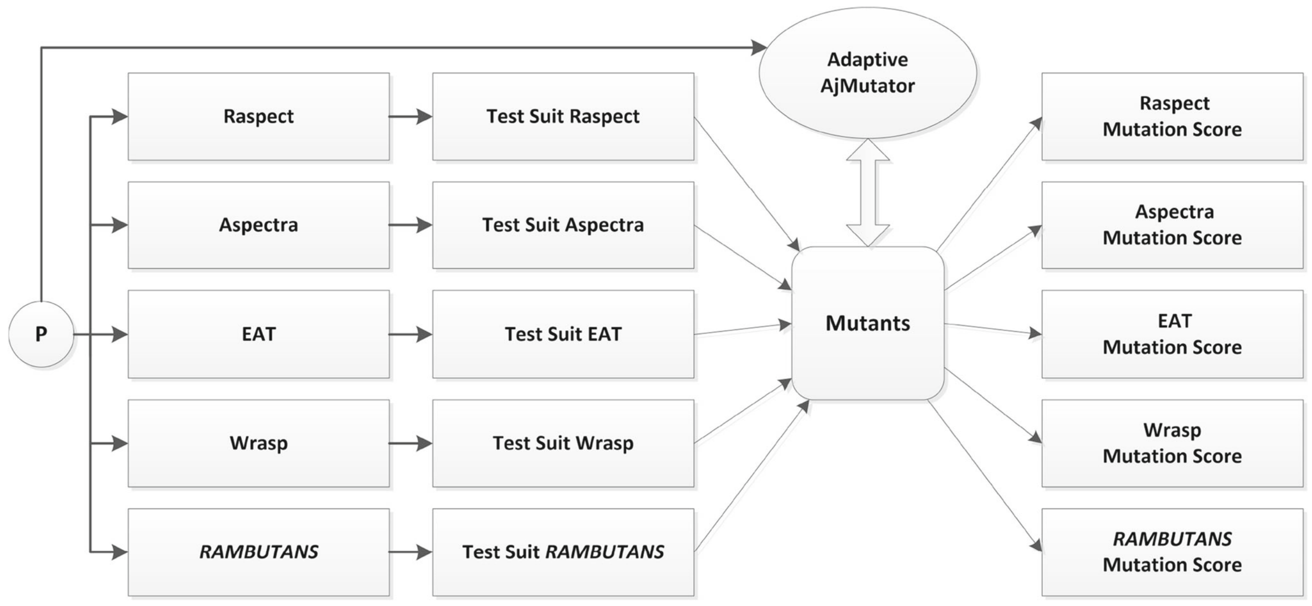


Fig. 4 Experimental process

compiled and used appears in Table 4, and the detailed distribution of the number of mutants per operator is additionally provided in Appendix B.

Test cases and data were repeatedly produced by applying the test generation approaches. That is, Wrasp, Aspectra, Raspect, EAT, and *RAMBUTANS* internally produced their own test cases. In total, 15 test suites (with every approach generating three suites) were independently generated for every considered study using the automated testing aspect-oriented approaches, which ultimately stored in a test repository.

During all iteration, the test suite exercised all the mutants of a given object program prior to resetting the test tool in which test data were discarded. The test pool is reset due to: (a) heterogeneous test data requirements of object programs, (b) inability to assess the fault detection by the already generated data, or (c) divergence in test efforts efficiency among numerous approaches.

Finally, the mutation system and adaptive AjMutator accepted the generated data to verify the results of the mutant programs on them. Subsequently, the object programs which hold the test data that are already applied to a given mutant are identified to apply the constructed test suite. Moreover, the test data and mutants were randomly selected from test pool and mutant pool accordingly. To effectively classify the mutants, mutation system was used to exercise the result checking and measurement collections. A fault is said to be detected if a fault-seeded version (i.e., mutant) behaves differently from the selected program, and the mutant is accordingly said to be killed.

Figure 4 illustrates the experimental process as a whole. The three AO programs are represented by the leftmost circle, *P*. Each of the five test generators were used to create suits of tests.

Then, AjMutator was used to generate mutants for each object program, and run all five test sets against the mutants (as presented in Table 4). This resulted in five mutation scores, i.e., the percentage of mutants detected/killed by the test suites, for each program.

4.4 Results and discussion

The summary of results is shown in Table 5. Table 5 shows the number of tests in each test suit ("T") and the mutation scores

Table 5 Experimental data

Projects	Aspectra			Wrasp			EAT			Raspect			*RAMBUTANS*		
	T	*%K*	*K/T*	*T*	*%K*	*K/T*	*T*	*%K*	*K/T*	*T*	*%K*	*K/T*	*T*	*%K*	*K/T*
`Health Watcher(HW)`	101	32.30	1.46	121	21.70	0.81	126	41.70	1.51	131	42.10	1.46	97	65.60	3.10
`AJHotDraw`	143	28.40	0.87	117	31.40	1.18	201	58.40	1.28	119	31.40	1.16	106	77.20	3.23
`iBATIS (iB)`	198	29.80	0.86	234	35.00	0.85	241	55.30	1.32	186	47.80	1.45	172	76.40	2.03
Total	442	30.16	1.06	472	29.36	0.94	568	51.80	1.37	436	40.43	1.35	375	73.06	2.79

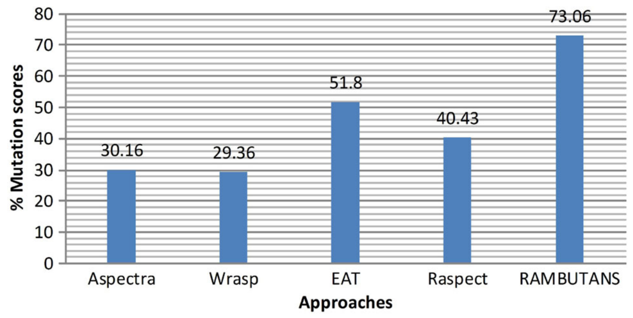


Fig. 5 Total percent mutants killed by each test suit

on the mutants, in terms of the percentage of mutants killed ("%K") in relation to total mutants, as well as efficiency in terms of the number of killed mutants per test ("K/T"). The total row gives the sum of the tests for the object programs, the average mutation score, and the average efficiency across all programs.

Given the results, the following sub-sections provide analysis (including both descriptive and statistical tests) in answering our original research questions.

4.4.1 RQ1: faults detected

Figure 5 illustrates the total percent of mutants killed by each test suit in a bar chart. The difference between the *RAMBUTANS* and the others is remarkable.

As can be seen from the figure, Aspectra achieved an average mutation score of 30.16 %. The Raspect tests (40.43 %) did better than the Aspectra and Wrasp (29.36 %) tests, but not as well as the EAT (51.8 %) tests. The EAT tests, however, kill 21 % less mutants than the strongest tool, *RAMBUTANS*, with more tests. The Wrasp tests are the weakest, Aspectra and Wrasp tests are fairly close, and, however, Aspectra was slightly better. The *RAMBUTANS* tests are still far stronger, killing 42 % more mutants than the weakest performing tool (Wrasp).

There could be possible reasons for *RAMBUTANS* outperformance, such as variation in features of considered object programs, fault injection process, and capability of AOP testing approaches (i.e., basic test selection and generation techniques for fault detection). Note that considered case studies and related mutants after applying fault injection process were taken as constant during the empirical evaluation process. Furthermore, the variability divided into treatments and blocks by the selected design. It shows that treatment effects were examined without any interference of blocks covering the output of the conducted experiment. We expect that the related diversity-based test selection mechanism conceived in the underlying technique (performed by three probabilities displayed in Table 1) to parameterize the randomization process in *RAMBUTANS* can have a significant effect on the tests' efficiency and eventually permits to enhance the state of the art. Because the thoroughness of random testing has been said to be exceptionally subject to when and how randomization is connected in the process [47]. Notwithstanding the randomized nature, *RAMBUTANS* have one of a kind attributes and contemplations amid during random generation, including having AOP-specific focus which was shown to improve the effectiveness of tests, considering interesting or special values, and reusing the return types that can affect the outcomes.

Hypothesis testing IBM SPSS version 22.0 was used to conduct statistical tests to examine the acceptance or rejection of null hypothesis, $H_{0\text{RQ1}}$. In the observation, already defined hypothesis, variable nature, residual analysis, and experimental data distribution could be verified using probability plots, such as q–q and p–p. In addition, the parametric ANOVA test which is reliable and robust was used. The significance level of 1 % was exploited in hypothesis testing at 99 % confidence level. The null hypothesis is rejected by 1 % probability of type-I-error. Moreover, statistical tests assumed equal variances of the independent treatments. This assumption was bolstered by experimental data before conducting the statistical tests. As indicated by the experimental data (i.e., particularly mutation scores) exhibited in Table 5, the randomized block ANOVA was figured as appeared in Table 6.

As it can be seen from Table 6, we got higher F value (F_{obs}) of 20.4910 than the critical value ($F_{\text{crit}} = 3.828$) for 4 and 40 F- distribution degrees of freedom and 99 % confidence level. According to the choice principle, reject $H_{0\text{RQ1}}$ if F value > F_{crit} (20.4910 > 3.828) or, comparably, if the p value = Sig. $< \alpha$ (0.0001 < 0.01). As for the above data, since p value ∼

Table 6 Results of ANOVA test on effectiveness

Source of variation	Sum of squares	Degree of freedom	Mean squares	*F* value	Sig. (*p* value)
Between treatments	3957.767	4	989.442	20.491	0.0001
Within treatments	482.867	40	48.287		
Total	4440.634	44			

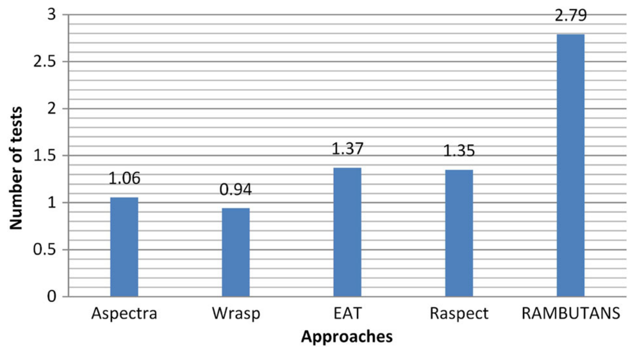


Fig. 6 Total number of mutants killed per test

0.0 < 0.01, thus we should reject the null hypothesis, H_{0RQ1}. At this stage, it can be found out that there are statistically significant differences between the approaches.

4.4.2 RQ2: number of tests required

Because the number of tests diverged widely, we asked the question "how efficient is each test generation approach?" To approximate efficiency, we computed the number of mutants killed per test. Figure 6 shows the data for all programs for each approach.

Not surprisingly, the *RAMBUTANS* tests were at the high end with a number of 2.79. The EAT generated the highest number of tests, but came out as being the second most efficient approach, 1.37. The Raspect (1.35) was almost found as efficient as EAT but required fewer tests. The Wrasp (0.94) and the Aspectra (1.06) tests were the least efficient; they generated a lot of tests without much obvious benefit, which adds a burden on the developers who must evaluate the results of each test.

An interesting observation is the high numbers of tests required by EAT compared to others. In this regard, EAT can be the least effective in terms of required test effort contrasted with other approaches. For example, the runtime overhead of applying EAT was entirely high and that can matter if the approach is utilized for real-time systems, where every second counts. This might be due to the search-based testing, especially genetic algorithm has been observed an expensive technique which offers high coverage value that has higher efficiency. This higher efficiency with the greater amount of essential efforts is mainly because of the nature of search-based techniques that involve significant tests due to automated test generation. The EAT had also the same case in which experimental results disclosed the issue. In actuality, *RAMBUTANS* was appeared to be very quick in test generation. The reason is that *RAMBUTANS* benefits from randomized testing to which the capacity to create numerous tests in a brief timeframe has been stamped as its favorable advantage [47].

Hypothesis testing The parametric ANOVA test was applied to test formulated hypothesis (H_{0RQ2}) on results to express the differences in efficiency between considered test generation approaches. The significance level in relation with the preceding test was set to $\alpha = 0.01$. According to the experimental data, presented in Table 5, the randomized block ANOVA was calculated, as shown in Table 7.

From Table 7, it can be examined that suggested results reject H_{0RQ2} at 0.01 level of significance (p value = Sig. < α, i.e. $0.0006 < 0.01$). Consequently, it was decided to reject the null hypothesis.

At last, the results of the experiment present proof of the effectiveness and efficiency of the proposed tool, which had an AOP-specific focus in its test generation. The outcomes additionally concern the degree of improvement over the current state of the art, i.e., prior automated AOP testing tools. From a different point of view, such empirical results in the AOP testing area can frame an assortment of knowledge over time, giving testers/developers the flexibility to choose the best tool or mix of tools concerning their own thinking and suitable conformity between efficiency and effectiveness.

4.4.3 Execution time

The execution time of the different tools taken to produce the test suites is a feature involving test effort that matters in industrial cases. In this view, we have reported the execution time of all AOP testing tools in the production of test suites. We determined the execution time, measured in minutes as elapsed time, of each tool required for the generation of corresponding number of test cases presented in Table 5. The amount of execution time taken to produce the test suits recorded from the experiment is presented in Table 8.

Table 7 Results of ANOVA on efficiency

Source of variation	Sum of squares	Degree of freedom	Mean squares	F value	Sig. (p value)
Between treatments	6.569	4	1.642	12.8900	0.0006
Within treatments	1.274	40	0.127		
Total	7.843	44			

Table 8 Execution time

Tools	Execution time (min)
Aspectra	26.05
EAT	37.85
RAMBUTANS	14.72
Raspect	29.11
Wrasp	27.39

According to the table, *RAMBUTANS* required about 14.72 min to generate the total number of 375 test cases, whereas EAT, Raspect, Aspectra, and Wrasp required 37.85, 29.11, 26.05, and 27.39 min, respectively, to produce 568, 436, 442, and 472 test cases. It can be observed that there is a functional relationship between the number of test cases and execution time in which the number of test cases run determines the time. Seen in this way, it can be said that the more the test cases, the longer the time to produce the test cases. There is, however, an exception to this observation, and Raspect was shown to require less number of test cases than Aspectra and Wrasp, but it took a higher execution time for its test productions. Overall, *RAMBUTANS* could be seen as much faster tool than others could as it produced a set of more-effective test cases in the shortest time.

4.5 *RAMBUTANS* vs. classical Java random testing tool

The objective of this section is to provide a comparison between the proposed tool, *RAMBUTANS*, and a classical Java random testing tool that does not take the aspects into account. Since AspectJ programs are eventually compiled into Java bytecode, it would be very interesting to realize whether focusing on the aspect features would be a good technique (as devised in our tool) over other tools that are meant for testing object-oriented (OO) programs. In other words, we will investigate what it will be the essential differences (in terms of effectiveness and efficiency) between a general-purpose Java tool to generate tests for aspect-oriented (AO) programs in comparison with our tool that has a AOP-specific focus.

There are few numbers of existing well-designed Java test generation tools (e.g., RANDOOP [48] and Jartege [49]) in the literature. In this particular experiment, we used RANDOOP, which uses a guided random testing technique to compare its performance with *RAMBUTANS*. To perform this experiment and arrive to results, we have followed the experimental process as listed below:

1. *Preparing programs* As first step, we needed to compile all the three AspectJ programs to obtain their woven bytecode. To this end, we used AspectJ *ajc* compiler. It is important to note that, we had another choice to make use of RANDOOP with AspectJ programs, as it is not primarily designed for aspects. This could include rewriting all the aspects in the programs using the annotation style, which is a feature of AspectJ 5, also known as @AspectJ annotation. Using this feature, we can write aspects and aspect members with regular Java syntax, so that they can be interpreted by the AspectJ weaver. In the annotation style, aspects are declared as classes and advice is a method. This feature allows RANDOOP to recognize these constructs and generate calls for them. However, due to huge manual effort to rewrite all the programs and considering the fact that it is very likely to be an error-prone process, we decided to use the bytecode of programs as feed to the tool.
2. *Test generation* Once the programs are ready to feed, we used RANDOOP tool to generate tests. The woven bytecode of each AspectJ program was given to the tool. For each program, we set RANDOOP to generate the same number of test cases as with *RAMBUTANS*, resulting in three test suites for all the programs (97, 106, and 172 tests for HW, AJHotDraw, and iB programs, respectively; 375 in all).
3. *Applying tests and measurement* As last step, tests generated by RANDOOP were run on mutants of each program. Hence, we measured the mutation scores on the mutants, in terms of the percentage of mutants killed (“%K”) in relation with total mutants, as well as efficiency in terms of the number of killed mutants per test (“K/T”). Note that we have used the exact same mutants that were generated and used on *RAMBUTANS* tests.

The results of this experiment on the same basis (in terms of mutants and number of tests) with regard to the abovementioned metrics are shown in Table 9. The “Total” row gives

Table 9 Comparison of RANDOOP and *RAMBUTANS*

Programs	*%K*		*K/T*	
	RANDOOP	*RAMBUTANS**	RANDOOP	*RAMBUTANS**
`HW`	31.05	65.6	1.46	3.1
`AJHotDraw`	27.79	77.2	1.16	3.23
`iBATIS (iB)`	33.85	76.4	1.12	2.03
Total	30.89	73.06	1.24	2.79

* The results of *RAMBUTANS* were reused from the first experiment (see Table 5)

the average mutation score and the average efficiency across all programs.

The results show that there is a considerable difference in the mutation score (%*K*) obtained by both the tools. It can be seen that utilizing an AOP-specific technique, *RAMBUTANS*, over OO-specific technique, RANDOOP, the effectiveness of tests can be increased by over 42 %. An interesting point which can also be noticed is that the total %*K* obtained by RANDOOP was slightly better than Aspectra and Wrasp tools in the first study (see Table 5). Efficiencywise (K/T), there was a larger relative change of 44 % between RANDOOP and *RAMBUTANS*, which is approximately twice as high as that of its competitor. Once again and interestingly, the total figure of K/T obtained by RANDOOP was seen to be better than Aspectra and Wrasp tools in the first experiment.

Overall, the results from this particular experiment indicate that classical Java testing tools may not be perfectly suitable for testing AO programs, though applicable. The underlying reason could be the ignorance of the aspectual features by such tools, which switch the generation entrance from aspects to objects. These results could provide evidence that the essential difference imposed by AOP-specific focus can make our tool win against general-purpose Java random testing competitors. On a specific note, results from such comparison would further help convince the need of designing the proposed automated test generation tool for aspectual features in this research.

4.6 Threats to validity

This section explains the validity threats considered for this study and focuses concentration on common threats that affect results.

The external validity threats for the most part concentrate on legitimizing how illustrative the considered case studies (i.e., object programs), AOP testing approaches, and supporting tools are. To handle the external validity threats, we used three popular AspectJ benchmarks, real-world programs with different characteristics along with frequently used aspects. Having three large objects is such a reasonable number that can have a representative of the developers in the industry. From the investigation of the other related work, it has been watched that only two or three small programs are regularly used as objects in the experiments.

Regarding the object program size used in the experiment, it merits underscoring that we have utilized three industrial-sized AOP systems to incorporate into the experiment. Despite the fact that including and breaking down more large-scale systems would be essential for a superior comprehension of the relative performance, we trust the present results exhibited in the paper and started from the three programs can even now give bits of knowledge into the way we evaluate the existing AOP testing approaches. Thus, the object representativeness might not be a worrying threat, as the programs are moderately sensible to make the experiment comes about more summed up.

The selected approaches are only available reported approaches in the context of AOP testing, which were used in the experiment to evaluate *RAMBUTANS*. In fact, this threat could be minimized by conducting extensive experiments on AOP testing approaches that can be presented in future research. The occurrence of faults in Adaptive AjMutator as the auxiliary tool is another potential external validity threat to experiment, since it was originally modified from AjMutator tool to tackle its limitations. Therefore, we tested and examined before applying the tool to experiment to handle this tool related threat. To this end, we shaped a review group of three Ph.D. students (majoring in Software Engineering) of the University of Malaya. Students individually analyzed the code (typically spend 4–6 h) to review all the production classes in the tool, testing them with sample inputs and programs. Once students tested the tool, they performed an open discussion with researchers to solve conflicts and reach a consensus on a couple of defects found.

The implementation effects and measurements are the possible threats to internal validity that can bias the results. Implementation of the peer approaches could be of a concern to which, how we are sure that the implementation if faithful. To mitigate this thread, we replicated experiments/examples from the original authors to check that the same results were obtained. It turned that the results were pretty much identical with no major differences. As such, this could not be a bias in the experiments. In addition, artificial fault's injection instead of real faults might change the outcome in the above mentioned effects. The seeded faults using mutation operators have no significant effect on software development in the aspect-oriented context. The unavailability of fault's documentation and benchmark mutants that occur during development is the main problems in mitigating this threat. Conversely, the occurrence of this threat can be minimized as concluded in a previous study [45]. In that study, promising results were achieved using mutation operators. The selection of mutation operators was grounded in a relatively recent research, which suggests a number of important aspect-oriented mutation operators. Consequently, the mutation operators represent real faults.

The conclusion validity threats are mainly concerned with the statistical analysis. Appropriate statistical tests, such as ANOVA, were performed to assess the null hypotheses. In addition, statistical tests' assumptions were met, and thus no obvious error rate was observed. Finally, this conclusion validity related threat was marginal.

5 Implications and limitations

This tool paper offers some implications for practitioners and researchers. The analysis of the specific results extracted from the conducted experiments would facilitate decision makers with descriptive and statistical information, which could be useful to assist them to choose the most suitable test generation tool for a given project. Program testers might consider *RAMBUTANS* as an appropriate test generation approach in AO projects in which the prioritization of aspectual features is needed, the number of aspect is large, and the generation process needs to be fast and simple while preserving effectiveness. Manually generating a large number of test cases is nearly impractical, but automated test generation tools render such a task possible. The results reported in this paper suggest that the proposed tool could be of significant benefit to software testers.

The findings of this study could also be used as a guideline by interested researchers for identifying trends before initiating a new approach in the future or evaluating existing ones. It is challenging to claim that the results of this study could be generalized to business settings due to the usage of students and researchers as subjects. To have rigorous evidence of confirming or contradicting this claim, it would be of interest for researchers to investigate the replication of the empirical part by participating professionals as subjects. It would also be worthwhile for researchers to conduct further empirical studies on a larger number of AO projects to figure out how similar would be the results with the findings of this study.

The limitations of the *RAMBUTANS* tool include the following:

Support of AspectJ features AspectJ is a very feature-rich language. The presented tool aims at effective aspect-level testing of crosscutting concerns (which can be seen as a sort of unit/ module testing), specifically those features related to aspectual implementation of concerns. However, it only considers the most generally used join points related to constructors and methods (call, execution), as they are the building blocks of the object-oriented software systems and speak to the most valuable points at which the crosscutting behaviors can be woven.

An aspect mainly interacts with base code/program via weaving rules which specifies what actions to carry out when particular points in the execution of the program are reached (it is important to note that the weaving takes place only in the bytecode produced by the AspectJ compiler). These weaving rules for sure are those crosscutting requirements/behaviors executed by means of advice in AspectJ (what is known as dynamic crosscutting). All together for this interaction to happen appropriately, the pointcut associated with advice, inter-type declarations (what is known as static crosscutting), and built-in methods of aspects should work, as they are composed. In the presented tool, it is assumed that these constructs work, as they designed. The reason lies in fact that the fundamentally usage of these constructs and especially static crosscutting is to give support for the enforcement of dynamic crosscutting but not changing the runtime conduct of the base program. For example, the infusion of new fields or methods through inter-type declaration does not change the state of the base code. In our perspective, this limitation could not restrict generality of the proposed tool, as the focus is still to provide effective dynamic tests for aspects.

Test coverage Constrained execution paths may be difficult to exercise without resorting to constraint satisfaction techniques. In other words, how much testing would be enough? This is a concern that has been raised and discussed in many testing settings. It is still an area of active debate in the software testing community. Apparently, the answer to this question is completely intuitive and depends on the decision made by the tester about how much testing is enough. This means, in this context, the burden would be on tester to determine which aspects should be tested and to what extent. However, *RAMBUTANS* uses flexible stopping criteria (either a single criterion, such as time limit, aspect coverage, and number of test cases or a combined one) that can give the tester the flexibility of adjusting testing period according to their own reasoning or intuitions about the matter.

Parsing capabilities Parsing source code based on regular expressions might be adequate only for the so-called regular languages, which could be the case of Java or AspectJ. However, it could be better to resort to existent ‘official’ parsers for Java and AspectJ to make the tool more robust.

6 Conclusion and future work

This paper presented a new AOP-specific tool that automates the generation of tests, which is the most technically challenging part of testing, for aspects in AspectJ. No tool currently exists that provides automatic test generation for aspects. Our tool is specifically targeted towards automatically testing aspects by generating test data to exercise the advice and pointcuts, and thereby serves as a means of verifying the correctness and quality of aspects. The tool is capable of producing a large number of tests that in turn would be handy for regression testing of AspectJ programs and, consequently, integrating testing with manual testing, i.e., supplying hand-written test data to the automatically generated test classes.

Since random testing techniques can lead to statistical analysis, hence they can be used in reliability assessment and estimation of a program from test results (in this context reliability can be a measure of the proportion drawn from a given input distribution that the program treats correctly

[50]). Thus, the AOP testing tool proposed in this study can also be seen as an AOP risk and/or reliability assessment means to address the risk associated with the adoption of these programs.

In the future, we would like to work on the scalability of the tool and also extend it into a multi-language tool (other than AspectJ) that can recognize other programming languages (such as C [51] or C++), whose syntax is comparable to that of AspectJ.

Acknowledgements R. Meimandi Parizi would like to thank the New York Institute of Technology (NYIT) for providing financial support. A.A. A. Ghani's work was supported in part by Fundamental Research Grant Scheme with reference FRGS/2/2010 /SG/UPM/01/2. S. P. Lee's work was carried out within the research project supported by High Impact Research Grant with reference UM.C/625/1/HIR/MOHE/FCSIT/13, funded by the Ministry of Education, Malaysia. S.U.R. Khan would like to acknowledge the financial support of the Bright Sparks Program at University of Malaya, Malaysia under reference BSP-151(3)-11.

7 Appendix A

Internal algorithms of components used in *RAMBUTANS* can be found at: http://goo.gl/DG0OGv.

8 Appendix B

The number of the final mutants was made to the programs during the experiment. The following table shows the distribution of mutants per mutation operator in relation with the three programs (Table 10).

Table 10 Details of mutants

Operators	Health Watcher	AJHotDraw	iBATIS	Total
PWIW	22	19	28	69
PWAR	21	23	33	77
PSWR	19	16	37	72
POPL	38	32	27	97
POEC	29	25	26	80
PCTT	28	28	42	98
PCCE	32	24	38	94
PCGS	23	19	29	71
PCLO	33	29	37	99
PCCC	20	23	33	76
ABAR	31	29	42	102
APSR	26	33	35	94
APER	32	38	40	110
AJSC	26	30	36	92
ABHA	40	36	46	122
ABPR	37	39	43	119
Total	457	443	572	1472

References

1. Kiczales, G.: Aspect-oriented programming. ACM Comput. Surv. **28** (1996)
2. Kiczales, G., Lamping, J., Lopes, C.V., Hugunin, J.J., Hilsdale, E.A., Boyapati, C.: Aspect-oriented programming, United States Patent 6467086, Xerox Corporation Patent (2002)
3. Hoffman, K., Eugster, P.: Cooperative aspect-oriented programming. Sci. Comput. Program. **74**, 333–354 (2009)
4. Nakagawa, E.Y., Ferrari, F.C., Sasaki, M.M.F., Maldonado, J.C.: An aspect-oriented reference architecture for Software Engineering Environments. J. Syst. Softw. **84**, 1670–1684 (2011)
5. Kallel, S., Charfi, A., Mezini, M., Jmaiel, M.: Combining formal methods and aspects for specifying and enforcing architectural invariants. In: Proceedings of the 9th International Conference on Coordination Models and Languages, pp. 211–230 (2007)
6. Garcia, A., Batista, T., Rashid, A., Sant'Anna, C.: Driving and managing architectural decisions with aspects. ACM SIGSOFT Softw. Eng. Notes **31**, 1–8 (2006)
7. Mets, J., Maoz, S., Katara, M., Mikkonen, T.: Using aspects for testing of embedded software: experiences from two industrial case studies. Softw. Qual. J. **22**, 185–213 (2013)
8. Linehan, E., Clarke, S.: An aspect-oriented, model-driven approach to functional hardware verification. J. Syst. Archit. **58**, 195–208 (2012)
9. Driver, C., Reilly, S., Linehan, E., Cahill, V., Clarke, S.: Managing embedded systems complexity with aspect-oriented model-driven engineering. ACM Trans. Embed. Comput. Syst. **10**, 1–26 (2010)
10. Cleland-Huang, J., Settimi, R., Zou, X., Solc, P.: The detection and classification of non-functional requirements with application to early aspects. In: Proceedings of the 14th IEEE International Conference Requirements Engineering, pp. 39–48 (2006)
11. Amar, B., Leblanc, H., Coulette, B., Nebut, C.: Using aspect-oriented programming to trace imperative transformations. In: Proceedings of the 14th IEEE International Enterprise Distributed Object Computing Conference, pp. 143–152 (2010)
12. Voelter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software development. In: Proceedings of the 11th International Software Product Line Conference, pp. 233–242 (2007)
13. Lee, J.-S., Bae, D.-H.: An aspect-oriented framework for developing component-based software with the collaboration-based architectural style. Inf. Softw. Technol. **46**, 81–97 (2004)
14. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: Proceedings of Aspect-Oriented Software Development, pp. 21–35 (2005)
15. Lemos, O.A.L., Franchin, I.G., Masiero, P.C.: Integration testing of Object-Oriented and Aspect-Oriented programs: A structural pairwise approach for Java. Sci. Comput. Program. **74**, 861–878 (2009)
16. Meimandi Parizi, R., Ghani, A.A.A., Abdullah, R., Atan, R.: Empirical evaluation of the fault detection effectiveness and test effort efficiency of the automated AOP testing approaches. Inf. Softw. Technol. **53**, 1062–1083 (2011)
17. Xie, T., Zhao, J., Marinov, D., Notkin, D.: Automated test generation for AspectJ programs. In: Proceedings of the 1^{st} Workshop on Testing Aspect-oriented Programs, pp. 1–6 (2005)
18. Xie, T., Zhao, J.: A framework and tool support for generating test inputs of AspectJ programs. In: Proceedings of the 5th International Conference on Aspect-Oriented Software Development, pp. 190–201 (2006)
19. Xie, T., Zhao, J., Marinov, D., Notkin, D.: Detecting redundant unit tests for AspectJ programs. In: Proceedings of the 17th International Symposium on Software Reliability Engineering, pp. 179–190 (2006)

20. Harman, M., Islam, F., Xie, T., Wrappler, S.: Automated test data generation for aspect-oriented programs. In: Proceedings of the 8th International Conference on Aspect-Oriented Software Development, pp. 185–196. Charlottesville (2009)
21. Duran, J.W., Ntafos, S.C.: An evaluation of random testing. IEEE Trans. Softw. Eng. **10**, 438–444 (1984)
22. Arcuri, A., Iqbal, M.Z., Briand, L.: Random testing: theoretical results and practical implications. IEEE Trans. Softw. Eng. **38**, 258–277 (2012)
23. Godefroid, P., Klarlund, N., Sen, K.: DART: directed automated random testing. In: Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation, pp. 213–223. Chicago (2005)
24. Ciupa, I., Leitner, A., Oriol, M., Meyer, B.: ARTOO: Adaptive random testing for object-oriented software. In: Proceedings of the 30th International Conference on Software Engineering, pp. 71–80. Leipzig (2008)
25. Bertolino, A.: Software testing research: achievements, challenges, dreams. In: Proceedings of the 2007 Future of Software Engineering Conference, pp. 85–103 (2007)
26. Tsoukalas, M.Z., Duran, J.W., Ntafos, S.C.: On some reliability estimation problems in random and partition testing. IEEE Trans. Softw. Eng. **19**, 687–697 (1993)
27. Meimandi Parizi, R., Ghani, A.A.A., Lee, S.P.: Automated test generation technique for aspectual features in AspectJ. Inf. Softw. Technol. **57**, 463–493 (2015)
28. Ferrari, F.C., Burrows, R., Lemos, O.A.L., Garcia, A., Maldonado, J.C.: Characterising faults in aspect-oriented programs: towards filling the gap between theory and practice. In: Proceeding of the 2010 Brazilian Symposium on Software Engineering, pp. 50–59 (2010)
29. Kiczales, G., Hilsdale, E.A., Hugunin, J.J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Proceedings of the 15th European Conference on Object-Oriented Programming, pp. 327–353 (2001)
30. Lopes, C.V., Ngo, T.C.: Unit testing aspectual behavior. In: Proceedings of the 1st Workshop on Testing Aspect Oriented Programs, pp. 1–6 (2005)
31. Yamazaki, Y., Sakurai, K., Matsuura, S., Masuhara, H., Hashiura, H., Komiya, S.: A unit testing framework for aspects without weaving. In: Proceedings of the 1st Workshop on Testing Aspect-oriented Programs, pp. 1–5 (2005)
32. Csallner, C., Smaragdakis, Y.: JCrasher: an automatic robustness tester for Java. Softw Pract. Exp **34**, 1025–1050 (2004)
33. Parasoft Jtest. http://www.parasoft.com/jsp/products/jtest.jsp. Accessed 7 June 2015
34. Xie, T., Marinov, D., Notkin, D.: Rostra: A framework for detecting redundant object-oriented unit tests. In: Proceedings of the 19th IEEE International Conference on Automated Software Engineering, pp. 196–205 (2004)
35. Alexander, R.T., Bieman, J.M., Andrews, A.A.: Towards the systematic testing of aspect-oriented programs. Colorado State University, Technical Report CS-4-1052003
36. Meimandi Parizi, R., Ghani, A.A.A.: A theoretical evaluation of automated aspect-oriented program testing approaches. In: Proceedings of the Annual International Conference on Software Engineering, pp. 11–19. Phuket (2010)
37. Nakagawa, E.Y., Simo, A.d.S., Ferrari, F.C., Maldonado, J.C.: Towards a reference architecture for software testing tools. In: Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering (2007)
38. Rashid, A., Cottenier, T., Greenwood, P., Chitchyan, R., Meunier, R., Coelho, R., et al.: Aspect-oriented software development in practice: Tales from AOSD-Europe. Computer **43**, 19–26 (2010)
39. Soares, S., Borba, P., Laureano, E.: Distribution and persistence as aspects. Softw. Pract. Exp. **36**, 711–759 (2006)
40. Ferrari, F.C., Burrows, R., Lemos, O., Garcia, A., Figueiredo, E., Cacho, N., et al.: An exploratory study of fault-proneness in evolving aspect-oriented programs. In: Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering, pp. 65–74 (2010)
41. Wappler, S.: Automatic generation of object-oriented unit tests using genetic programming, Ph.D. thesis, Technical University of Berlin (2008)
42. Jia, Y., Harman, M.: An analysis and survey of the development of mutation testing. IEEE Trans. Softw. Eng. **37**, 649–678 (2011)
43. Offutt, J.: A mutation carol: Past, present and future. Inf. Softw. Technol. **53**, 1098–1107 (2011)
44. Ferrari, F.C., Rashid, A., Maldonado, J.C.: Towards the practical mutation testing of AspectJ programs. Sci. Comput. Program. **78**, 1639–1662 (2013)
45. Ferrari, F.C., Maldonado, J.C., Rashid, A.: Mutation testing for aspect-oriented programs. In: Proceedings of the 1st International Conference on Software Testing, Verification, and Validation, pp. 52–61 (2008)
46. Delamare, R., Baudry, B., Le Traon, Y.: AjMutator: A tool for the mutation analysis of AspectJ pointcut descriptors. In: Proceedings of the 2nd International Conference on Software Testing, Verification, and Validation Workshops, pp. 200–204 (2009)
47. Andrews, J.H., Menzies, T., Li, F.C.H.: Genetic algorithms for randomized unit testing. IEEE Trans. Softw. Eng. **37**, 80–94 (2011)
48. Pacheco, C., Ernst, M.D.: Randoop: feedback-directed random testing for Java. In: Proceedings of the Companion to the 22nd ACM SIGPLAN Conference on Object-oriented Programming Systems and Applications, pp. 815–816. Montreal (2007)
49. Oriat, C.: Jartege: A tool for random generation of unit tests for Java classes. In: Proceedings of the 1st International Conference on the Quality of Software Architectures, pp. 242–256 (2005)
50. Tsoukalas, M.Z., Duran, J.W., Ntafos, S.C.: On some reliability estimation problems in random and partition testing. IEEE Trans. Softw. Eng. **19**, 687–697 (1993)
51. Novikov, E.M.: An approach to implementation of aspect-oriented programming for C. Program. Comput. Softw. **39**, 194–206 (2013)