Ferrari *et al. Journal of the Brazilian Computer Society* (2015) 21:20
DOI 10.1186/s13173-015-0040-1

Journal of the Brazilian Computer Society
a SpringerOpen Journal

RESEARCH Open Access

Testing of aspect-oriented programs: difficulties and lessons learned based on theoretical and practical experience

Fabiano C. Ferrari[1*], Bruno B. P. Cafeo[2], Thiago G. Levin[1], Jésus T. S. Lacerda[1], Otávio A. L. Lemos[3], José C. Maldonado[4] and Paulo C. Masiero[4]

Abstract

Background: Since the first discussions of new challenges posed by aspect-oriented programming (AOP) to software testing, the real difficulties of testing aspect-oriented (AO) programs have not been properly analysed. Firstly, despite the customisation of traditional testing techniques to the AOP context, the literature lacks discussions on how hard it is to apply them to (even ordinary) AO programs based on practical experience. Secondly, and equally important, due to the cautious AOP adoption focused on concern refactoring, test reuse is another relevant issue that has been overlooked so far. This paper deals with these two issues. It discusses the difficulties of testing AO programs from three perspectives: (i) structural-based testing, (ii) fault-based testing and (iii) test set reuse across paradigms.

Methods: Perspectives (i) and (ii) are addressed by means of a retrospective of research done by the authors' group. We analyse the impact of using AOP mechanisms on the testability of programs in terms of the underlying test models, the derived test requirements and the coverage of such requirements. The discussion is based on our experience on developing and applying testing approaches and tools to AspectJ programs at both unit and integration levels. Perspective (iii), on the other hand, consists of recent exploratory studies that analyse the effort to adapt test sets for refactored systems and the quality of such test sets in terms of structural coverage.

Results: Building test models for AO programs imposes higher complexity when compared to the OO paradigm. Besides this, adapting test suites for OO programs to AO equivalent programs tends to require less effort than doing the other way around, and resulting suites achieve similar quality levels for small-sized aplications.

Conclusions: The conclusion is that building test models for AO programs, as well as deriving and covering paradigm-specific test requirements, is not straightforward as it has been for procedural and object-oriented (OO) programs at some extent. Once you have test suites in conformance with programs implemented in both paradigms, the quality of such suited in termos of code coverage may vary depending on the size and characteristics of the applications under testing.

Keywords: Software testing, Aspect-oriented programming, Object-oriented programming, Software refactoring, Test reuse

*Correspondence: fabiano@dc.ufscar.br
[1]Computing Department, Federal University of São Carlos, Rod. Washington Luis, km 235, 13565-905 São Carlos, SP, Brazil
Full list of author information is available at the end of the article

Introduction

In 2004, Alexander et al. [1] first discussed the challenges posed by aspect-oriented programming (AOP) to the software testing researchers. They enumerated potential sources of faults in aspect-oriented (AO) programs, ranging from the base code itself (i.e. not directly related to aspectual code) to emerging properties due to multiple aspect interactions. In the same report, they proposed a candidate, coarse-grained fault taxonomy for AO programs. Ever since, the software testing community has been investigating ways of dealing with the challenges described by them. In summary, research on testing of AO programs (hereafter called *AO testing*) has been mainly concerned with: (i) the characterisation of fault types and bug patterns [2–7], (ii) the definition of underlying test models and test selection criteria [8–18] and (iii) the provision of automated tool support [11, 14, 16, 18–22]. In particular, structural-based and mutation-based testing have been on focus by several research initiatives [8, 9, 11–29].

Despite the variety of approaches for testing AO software, too little has been reported about the difficulties of applying them based on practical experience. In other words, researchers rarely discuss the difficulty of fulfilling AO-specific test requirements and the ability of their approaches in revealing faults in AO programs. For example, questions like "how hard is for one to create a test case to traverse a specific path in an AO program graph (in structural-based testing)?" and "how hard is for one to kill an AO mutant (in mutation-based testing)?" can hardly be answered based on the analysis and discussions presented in the existing literature. Besides this, we observe that, even after almost two decades of the AOP dissemination, it is still adopted with caution by practitioners and researchers. This fact was observed in two relatively recent reports [30, 31]. From our experience and observations, when adopted, AOP is applied to refactor existing object-oriented (OO) systems to achieve better modularisation of behaviour that appears intertwined or spread across the system modules (these are the so-called *crosscutting concerns* [32]). Examples of AOP applied in this context can be found in the work of van Deursen et al. [2], Mortensen et al. [33], Ferrari et al. [34] and Alves et al. [35, 36], not limited to particular technologies such as Java and AspectJ.

Our previous research investigated the fault-proneness of AO programs based on faults identified during the testing of real-world AO applications [34]. This is related to the first aforementioned topic (i.e. fault characterisation). The conclusion was that, amongst the main mechanisms commonly used in AO programs, none of them stands out in terms of fault-proneness. In that exploratory study, we used test sets built upon the OO versions of the applications and then used such test sets to evaluate the AO counterparts with some test set customisations. Even though that study [34] addressed the reuse of test suites in refactoring scenarios, we did not provide any discussion with respect to the achieved code coverages, neither with respect to effort required for reusing test sets.

In this paper, we revisit our contributions on AO testing achieved by our research group along the last decade. We discuss the challenges and difficulties of testing AO programs from three perspectives: (i) structural-based testing, (ii) fault-based testing and (iii) test set reuse across programming paradigms. Regarding perspectives (i) and (ii), we analyse the impact of using AOP mechanisms on the testability of programs in terms of the definition of the underlying models, the derivation of test requirements and the coverage of the requirements. In regard to the perspective (iii), considering the OO and AO paradigms, we address the effort for adapting test suites from one paradigm to the other and analyse the quality of reused test sets in both paradigms.

We highlight upfront that this paper extends the discussions and results presented in a previous publication [37]. In order to extend our previous work, we focused on the aforementioned perspective (iii)—test set reuse across programming paradigms. We report on the results of a recently performed exploratory study that measures the effort (in terms of code changes) required to adapt test suites from one paradigm to the other and vice versa. Beyond this, we measure the structural coverage that results from the applied test sets. The reader should notice the points presented in this paper rely on our practical experience of establishing and applying approaches to test AO programs by means of theoretical definitions and exploratory assessments.

The remainder of this paper is organised as follows: section 'Background' describes basic background on structural and fault-based testing. It also presents basic concepts of aspect-oriented programming and the AspectJ language. Sections 'Structural-based viewpoint analysis' and 'Mutation-based viewpoint analysis' revisit the contributions of our research group on structural and mutation testing of AO programs, respectively. Section 'Reuse-centred viewpoint analysis' brings novel results of an exploratory study that addressed the reuse of test sets across the OO and AO paradigms. Examples and experimental results are presented along sections 'Structural-based viewpoint analysis', 'Mutation-based viewpoint analysis' and 'Reuse-centred viewpoint analysis'. Section 'Related work' summarises related research. Finally, section 'Final remarks, limitations and research directions' points out future research directions and concludes this work.

Background

Structural testing

Structural testing—also called *white-box testing*—is a technique based on internal implementation details of the software. In other words, this technique establishes testing requirements based on internal structures of an application. As a consequence, the main concern of this technique is with the coverage degree of the program logic yielded by the tests [38].

In structural testing, a *control flow graph* (CFG) is typically used to represent the control flow of a program. A CFG is a directed graph representing the order in which the individual statements, instructions or function calls of program are executed. In a CFG, nodes represent a statement or a block of statements, and edges represent the flow of control from one statement or block of statements to another. In the context of this paper, we define a block of statement as a set of statements of a program. After the execution of the first statement of the block, the other statements within the block are sequentially executed according to the control flow. Each block corresponds to a node in the CFG and the transfer of control from one node to another is represented by directed edges between nodes.

Test selection criteria (or simply *testing criteria*) based on control flow use only information about the execution flow of the program such as statements and branches to determine which structures need to be tested. Typical structural-based testing criteria defined based on a CFG are all-nodes, all-edges and all-paths [38]. These criteria require test cases that exercises all nodes (i.e. all statements), all edges (i.e. all branches), and all paths (i.e. all possible combination of nodes and edges) that compose a CFG, respectively. It is important to notice that, although desirable, the coverage of all of these criteria is unfeasible in general. For instance, the coverage of the all-paths criterion may be impracticable due to the high number of paths in a CFG. This and other limitations of the control flow-based criteria motivated the introduction of data flow-based criteria.

For data flow-based testing, the *def-use graph* extends the CFG with information about the definitions and uses of variables [39]. Data flow-based testing uses data flow analysis as source of information to derive testing requirements. In other words, the interactions involving definition of variables and use of such definitions are explored to derive test requirements. For our purposes, the occurrence of a variable in a program is classified either as a definition or a use. We consider as a *definition* a value assignment to a variable. With respect to use occurrences, we consider as a predicate use (*p-use*), a use of a variable associated with the decision outcome of the predicate portion of a decision statement—e.g. *if (x == 0)*—and as a computational use (*c-use*), a use of a variable that directly affects a computation and it is not a p-use—e.g. $y = x + 1$. P-uses are associated to the def-use graph edges and c-uses are associated to the nodes. A definition clear path (*def-clear path*) is a path that goes from the definition place of a variable to a subsequent c-use or p-use, such that the variable is not redefined along the way. A def-use pair with respect to some variable is then a pair of definition and subsequent use locations such that there is a def-clear path with respect to that same variable from the definition to the use location [39]. If a def-use graph is used as the underlying model, typical criteria are all-defs and all-uses [39]. In short, such data flow-based criteria require test cases that traverse paths that include the definition and subsequent uses of variables of the program. For more information about the structural-testing criteria mentioned in this section, the reader may refer to seminal studies of structural testing [38, 39]).

Fault-based testing and mutation testing

The fault-based testing technique derives test requirements based on information about recurring errors made by programmers during the software development process. It focuses on types of faults which designers and programmers are likely to insert into the software and on how to deal with this issue in order to demonstrate the absence of such prespecified faults [40]. In this technique, fault models (or fault taxonomies) guide the selection or design of test cases that are able to reveal fault types characterised on such models. Fault models and taxonomies can be devised from a combination of historical data, researchers' and practitioners' expertise and specific programming paradigm concepts and technologies.

The most investigated and applied fault-based test selection criterion is the mutant analysis [41], also known as mutation testing. Basically, it consists in creating several versions of the program under testing, each one containing a simple fault. Such modified versions of the program are called *mutants* and are expected to behave differently from the original program. Each mutant is executed against the test data and is expected to produce a different output when compared to the execution of the original program.

In mutation testing, given an original program P, *mutation operators* encapsulate a set of modification rules applied to P in order to create a set of mutants M. Then, for each mutant m, $(m \in M)$, the tester runs a test suite T originally designed for P. If $\exists t, (t \in T) \mid m(t) \neq P(t)$, this mutant is considered *killed*. If not, the tester should enhance T with a test case that reveals the difference between m and P. If m and P are equivalent, then $P(t) = m(t)$ for all test cases that can be derived from P's input domain.

Mutation testing can be applied with two goals: (i) evaluation of the program under test (i.e. P) or (ii) evaluation

of the test data (i.e. T). In the first case, faults in P are uncovered when *fault-revealing* mutants are identified. Given that S is the specification of P, a mutant is said to be fault-revealing when it leads to the creation of a test case that shows that $P(t) \neq S(t), (t \in T)$ ([42] p. 536).

In the second case, mutation testing evaluates how sensitive the test set is in order to identify as many faults simulated by mutants as possible.

Mutation testing is usually performed in four steps [41]: (1) execution of the original program, (2) generation of mutants, (3) execution of the mutants and (4) analysis of the mutants. After each cycle of mutation testing, the current result is calculated through the mutation score, which is the ratio of the number of killed mutants to the total number of generated (non-equivalent) mutants. The mutation score is a value in the interval $[0, 1]$ that reflects the quality of the test set with respect to the produced mutants. The closer to 1 the mutant set is, the higher the quality of the test set [42].

Aspect-oriented programming

Aspect-oriented programming (AOP) [32] relies in the principle of *separation of concerns* (SoC) [43]. Software concerns, in general, may address both functional requirements (e.g. business rules) and non-functional properties (e.g. synchronisation or transaction management). In the context of AOP, a concern is handled as a coarse-grained feature that can be modularised within well-defined implementation units. In AOP, the so-called *crosscutting concerns* cannot be properly modularised within conventional units [32]. For example, in traditional programming approaches like procedural and object-oriented programming (OOP), code that implements a crosscutting concern usually appears scattered over several modules and/or tangled with other concern-specific code. Other (non-crosscutting) concern codes comprise the *base code* of the software.

To improve the modular implementation of crosscutting concerns, AOP introduces the notion of *aspects*. An aspect can be either a conceptual programming unit or a concrete, specific unit named *aspect* (as in widely investigated languages such as AspectJ[1] and CaesarJ[2]). Once both aspects and base code are developed, they are combined during a *weaving* process [32] to produce a complete system.

In AspectJ, which is the most investigated AOP language and whose implementation model has inspired the proposition of several other languages, aspects have the ability to modify the behaviour of a program at specific points during its execution. Each of the points at which aspectual behaviour is activated is called a *join point*. A set of join points is identified by means of a *pointcut descriptor* or simply *pointcut*. A pointcut is represented by a language-based matching expression that identifies a set of join points that share some common characteristic (e.g. based on properties or naming conventions). This selection ability is often referred to as *quantification* [44].

During the program execution, once a join point is identified, a method-like construct named *advice* may run, depending or not of some runtime checking routine. Advices can be of different types depending on the supporting technology. For example, in AspectJ, advices can be defined to run at three different moments when a join point is reached: *before*, *after* or *around* (in place of) it.

AspectJ can also perform structural modifications of modules that comprise the base code. These modifications are achieved by the so-called *intertype declarations* (ITDs). Examples of intertype declarations are the introduction of a new attribute or method into a base module or a change in the class' inheritance.

Structural-based viewpoint analysis

This section revisits the contributions of our research group on structural testing of AO programs. It addresses three main concerns of systematic testing: the establishment of underlying structural models (section 'Creating an underlying model'), the identification of relevant test requirements based on that models (section 'Deriving test requirements') and the difficult to analyse and cover such requirements (section 'Covering and analysing test requirements').

Creating an underlying model

As described in Section 'Structural testing', the basic idea behind structural testing criteria is to ensure that specific elements (control elements and data structures) in a program are exercised by a given test set, providing evidence of the quality of the testing activity. It is supposed that the underlying model represents the dynamic behaviour of programs based on static information to generate relevant test requirements. In general, such static information is extracted from the source code. However, there may be differences between what is extracted from source code and what is the real dynamic behaviour. In techniques such as OO programming, such differences can be seen in cases of, for example, member (e.g. method or attribute) overriding and method overloading. In such cases, a special representation of these cases in the underlying model can help to reveal problems related to the dynamic behaviour.

In AOP, this situation seems to be more critical. Underlying models for AO testing are often adapted from other paradigms and programming techniques. Such models adapt existing abstractions by simply adding nodes and edges to represent the integration of some aspectual behaviour with the base program [8, 15, 45]. This is a limitation because the gap between the static information used to build the underlying model in AOP and the its

dynamic behaviour is more evident. For example, AOP allows the use of different mechanisms, such as the *cflow* command or the *around* advice, which are inherently runtime-dependent.

To ameliorate the aforementioned problem, our research group applies a more sophisticated approach. We devised a series of underlying test models based on static information which are closer to the dynamic behaviour of the program. We consider specific situations that happens in OO and AO to be represented in the models and then generate relevant test requirements for testing dynamic behaviour of that program. We use the Java bytecode to generate the underlying model for programs written in Java and AspectJ [11, 14, 16, 24, 46]. We take advantage of the AspectJ weaving process to extract static information of two different programming languages from one unified representation (the Java bytecode). This reduces the gap between static information and dynamic behaviour of a program. Moreover, our approach handles some particular cases where the bytecode does not have sufficient information for building the underlying model. This is related to information that enables the generation of relevant test requirements for testing OO and AO programs such as overriding, recursion and around advice.

Deriving test requirements

Structural testing uses an internal perspective of the system to define testing criteria and derive test requirements. These test requirements aim at exercising the program's data structures and its control flow. To better analyse the issues of deriving test requirements in AO programs, we summarise some research that has proposed structural testing criteria for procedural and OO programs. Afterwards, we describe adapted (procedural and OO) criteria to AO programs and contrast them with AO-specific to emphasise the tricks of deriving test requirements in AO programs.

Structural requirements for procedural and OO programs

Control flow- and data flow-based criteria for procedural programs (e.g. all-nodes, all-edges and all-uses) are well-established. They date from 30 years ago [39] and have been evolved to address the integration level [47]. The underlying models explicitly show the internal logic of units and the data interactions when either unit or integration testing is on focus.

For OO programs, control flow and data flow criteria are evolutions of criteria defined for procedural programs. For instance, Harrold and Rothermel [48] addressed the structural testing of OO programs by defining data flow-based criteria for four test levels: intra-method, inter-method, intra-class and inter-class. The authors addressed only explicit unit interactions; dealing with polymorphic calls and dynamic binding issues—i.e. OO specificities—was listed as future work [48].

Inspired by Harrold and Rothermel's criteria, Vincenzi et al. [49] presented a set of testing criteria based on both control flow and data flow for unit (i.e. method) testing. Vincenzi et al. approach relies on Java bytecode analysis and is automated by the JaBUTi tool. As the reader can notice, unit interactions was again not addressed by the author.

Structural requirements for AO programs

In our research [11], we developed an approach for unit testing of AO programs considering a method or an advice as the unit under testing. We proposed a model to represent the control flow of a unit and the join points that may activate an advice. Special types of nodes, the so called *crosscutting nodes*, are included in the CFG to represent additional information about the type of advice that affects that point, as well as the name of the aspect the advice belongs to. Control flow and data flow testing criteria are proposed to particularly require paths that include the crosscutting nodes and their incoming and outgoing edges.

To address the integration level, we explored the pairwise integration testing of OO and AO programs [14]. In short, the approach combines two communicating units into a single graph. We also defined a set of control flow and data flow criteria based on such representation. Figure 1 exemplifies the integration of two units (caller and called). Note that one of the units is affected by a before advice, which is represented with the crosscutting node notation. Note that crosscutting nodes are represented as dashed, elliptical nodes.

Neves et al. [46] developed an approach for integration testing of OO and AO programs in which a unit is integrated with all the units that interact with it in a single level of integration depth. We presented an evolution [24] of the approaches presented by ourselves [11, 14] and by Neves et al. [46]. We augmented the integration of units considering deeper interaction chains (up to the deepest level), without making the integration testing activity too expensive, since we integrate units in a configurable level of integration depth. Such augmented integration approach also brings customised control flow and data flow criteria. We highlight that all the representation models we proposed relies on Java bytecode analysis; furthermore, they all represent crosscutting nodes using a special type of node as shown in Fig. 1.

Our most recent approach characterises the whole execution context for a given piece of advice in a model that represents the execution flow from the aspect perspective [16]. A set of control flow and data flow criteria was proposed to require the execution of paths related to base code-advice integration.

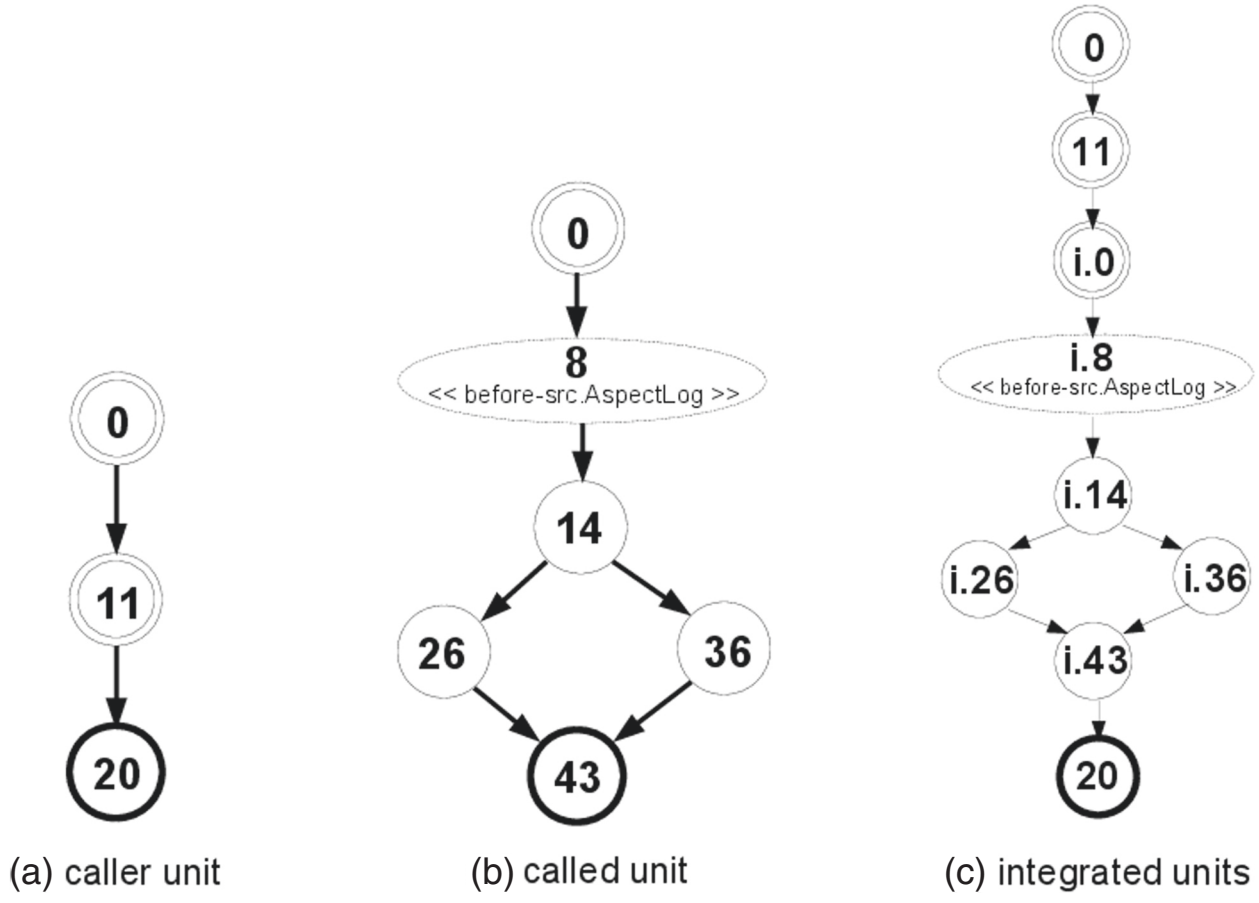

Fig. 1 Example of an integrated CFG for the pairwise approach [14]

Covering and analysing test requirements

In a series of preliminary assessment studies, we emphasised the effort required to cover test requirements derived from the proposed criteria for pairwise testing [14], multi-level integration testing [24] and pointcut-based integration testing [16]. A summary of the results is depicted in Table 1.

For each application we collected, for example, the number of test cases required to cover 100 % of all-nodes, all-edges and all-uses of each unit (#u.TCs in Table 1)

Table 1 Results of evaluation study of structural-based testing approaches

Application and basic metrics				Pairwise [14]			Multi-level integration [24]				Pointcut-based [16]		
								Max					
	#C	#A	#u	#u. TCs	#ad. TCs	%ad. TCs	#u. TCs	Depth Depth	#ad. TCs	%ad. TCs	#u. TCs	#ad. TCs	%ad. TCs
1. Stack	4	2	13	5	0	0	5	4	0	0	5	0	0
2. Subj-obs	5	2	14	6	0	0	6	2	0	0	6	0	0
3. Bean	1	1	15	5	0	0	5	4	0	0	5	0	0
4. Telecom	6	3	46	22	2	9	23	3	2	9	22	1	5
5. Music	10	2	45	19	3	16	22	4	4	18	19	3	16
6. Shape	5	1	52	25	6	24	14	6	21	150	25	0	0
Average	5.2	1.8	30.8	13.7	1.8	8.2	12.5	3.8	4.5	29.5	13.7	0.7	3.5

#C number of classes, *#A* number of aspects, *#u* number of units, *#u.TCs* number of tests for units, *#ad.TC* number of tests added to cover criteria, *%ad.TC* % of tests added to cover criteria

and the number of additional test cases required to cover requirements derived from the testing criteria of each approach (#ad.TCs in Table 1). Note that in these studies, we targeted optimal test sets with the minimum number of test cases as possible.

Analysing Table 1, we notice that in three applications (Stacks, Subj-obs and Bean), no additional effort was necessary considering all testing approaches. The other three applications (Telecom, Music and Shape) needed less than 25 % of additional test cases from the initial unit test set to cover the testing criteria of each approach. Thus, it is possible to say that the average of additional test cases needed to cover the requirements for integration testing criteria is not high. The cost of using these criteria is not high compared to the possible benefits achieved by applying such criteria. The only exception was the number of additional test cases of the multi-level integration approach in the Shape application. In this case, due to the depth considered during the generation of the test requirements, the cyclomatic complexity of some units largely increased the number of required test cases. In this way, we can say that, despite the possible applicability of the criteria, some of them may be heavily affected by structural characteristics of the implementation.

Despite the low number of additional test cases required to cover all test requirements of the proposed approaches, the analysis of the underlying model for creating test cases is not trivial. It is essential that the model facilitates the understanding of the dynamic behaviour of a program and thus the generation of relevant test cases.

The example of Fig. 2 illustrates how an *around* advice that is activated at a method call can be represented to enhance the comprehension of dynamic behaviour of a program. Is is obtained by applying the aforementioned multi-level integration approach by Cafeo and Masiero [24].

In this example, the node labelled with "0" represents the call to `m2` which happens inside `m1` (line 3). In this case, the CFG of the *around* advice is integrated in place of the `m2`'s CFG (this integration starts in node labelled with "(1).1.0"). Along the *around* execution, the `proceed` instruction may be invoked, depending on a predicate evaluation (line 18, which is included in the "(1).1.0" node). If the `proceed` is invoked, then the original join

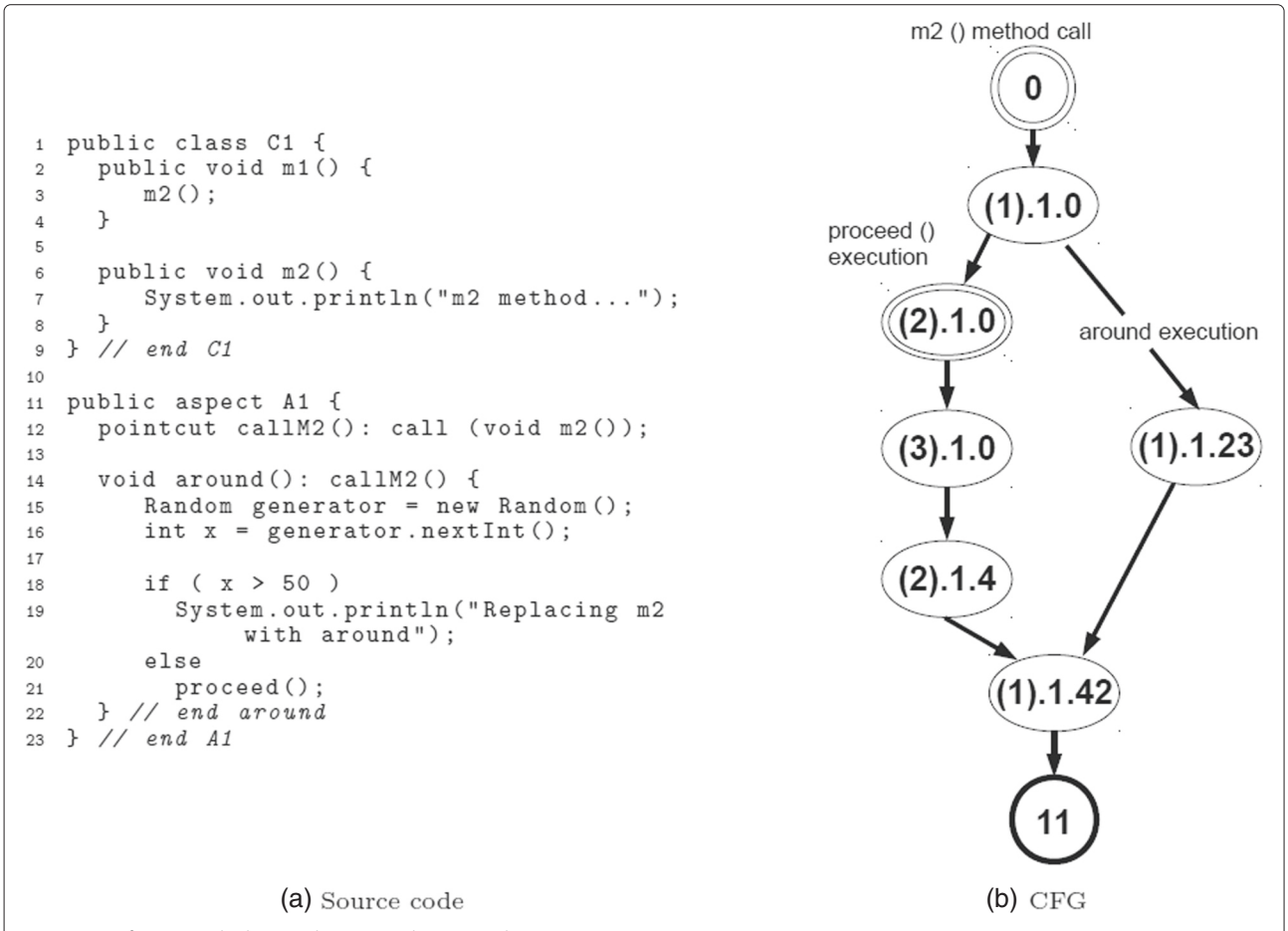

Fig. 2 CFG of an around advice with a proceed command

point is executed (nodes labelled with "(2).1.0", "(3).1.0" and "(2).1.4"). Alternatively, only *around* instructions are executed (this is represented by the node labelled with "(1).1.23").

In short, the CFG shown in Fig. 2 tries to represent an execution that depends on a runtime evaluation by showing the replacement of the join point by the advice and the return of the execution flow to the join point by the execution of the proceed command.

Related work on structural-based testing of AO programs

To the best of our knowledge, few approaches and testing criteria have been defined for structural testing of AO programs. Table 2 shows a list of them. Such pieces of work either propose testing approaches or explore internal implementation details of the software to support testing activity. The studies were selected based on recent literature and on a systematic review about AO software testing [50]. For each work listed in the table, the following information is presented: authors (Authors), year of publication (Year), testing level (Level), whether the approach defines testing criteria (Criteria) and whether the work implements a supporting tool (Tools). The table highlights in italics the contributions that are not from our research group in order to compare them with our work.

Zhao [8, 51] developed a data-flow testing approach for AO programs, based on the OO approach proposed by Harrold et al. [48], addressing the testing of interfaces from the aspect perspective and from the class perspective. Differently from the contributions of our research group, Zhao considers a unit to be a class or an aspect and relies on source code analysis to enables the graph generation.

Table 2 List of related work on structural testing of AO programs

Number	Authors	Year	Level	Criteria	Tools
1	*Zhao [51]*	*2002*	*Unit*	*N*	*Y*
2	*Zhao [8]*	*2003*	*Unit*	*Y*	*N*
3	*Xie and Zhao [52]*	*2006*	*–*	*Y*	*Y*
4	Lemos and Masiero [11]	2007	Unit	Y	Y
5	*Bernardi and Lucca [45]*	*2007*	*Integration*	*Y*	*Y*
6	*Xu and Rountev [53]*	*2007*	*Unit*	*N*	*Y*
7	Lemos et al. [14]	2009	Integration	Y	Y
8	Neves et al. [46]	2009	Integration	Y	Y
9	*Wedyan and Gosh [15]*	*2010*	*Integration*	*Y*	*Y*
10	Lemos and Masiero [16]	2011	Integration	Y	Y
11	Cafeo and Masiero [24]	2011	Integration	Y	Y
12	*Mahajan et al. [25]*	*2012*	*–*	*N*	*N*
13	*Wedyan et al. [29]*	*2015*	*Integration*	*Y*	*Y*

Y/N yes/no, — not mentioned

Xie and Zhao [52] presented an approach for structural- and state-based testing with support of a framework called Aspectra. In their approach, the framework generates wrapper classes. These classes are the input of a tool that generates test cases for aspectual behaviour considering structural and state-based coverage. This approach is a mixed approach (structural- and state-based) focusing on test case generation. Our contributions focus on proposing different control flow and data flow testing criteria for AO programs.

Bernardi and Lucca [45] also proposed a similar approach to our work [14, 16, 24, 46]. They defined a graph to represent the interactions between a base program and the pieces of advice interacting with it. They also defined some control flow-based criteria from such model. However, their approach does not incorporate data-flow analysis. Furthermore, to the best of our knowledge, no implementation of the approach has been presented yet.

Xu and Rountev [53] proposed an approach for regression testing of AO programs. This approach uses a control flow graph to analyse additional behaviour added by aspects as a way of generating regression testing requirements. Despite using control flow graph and proposing a tool for generating test requirements, Xu and Rountev did not propose testing criteria for AO programs.

Mahajan et al. [25] applied genetic algorithm to improve data flow-based test data generation. In this approach, the authors use the CFG to generate the data flow model of the program under test (i.e. def-use graph). Based on this information, they apply a genetic algorithm on it with many different parameters. The goal is to generate several test sets in order to reach 100 % of coverage of the all-uses criterion. Differently from the contributions of our research group, Mahajan et al. focus on generating test sets based on structural information instead of presenting an approach with an underlying model and testing criteria.

Finally, Wedyan and Gosh [15] and Wedyan et al. [29] presented an approach and tool implementation for measuring data flow coverage based on state variables defined in base classes or aspects. The goal of the approach is to prevent faults resulting from interactions (i.e. data flow) between base classes and aspects by focusing on attributes responsible for change the behaviour of both (state variables). Similarly to the work of our research group, they also define data flow criteria for AO programs. However, they only focus on the interaction between base classes and aspects established by the so-called state variables.

Mutation-based viewpoint analysis

Similarly to section 'Structural-based viewpoint analysis', this section revisits the contributions of our research

group on fault-based testing (in particular, mutation testing) of AO programs.

Creating an underlying model

As introduced in section 'Fault-based testing and mutation testing', fault-based testing relies on fault models and fault taxonomies—that is, sets of prespecified faults [40]. For AO software, fault taxonomies are mostly based on the pointcut–advice–intertype declaration (ITD) model implemented in AspectJ. We proposed a preliminary fault taxonomy for AO programs that take into consideration only faults related to pointcuts [54]. Afterwards, we identified, grouped together and added to our taxonomy several fault types for AO software that have been described by other researchers [1–5]. Additionally, we included new fault types that can occur in programs written in AspectJ [7, 12].

In total, our taxonomy encompasses 26 different fault types distributed over four categories. Category F1 includes eight pointcut-related fault types that address, for instance, incorrect join point quantification, misuse of primitive pointcut designators and incorrect pointcut composition rules. Category F2 includes nine fault types that regard ITD- and declare-like expressions. Examples of fault types in this category are improper class member introduction, incorrect changes in exception-dependent control flow and incorrect aspect instantiation rules. Category F3 describes six types of faults related to advice definition and implementation. Examples of F3 fault types are improper advice type specification, incorrect advice logic and incorrect advice-pointcut binding. Finally, category F4 includes three faults types whose root causes can be assigned to the base program. For instance, code evolution that causes pointcuts to break and duplicated crosscutting code due to improper concern refactoring.

We used the taxonomy to classify 104 faults documented from three medium-sized AO systems [7]. The chart of Fig. 3 summarises the distribution. In the x-axis, fault types 1.1–1.8 are related to pointcuts, 2.1–2.9 are related to ITDs, 3.1–3.6 are related do advices and 4.1–4.3 are related to the base code. Overall, the taxonomy has shown to be complete in terms of fault categories. It also helped us to characterise recurring faulty implementation scenarios[3] that should be checked during the development of AO software.

Deriving test requirements

According to section 'Fault-based testing and mutation testing', mutation testing [41] is a largely explored fault-based criterion. Based on a fault taxonomy, mutation operators are designed to insert faults into a program (i.e. to create the mutants). The mutants are used to evaluate the ability of the tests to reveal those artificial faults. In this context, in this section, we first summarise how mutation operators have been designed for procedural and OO paradigms. Then, we contrast this process with the designing of AO operators.

Mutation operators for procedural and OO programs

Agrawal et al. [55] designed a set of unit mutation operators—77 operators in total—for C programs, which was based on an existing set of 22 mutation operators for Fortran [56]. Although the number of C-based mutation operators is much larger than the number of Fortran-based ones, Agrawal et al. explain that their operators are either customisations or extensions of the latter, however considering the specificities of the C language.

Delamaro et al. [57] addressed the mutation testing of procedural programs at the integration level. The authors characterised a set of integration faults related to communication variables (i.e. variables that are related to the communication between units such as formal parameters, local and global variables and constants).

They then proposed the interface mutation criterion, which focuses on communication variables and encompasses a set of 33 mutation operators for C programs.

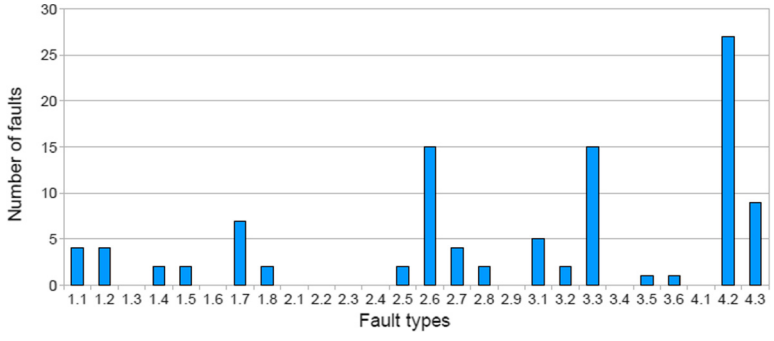

Fig. 3 Distribution of faults through the analysed systems

In 2004, Vincenzi [58] analysed the applicability of these two sets of C-based operators in the context of OO programs. The author focused on C++ and Java programs. With a few customisations and restrictions, Vincenzi concluded that most of the operators are straightforwardly applicable to programs written in these two languages.

The 24 inter-class mutation operators for Java programs proposed by Ma et al. [59] intend to simulate OO-specific faults. They focus on changes of variables but also address the modification of elements related to inheritance and polymorphism (e.g. deletion of an overriding method or class field or removal of references to overridden methods and fields). This is clearly an attempt to address paradigm-specific issues, even though some preliminary assessment has shown that the operators are not effective in simulating non-trivial faults [60][4].

Based on this brief analysis, we conclude that designing those operators was a "natural" evolution of operators previously devised for procedural programs, despite addressing different testing levels (i.e. unit and integration testing) and fault types. A few exceptions regard some class-level operators [59] which still require assessment through empirical studies.

Mutation operators for AO programs

Similarly to structural-based approaches for OO programs, all the mentioned sets of mutation operators can be applied to AO programs. However, they are not intended to cover AOP-specific fault types.[5] To apply mutation testing to AO programs properly, one must consider the new concepts and, in particular, the AOP mechanisms together with fault types that can be introduced into the software. The design of mutation operators for AO programs must take these factors into account.

In our previous research, we designed a set of 26 mutation operators for AspectJ programs [12]. The operators address instances of several fault types (18 in total) described in the taxonomy mentioned in section 'Creating an underlying model'. In particular, the operators simulate instances of faults within the first three categories (the groups are named G1, G2 and G3, each one simulating faults of categories F1, F2 and F3, respectively). These fault types are strictly related with the main concepts introduced by AOP.

In a preliminary assessment study, we checked the ability of the operators to simulate non-trivial faults [26]. We applied the operators to 12 small AspectJ applications and ran the non-equivalent mutants on a functional-based test set. Table 3 summarises the study results. It includes some metrics for the systems (e.g. the number of classes and aspects); the number of mutants by group of operators; the number of equivalent, anomalous and live mutants; the number of mutants killed by the original test set; and the number of test cases that have been added to kill the mutants that remained alive.

Regarding the numbers of mutants for each group of operators (columns four to six in the table), we can observe that changes applied to pointcuts (i.e. operators from G1 group) yield the largest number of mutants for all systems except for FactorialOptimiser. In total, they represent nearly 76 % of mutants (703 out of 922). This was expected since G1 is the largest operator group,

Table 3 Results of evaluation study of mutation-based testing

Application	#C[a]	#A	Mut. G1	Mut. G2	Mut. G3	Total	Equiv. Aut.	Equiv. Man.l	Anom.	Alive	Killed by original TCs	Added TCs
1. BankingSystem	9	6	108	2	26	136	68	–	18	50	50	–
2. Telecom	6	3	82	2	27	111	46	10	12	53	31	4
3. ProdLine	8	8	158	0	41	199	125	–	16	58	58	–
4. FactorialOptimiser	1	1	14	0	15	29	8	1	6	15	14	–
5. MusicOnline	7	2	47	0	10	57	25	2	5	27	22	2
6. VendingMachine	1	3	82	2	29	113	58	13	8	47	23	5
7. PointBoundsChecker	1	1	46	0	24	70	32	–	10	28	28	–
8. StackManager	4	3	34	0	11	45	24	–	0	21	21	–
9. PointShadowManager	2	1	38	0	12	50	25	5	4	21	13	2
10. Math	1	1	16	0	4	20	13	2	0	7	4	1
11. AuthSystem	3	2	45	0	7	52	28	1	3	21	17	2
12. SeqGen	8	4	33	0	7	40	19	8	3	18	4	3
Total	51	35	703	6	213	922	471	42	85	366	285	19

[a]It considers only relevant classes, excluding the driver ones

and the mutation rules encapsulated in these operators addresses varied parts of pointcuts. Nonetheless, as discussed in the next section, the analysis step for pointcut-related mutants can be partially automated, thus reducing the effort required for this task.

Covering and analysing test requirements

According to the results presented in Table 3, the operators were able to introduce non-trivial faults into the systems. In total, 39 mutants remained alive after their execution against the respective test sets in 7 out of 12 systems.

The main point with respect to covering and analysing mutation-based test requirements regarded the analysis of mutants to figure out if we needed to either classify them as equivalent or devise new test cases to kill them. The analysis of conventional mutants (i.e. derived from non-AO programs) is typically unit-centred; the task is concentrated on the mutated statement and perhaps on its surrounding statements. For AO mutants, on the other hand, detecting the equivalent ones may require a broader, in-depth analysis of the woven code.[6] This is due to the quantification and obliviousness properties [44] that are realised by AOP constructs such as pointcuts, advices and declare-like expressions.

In the sequence, we present an example to illustrate scenarios in which in-depth system analyses were required in order to classify mutants as equivalent.

The code excerpts shown in Fig. 4 characterise a scenario in which an in-depth analysis was required. It consists of a pair advice–pointcut and a pointcut mutant produced by the PWIW operator (pointcut weakening by insertion of wildcards) for the MusicOnline system, which consists in an online music service presented by Bodkin and Laddad [62]. The mutation is the replacement of a naming part of the pointcut (i.e. the `owed` attribute that appears in line 2) with the "∗" wildcard. At the first view, the mutant could not be classified as equivalent, since the mutant pointcut matched four join points in the base code, while the original pointcut matched only three. This additional activation of the *after returning* advice represents undesired control flow. However, the extra advice execution did not produce an observable failure. In this case, the advice logic sets the account status—suspended or not—according to the current credit limit. The extra advice execution would set the `suspended` attribute as false twice in a roll; nevertheless, the system behaves as expected despite this undesired execution control flow. Consequently, this mutant must be classified as equivalent. For this mutant, the conclusion is that even though the mutation impacted on the quantification of join points, the behaviour of the woven application remained the same.

Mutations such as the one shown in Fig. 4 requires dynamic analyses of the woven code to help one identify (un)covered test requirements, since the aspect-base interactions cannot be clearly seem at the source code level. Even though current IDEs such as AJDT[7] provide the developer with hints about the relationship about aspects and the base code, understanding the behaviour of the woven application to decide about equivalence regarding semantics cannot be feasible based only on static information. On the other hand, as shown in Table 3, many mutants were automatically classified as equivalent.[8] In total, around 50 % of the mutants were automatically classified as equivalent (471 out or 922). They are all pointcut-related mutants, and the automatic detection of the equivalent ones is based on the analysis of *join point static shadows* [63]. If two pointcuts capture the same set of join points, they are considered equivalent, despite the dynamic residues left in the base code during the weaving process.

Recently, we investigated the cost reduction of mutation testing based on the identification of sufficient mutation operators [28]. We ran the sufficient procedure [64] on a group of 12 small AspectJ applications, which are the same

Original advice & pointcut:

```
1   after(Account account) returning:
2         set(int Account.owed) && this(account) {
3     if (account.getOwed() > account.getCreditLimit())
4       account.suspended = true;
5     else account.suspended = false;
6   }
```

Mutant:

```
1   after(Account account) returning:
2 >       set(int Account.*) && this(account) {
3   ...
```

Fig. 4 Example of an equivalent mutant of the MusicOnline application

applications tested by Ferrari et al. [26]. The procedure output is a subset of mutation operators that can keep the effectiveness of a reduced test suite in killing mutants produced by all operators. The results of our study point out a five-operator set that kept the mutation score close to 94 %, with a cost reduction of 53 % with respect to the number of mutants we had do deal with.

Related work on mutation-based testing

Apart from our contributions, some other researchers have been investigating fault based-testing for AO programs, mainly focusing on mutation testing. Their initiatives are summarised in Table 4. Such pieces of work either customise the mutation testing for AO programs, apply the criterion as a way of assessing other testing approaches, or describe a tool. Again, the studies were selected based on recent literature and on a systematic review about AO software testing [50]. For each work listed in the table, the following information is presented: authors (Authors), year of publication (Year), whether the approach customises mutation testing to be applied to AO programs and whether the work implements a supporting tool (Tools). The table highlights in italics the contributions that are not from our research group in order to compare them with our work.

Mortensen and Alexander [9] defined three mutation operators to strengthen and weaken pointcuts and to modify the advice precedence order. However, the authors did not provide details of syntactic changes and implications of each operator. In our research [12], we precisely described the mutations performed by each operator.

Table 4 List of related work on fault-based testing of AO programs

Number	Authors	Year	Criteria	Tools
1	*Mortensen and Alexander [9]*	*2005*	*Y*	*N*
2	Lemos et al. [54]	2006	N	N
3	*Anbalagan and Xie [13]*	*2008*	*Y*	*Y*
4	Ferrari et al. [12]	2008	Y	N
5	*Delamare et al. [23]*	*2009*	*Y*	*N*
6	Ferrari et al. [21]	2010	N	Y
7	*Wedyan and Ghosh [17]*	*2012*	*N*	*N*
8	*Omar and Ghosh [18]*	*2012*	*Y*	*Y*
9	Ferrari et al. [26]	2013	Y	N
10	Levin and Ferrari [27]	2014	N	N
11	Lacerda and Ferrari [28]	2014	N	N
12	*Parizi et al. [22]*	*2015*	*N*	*Y*
13	Leme et al. [65]	2015	Y	Y

Y/N yes/no

Anbalagan and Xie [13] automated two pointcut-related mutation operators defined by Mortensen and Alexander [9]. Mutants are produced through the use of wildcards as well as by using naming parts of original pointcut and join points identified from the base code. Based on heuristics, the tool automatically ranks the most representative mutants, which are the ones that more closely resemble the original pointcuts. The final output is a list of the ranked mutants; no other mutation step is supported. The set of mutation operators proposed in our research [12] includes and refines the operators defined by Anbalagan and Xie.

Delamare et al. [23] proposed an approach based on test-driven development concepts and mutant analysis for testing AspectJ pointcuts. Their goal was to validate pointcuts by means of test cases that explicitly define sets of join points that should be affected by specific advices. A mutation tool named AjMutator [20] implements a subset of our pointcut-related operators [12]. The mutant pointcuts are used to validate the effectiveness of their approach.

More recently, Wedyan and Ghosh [17] proposed the use of simple object-based analysis to prevent the generation of equivalent mutants for some mutation operators for AspectJ programs. They argue that reducing the amount of equivalent mutants generated by some operators would consequently reduce the cost of mutation testing as a whole. The authors used three testing tools (namely, AjMutator [20], Proteum/AJ [21] and MuJava [60]) to assess their technique. Apart from traditional class-level mutation operators [59], Wedyan and Ghosh applied a subset of operators defined in our previous work [12] using the Proteum/AJ and AjMutator tools.

Omar and Ghosh [18] presented four approaches to generate higher order mutants for AspectJ programs. The approaches were evaluated in terms of the ability to create mutants of higher order resulting in higher efficacy and less effort when compared with first order mutants. All approaches proposed can produce higher order mutants that can be used to increase testing effectiveness and reduce testing effort and reduce the amount of equivalent mutants. Differently from Omar and Ghosh's work, our work only considers first order mutations.

Parizi et al. [22] presented an automated approach for random test case generation and uses mutation testing as a way of assessing their approach. Basically, their automated framework analyses AspectJ object code (i.e. Java bytecodes) and exercises compiled advices (i.e. Java methods) as a way of validating the implementation of crosscutting behaviour. Mutants are generated with a modified version of the AjMutator tool [20], which implements a subset of operators defined in our previous work [12].

Reuse-centred viewpoint analysis

As discussed in section 'Introduction', AOP is typically applied to refactor existing systems to achieve better modularisation of crosscutting concerns [2, 33–36]. Given this scenario of AOP adoption, our recent work investigated the difficulty of testing AO and OO programs, in particular when there is a migration from one paradigm to the other. In particular, we aim to analyse the following: (i) the effort required to adapt a test suites from one paradigm to the other and vice versa, given that two equivalent programs (regarding their semantics) are available (one OO and another AO) and (ii) the structural code coverage yielded by such adapted test suites.

Results of objective (i)—effort to adapt test sets—were presented by Levin and Ferrari [27] and are summarised in section 'Effort to adapt test sets across paradigms'. In this paper, we extend Levin and Ferrari's work by testing a hypothesis using statistical procedures. In the sequence, section 'Structural coverage yielded by test sets across paradigms' brings novel results regarding objective (ii)—structural coverage of adapted test sets. We start by describing the study configuration, including target applications and applied procedures.

Study configuration

We identified 12 small applications plus one medium-sized application for which we fully created functional-based test sets in conformance with the systematic functional testing (SFT) criterion [66]. In short, SFT combines equivalence partitioning and boundary-value analysis [38] aiming at associating the benefits of functional testing (independent implementation) and greater code coverage on test [66]. The test set must include at least two test cases that cover each equivalence class and one test case to cover each boundary value. According to the SFT proponents, this minimises problems of coincidental correctness.

Table 5 brings general information for each application. Note that six applications have a "DP" suffix and consist of randomly selected examples of design patterns implemented by Hannemann and Kiczales [67]. Other columns show the number of classes (#C) and the number of aspects (#A) in each system.[9]

These applications were selected because they all had OO and AO equivalent implementations developed by third-party researchers. Furthermore, their source code was either available for download or listed in the original

Table 5 Target applications—study of reuse of test sets across paradigms

Application name	Description	Total LOC OO/AO	#C	#C/#A
1. AbstractFactory (DP)	Creates the initial GUI that allows the user to choose a factory and generate a new GUI with the elements that the respective factory provides [67]	90/97	4	4/1
2. Boolean	Testing boolean formulas with terms AND, OR, XOR, NOT and variables [68, 69]	301/316	12	10/2
3. Bridge (DP)	Decouple an abstraction from its implementation so that the two can vary independently [67]	76/82	6	6/1
4. Chess	Chess game containing GUI [35]	1155/945	13	13/1
5. Interpreter (DP)	This system implements an interpreter for a language of boolean expressions [67]	118/126	8	8/1
6. VendingMachine	VendingMachine consists in an application for a vending machine into which the customer inserts coins in order to get drinks [70]	209/245	9	9/1
7. Question Database	Facilitate the management, reuse and improving collection of questions of evidence prepared by teachers [71]	6447/6479	27	27/5
8. ATM-log	Manager application of the bank account [35]	496/519	12	11/1
9. ChainOfResponsability (DP)	This system implements an GUI interface based in design pattern ChainOfResponsability [67]	96/150	5	5/2
10. Flyweight (DP)	This system show on the screen a message with characters in upper or lower case according with the parameters [67]	44/61	4	4/2
11. Memento (DP)	This system records a value in a point of execution [67]	29/64	2	3/ 2
12. ShopSystem	Simplified e-commerce system [69]	360/381	10	8/8
13. Telecom	This system calculates and reports the charges and duration of phone calls (local and long distance calls) [72]	186/197	8	8/2

#C number of classes, *#C/#A* number of classes and aspects

reports (references can be found in Table 5). The specifications of these applications, which we used to define test requirements, were either documented in the original reports or were elaborated after analysing the source code. For Question Database (application #7 in Table 5), due to its size, we only tested non-functional concerns that are implemented with aspects in the AO version. Obviously, such concerns are also present in the OO implementation, though spread across or tangled with code of other concerns.

To design and perform the tests, initially we defined two groups of applications (namely, group A and group B), each one including OO and AO implementations of six programs plus two concerns[10] of the Question Database system. In group A, we created SFT-adequate test sets for the OO implementation (i.e. test sets written purely in Java). Then, we adapted test cases to make them executable in the AO equivalent implementations. On the other way around, in group B, we firstly created test sets for the AO implementations, then adapted such test sets to the OO counterparts.

Table 6 illustrates the specification of functional test requirements for an operation of the ATM-log application. The table shows the input/output conditions, the valid and invalid (equivalence) classes and the boundary values. This template of specification was applied to all tested systems and guided the creation of test cases for both groups of applications (group A and group B). The last three columns of Table 8 summarise the number of test requirements and the number of test cases with respect to each target application.

Effort to adapt test sets across paradigms

As described by Levin and Ferrari [27], in this investigation, we wanted to study the effect of different programming paradigms on the effort required to migrate (i.e. adapt) test code from OO to AO programs and vice versa. To extend the original analysis [27], we define the hypotheses listed in Table 7 (namely, H1, H2, H3 and H4). Note that the hypotheses are related to metrics described in the sequence and assume that there is no difference between the effort required to migrate test sets across OO and AO implementations (i.e. they represent null hypotheses).

Table 7 Hypotheses formulated for effort-related analysis

	(Null) hypotheses
H1	$\text{TOTAL-LOC-TC}_{OO \leftrightarrow AO} = \text{TOTAL-LOC-TC}_{AO \leftrightarrow OO}$
H2	$\text{ADD}_{OO \leftrightarrow AO} = \text{ADD}_{AO \leftrightarrow OO}$
H3	$\text{MOD}_{OO \leftrightarrow AO} = \text{MOD}_{AO \leftrightarrow OO}$
H4	$\text{REM}_{OO \leftrightarrow AO} = \text{REM}_{AO \leftrightarrow OO}$

Metrics and tool: The metrics we collected to evaluate the effort required to adapt test sets across paradigms focus on code churn. Code churn is generally used to predict the defect density in software systems, and it is easily collected from a system change history [73]. Usually, this kind of metric is used to compare system versions to measure how many lines were added, changed and removed. In particular, we collected the following: *Total-LOC-TC*—number of non-commented LOC in the test classes; *ADD*—number of lines added to the new version of a test class; *MOD*—number of lines changed in the new version of the test class in comparison with its previous version; and *REM*—number of lines removed from the previous version of a test class to create a new version. Note that by 'new version', we mean the test class that has been adapted to the new paradigm. We used the Meld tool[11] to provide visual support in the analysis of code changes between different implementations of the same application.

Results and analysis: Table 8 summarises results regarding the collected metrics. On average, for group A, adapting OO test sets to AO implementations required additions of 5.70 %, modifications in 4.46 % of test code lines, with no code removal in any application. For group B, on the other hand, more modifications and removals were needed than for group A. On average, test code was 9.57 % modified and 3.10 % removed to conform with OO implementations, while only 1.93 % lines were added to the test code.

Overall, our preliminary findings were that (i) less code is written for testing OO programs, specially because

Table 6 Example of a specification of functional test requirements for the withdraw operation (ATM-log system)

Input condition	Valid class	Invalid class	Boundary value
Withdrawn value "v"	(C1) v $\leqslant$ account balance	(I1) v> account balance	(B1) v = 0
			(B2) v = account balance
			(B3) v = account balance + 1
Output condition	Valid class		
Success message	(O1) "successful withdraw"		
Logging message	(O2) operation is logged		

Table 8 Results of test adaptation effort measurement—group A and group B

Application Name	Total LOC TC	%. size diff.	Churn LOC TC						Test requirements		
			ADD	%ADD	MOD	%MOD	REM	%REM	Equiv. classes	Bound. values	Total TC
`Group A: OO - OA`											
1. AbstractFactoryOO	20								4	1	4
AbstractFactoryOA	20	0	0	0	0	0	0	0			
2. BooleanOO	29								7	3	7
BooleanOA	37	+27.58	8	27.58	1	3.44	0	0			
3. BridgeOO	76								16	2	16
BridgeOA	76	0	0	0	0	0	0	0			
4. ChessOO	281								28	13	39
ChessOA	302	+7.47	21	7.47	8	2.84	0	0			
5. InterpreterOO	47								48	2	48
InterpreterOA	47	0	0	0	0	0	0	0			
6. VendingMachineOO	41								9	10	10
VendingMachineOA	43	+4.87	2	4.87	5	12.19	0	0			
7. QuestionDatabaseOO	47								5	3	8
QuestionDatabaseOA	47	0	0	0	6	12.76	0	0			
Average		+5.70		5.70		4.46		0			
`Group B: OA - OO`											
8. ATM-logOA	111								9	5	15
ATM-logOO	111	0	0	0	4	3.6	0	0			
9. ChainOfResponsabilityOA	108								6	0	6
ChainOfResponsabilityOO	96	−11.11	0	0	18	16.66	12	11.11			
10. FlyweightOA	36								4	4	4
FlyweightOO	36	0	2	5.55	4	11.11	2	5.55			
11. MementoOA	31								2	2	3
MementoOO	31	0	0	0	8	25.8	0	0			
12. ShopSystemOA	256								22	35	30
ShopSystemOO	256	0	0	0	0	0	0	0			
13. TelecomOA	257								12	16	23
TelecomOO	244	−5.05	0	0	15	5.83	13	5.05			
7. QuestionDatabaseOA	50								5	5	6
QuestionDatabaseOO	54	+8	4	8	2	4	0	0			
Average		−1.16		1.93		9.57		3.10			

test cases for AO implementations required more specific code to expose context information to build JUnit assertions, and (ii) test code for OO programs conforms better with the open-closed principle [74], since a higher number of changes were required to make test sets of group B executable in OO implementations and (iii) test code for OO programs is more reusable, which is reflected by the *MOD* and *REM* averages that indicate recurring interventions in test sets for AO systems in order to adapt them to OO implementations.

Figure 5 shows an example of how the test set for the Chess application was adapted from the OO implementation to the AO counterpart. Different test code lines are 15 (OO version) and 15–17 (AO version). In the first case, the `srtErrorMsg` attribute of the `pawn` object is used in the assertion. In the migrated (AO) test code, the `aspectOf()` AspectJ-specific method is used to allow the retrieval of the context information (i.e. the error message). In this example, the ADD metric accounts for 2 and MOD metric accounts for 1, respectively.

Original test case for ChessOO:

```
1  public void testPawnMovement_I4(){
2
3          startRow = 6; startColumn = 5;
4          desRow = 6; desColumn = 6;
5
6          assertFalse(pawn.legalMove(startRow, startColumn, desRow,
               desColumn,
7          cellMatrix.getPlayerMatrix(), player1 ));
8
9          startRow = 5; startColumn = 5;
10         desRow = 6; desColumn = 5;
11
12         assertFalse(pawn.legalMove(startRow, startColumn, desRow,
               desColumn,
13         cellMatrix.getPlayerMatrix(), player1 ));
14
15         assertEquals("Ilegal move", pawn.strErrorMsg);
16 }
```

Adapted (migrated) test case for ChessAO:

```
1  public void testPawnMovement_I4(){
2
3          startRow = 6; startColumn = 5;
4          desRow = 6; desColumn = 6;
5
6          assertFalse(pawn.legalMove(startRow, startColumn, desRow,
               desColumn,
7          cellMatrix.getPlayerMatrix(), player1 ));
8
9          startRow = 5; startColumn = 5;
10         desRow = 6; desColumn = 5;
11
12         assertFalse(pawn.legalMove(startRow, startColumn, desRow,
               desColumn,
13         cellMatrix.getPlayerMatrix(), player1 ));
14
15         ErrorMsg em = ErrorMsg.aspectOf();
16         String saida = em.getErrorMsg();
17         assertEquals("Ilegal move", saida);
18 }
```

Fig. 5 Test case example for the Chess application

To evaluate if the preliminary findings have statistical relevance, we tested the hypotheses defined in Table 7. Initially, we checked whether the data has normal distribution. For this, we applied the Shapiro-Wilk test. Results are summarised in Table 9.

Note that, for statistical significance, we adopted the traditional confidence of 95 %; thus, our analysis considers p values below 0.05 significant. For all statistical tests, we used the R language and environment.[12]

As the reader can notice, apart from TOTAL-LOC-$TC_{AO\leftrightarrow OO}$ and $MOD_{AO\leftrightarrow OO}$, all other p values are below the defined threshold of 0.05. Therefore, the null hypotheses (that is, the data has normal distribution) are rejected.

We then applied the non-parametric Mann-Withney test to compare the effort to migrate test sets across the two paradigms, given that such test does not assume normal distributions [75]. Results are summarised in Table 10.

The results reveal that, even though the preliminary findings favoured the OO paradigm regarding the analysed test sets (and their reuse), this could be not assessed with statistical rigour. Overall, the null hypotheses could

Table 9 Results of Shapiro-Wilk test for effort-related metrics

		p-value
Group A	TOTAL-LOC-$TC_{OO\leftrightarrow AO}$	0.00515
	$ADD_{OO\leftrightarrow AO}$	0.00515
	$MOD_{OO\leftrightarrow AO}$	0.01878
	$REM_{OO\leftrightarrow AO}$	0.00000
Group B	TOTAL-LOC-$TC_{AO\leftrightarrow OO}$	0.26280
	$ADD_{AO\leftrightarrow OO}$	0.00098
	$MOD_{AO\leftrightarrow OO}$	0.37110
	$REM_{AO\leftrightarrow OO}$	0.02156

Table 10 Results of Mann-Whitney test for effort-related metrics

	(Null) hypotheses test	*p* value
H1	$\text{TOTAL-LOC-TC}_{OO \leftrightarrow AO} = \text{TOTAL-LOC-TC}_{AO \leftrightarrow OO}$	0.10160
H2	$\text{ADD}_{OO \leftrightarrow AO} = \text{ADD}_{AO \leftrightarrow OO}$	0.35770
H3	$\text{MOD}_{OO \leftrightarrow AO} = \text{MOD}_{AO \leftrightarrow OO}$	0.17490
H4	$\text{REM}_{OO \leftrightarrow AO} = \text{REM}_{AO \leftrightarrow OO}$	0.07541

not be rejected due to the low probability of perceiving difference between the two paradigms with respect to the analysed issue. One should notice that not rejecting a hypothesis does not mean the hypothesis is accepted. In fact, we cannot accept a null hypothesis, but only find evidence against it. In our case (i.e. results presented in section 'Effort to adapt test sets across paradigms'), possible explanations for the lack of statistical significance of preliminary findings may rely on (i) the small number of analysed programs (14, in total) or (ii) the impossibility of showing differences between the two paradigms (i.e. there is no difference between them at all). Case (i) will be addressed in our future work, as stated in section 'Final remarks, limitations and research directions'. The impossibility (and consequent conclusion) regarding case (ii) can be assessed with the enlargement of our application sets.

Structural coverage yielded by test sets across paradigms

With the aim of assessing the quality of reused test sets, we now analyse the structural coverage that can be achieved when test sets are reused across paradigms. In other words, we want to study the effect of different programming paradigms on the test coverage with respect to the structure (statements and branches) of OO and AO programs. This investigation develops in terms of the hypotheses defined in Table 11:

Metrics and tool: To evaluate H5 and H6, we computed the code coverage yielded by SFT-adequate test sets considering the same groups of applications (i.e. group A and group B). The metrics we collected are statement coverage and branch coverage, which are similar to the all-nodes and all-edges traditional control flow-based criteria. For the Question Database system, we focused the analysis on parts of the code affected by the crosscutting behaviour that, in the AO implementation, was encapsulated within one or more aspects. For the remaining (small) applications, we considered the full code (base code and aspects, if any) for computing test requirements and coverage.

Table 11 Hypotheses formulated for coverage-related analysis

	(Null) hypotheses
H5	$\text{STATEMENT}_{OO \leftrightarrow AO} = \text{STATEMENT}_{AO \leftrightarrow OO}$
H6	$\text{BRANCH}_{OO \leftrightarrow AO} = \text{BRANCH}_{AO \leftrightarrow OO}$

The metrics collection task was automated by EclEmma[13], which is code coverage analysis tool developed as an Eclipse plugin. Note we had to manually inspect the coverage of AspectJ implementations due to the fact that EclEmma, as other Java-based coverage tools, processes ordinary bytecode (i.e. Java compiled code) to trace the traversed paths during test execution. When it comes to AspectJ, the standard ajc[14] compiler adds some structures to the compiled bytecode that are not recognised by EclEmma. These structures correspond to specific AOP constructions. For example, for each pointcut in the source code, the ajc compiler adds a method to the bytecode. Such method is often created only for retaining pointcut-related information that could be lost after compilation. However, EclEmma treats this spurious method as code that should be equally covered by the tests and hence must not be considered for coverage purposes. Such spurious requirements were spotted and discarded through a manual inspection step.

We highlight that the JaBUTi/AJ tool, developed by our group to support AO-specific structural criteria [11, 14, 16, 24, 46], is able to compute test requirements and trace the execution for particular modules of a system under testing, depending on the chosen level of integration. In other words, JaBUTi/AJ instruments and runs specific parts of a system, according to the tester's selection. Since we intended to compute the coverage for all system modules, to speed up the process, we adopted EclEmma. Such tool is able to run full test set in a single run and compute the coverage of the full application, even though manual inspection was necessary to achieve precise results.

Results and analysis: Table 12 shows the results regarding statement and branch coverage for small applications of group A and group B. Visual representation can be found in Figs. 6 and 7. Similarly, Table 15 and Figs. 8 and 9 present results for the Question Database application, though separately from the other small applications. Note that results with respect to the Question Database application will be later discussed in this section.

Regarding small applications, Table 12 and Figs. 6 and 7 indicate that there are only minimal differences in coverage when both criteria are considered. In group A, test sets yielded average statement coverage of 90.5 and 89.5 % for OO and AO implementations, respectively. For branch coverage in the same group, averages are 78.9 and 77.9 %. Individual differences can be checked in columns labelled with "diff %". Despite the lower coverages obtained for applications of group B, the values for different paradigms are again very close: 86.3 % of covered statements for OO implementations and 84.9 % for AO counterparts and

Table 12 Statement and branch coverage for small applications

Application name	% Covered Statem.	diff %	# Statem.	# Covered/ missing statem.	% Covered branches	diff %	# Branches	# Covered/ missing branches
`Group A: OO - AO`								
1. AbstractFactoryOO	100.0		134	134/0	n/a		n/a	n/a
AbstractFactoryAO	100.0	0.0	147	147/0	n/a	n/a	n/a	n/a
2. BooleanOO	87.5		431	377/54	66.7		24	16–8
BooleanAO	85.5	−2.0	532	455/77	70.8	4.2	24	17–7
3. BridgeOO	100.0		120	120/0	100.0		4	4/0
BridgeAO	100.0	0.0	151	151/0	100.0	0.0	4	4/0
4. ChessOO	75.8		955	724/231	63.8		232	148/84
ChessAO	76.8	1.0	964	740/224	65.3	1.5	248	162/86
5. InterpreterOO	92.0		225	207/18	71.4		14	10/4
InterpreterAO	85.5	−6.5	290	248/42	78.6	7.2	14	11/3
6. VendingMachineOO	87.9		321	282/39	87.5		16	14/2
VendingMachineAO	89.1	1.2	366	326/40	80.0	−7.5	5	4/1
Average OO	90.5				77.9			
Average AO	89.5	−1.0			78.9	1.1		
`Group B: AO - OO`								
8. ATM-logAO	72.7		326	237/89	71.4		14	10/4
ATM-logOO	80.8	0.0	271	219/52	83.3	11.9	12	10/2
9. ChainOfResponsabilityAO	76.7		257	197/60	68.8		16	11/5
ChainOfResponsabilityOO	77.7	1.1	157	122/35	66.7	−2.1	18	12/6
10. FlyweightAO	82.5		120	99/21	87.5		8	7/1
FlyweightOO	85.4	2.9	82	70/12	75.0	-12.5	8	6/2
11. MementoAO	100.0		112	112/0	n/a		n/a	n/a
MementoOO	100.0	0.0	44	44/0	n/a	n/a	n/a	n/a
12. ShopSystemAO	85.7		1581	1355/226	75.6		41	31/10
ShopSystemOO	82.6	−3.1	872	720/152	73.8	−1.9	80	59/21
13. TelecomAO	91.8		477	438/39	100.0		20	20/0
TelecomOO	91.6	−0.2	381	349/32	100.0	0.0	20	20/0
Average AO	84.9				67.2			
Average OO	86.3	1.4			66.5	−0.8		

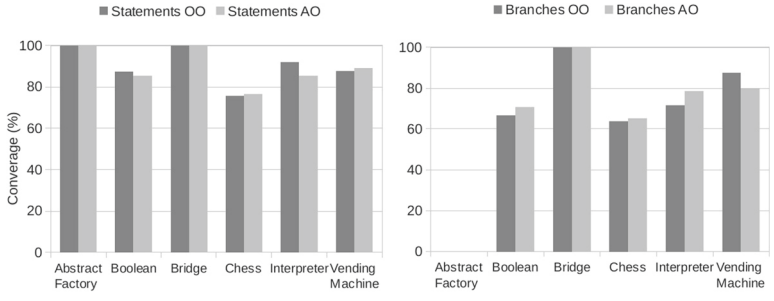

Fig. 6 Statement and branch coverage for small applications—group A

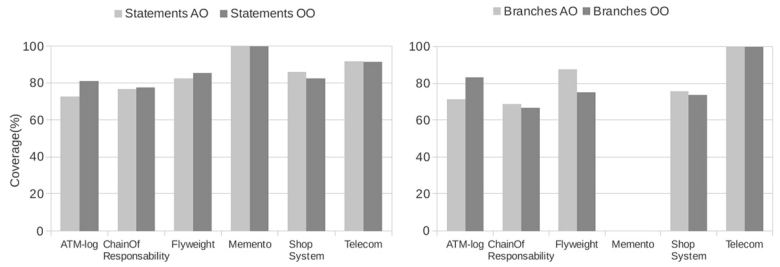

Fig. 7 Statement and branch coverage for small applications—group B

66.5 % and 67.2 % of covered branches for OO and AO implementations, respectively.

To evaluate whether such minimal coverage differences have statistical relevance, we tested the hypotheses defined in Table 11, considering the differences amongst coverages (statements and branches—"dif %" columns) in both groups and paradigms. Initially, again, we checked whether the data has normal distribution. For this, we applied the Shapiro-Wilk test. Results are summarised in Table 13.

Differently from the analysis presented in section 'Effort to adapt test sets across paradigms', the p values obtained for the STATEMENT_i and BRANCH_i metrics are all above the defined threshold of 0.05. Thus, the null hypotheses (that is, the data has normal distribution) cannot be rejected. We then applied the Student's t test to compare the structural coverage yielded by test sets that are originally built for programs written under one paradigm (namely, OO and AO) and then migrated to the other one. Results are summarised in Table 14. Note that the null hypotheses cannot be rejected, since p values are above 0.05.

The results confirms the preliminary findings that, for small applications, there is no difference between the two paradigms with respect to (control flow-based) structural coverage when test sets are reused across them.

Differently from results for small applications, tests executed on Question Database resulted in higher statement and branch coverage in all OO implementations (see Table 15 and Figs. 8 and 9). For example, for the Time concern in the OO implementation, statement and branch coverages were 72.2 and 44.6 %, respectively, while the same measures for the AO version were 25.8 and 3.8 %. On average, statement and branch coverage in group A were 59.2 and 33.9 % for OO implementation and 24.1 and 4.3 % for the AO implementation, respectively. Similar results (in terms of higher coverage for OO implementation) are observed for group B.

The numbers for the Question Database system have some peculiarities. Firstly, considering both paradigms, test execution resulted in low coverage rates for all concerns (the only exception is TimeOO—see Table 15). As mentioned in the beginning of this section, for this system the coverage analysis focused only on modules—aspects and classes—which are related to the selected crosscutting concerns. For them, the tool computed test requirements and their respective coverage. Despite this concern-driven analysis, we emphasise that test cases were designed to those particular concerns and hence did not exercise substantial parts of the involved modules.

Secondly, and equally important, we can notice a much higher number of test requirements in the AO implementations. Such difference relies basically on two reasons: (i) the generality of the aspect possibly to facilitate system evolution without breaking pointcuts and (ii) the strategy adopted by the developer to create aspects (and their internal parts) using AspectJ mechanisms. Both reasons are related to the conservative procedure to define

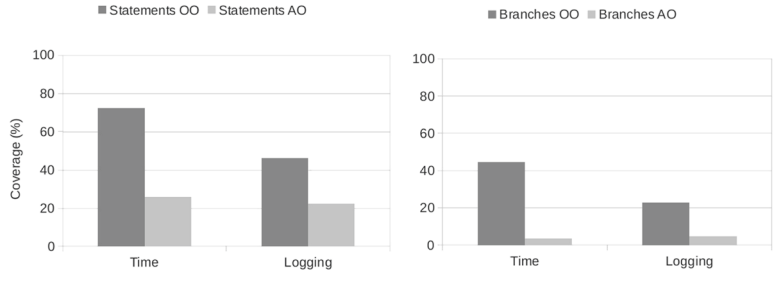

Fig. 8 Statement and branch coverage for Question Database—group A

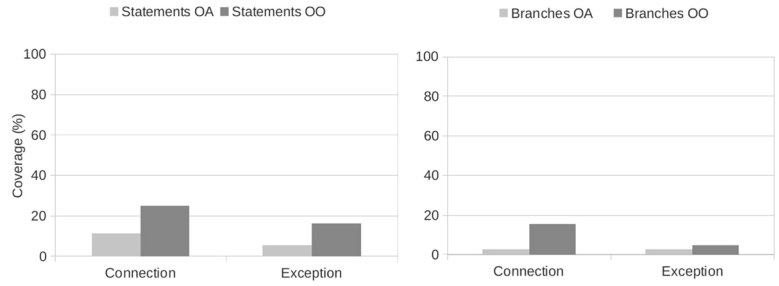

Fig. 9 Statement and branch coverage for Question Database—group B

pointcuts with wide scope[15]—i.e. they select a high number of join points—and with the advice activation logic that is resolved at runtime by the executing environment (see Fig. 10, which is described in the sequence). Besides this, the weaving process performed by the ajc compiler adds complexity to the internal logic of the base code. Examples of such added complexity are advice calls, which may or may not be nested within conditional structures inserted before, after or in place (around) the selected join points.

Figure 10 shows an example of a highly generic pointcut named `printStackTrace`, which captures pointcuts of the whole system, except from the `ExceptionLogging` aspect itself. The weaving of the associated *around* advice with the base code inserts, at each join point, conditional structures to decide on join point activation. As a consequence, a high number of statements and branches are processed as test requirements by the coverage tool, even though exceptions will never be raised in part of the selected join points (i.e. unfeasible requirements).

We call the reader's attention to the fact that, in the context of small systems (in which join point quantification is somehow restricted to a few modules) we can conclude systematically developed test sets, when properly adapted from one paradigm to the other, may result in similar code coverage levels. However, as long as the quantification of join points increases (as in the case of the Question Database system), the existing test set produces higher coverage in OO code. The cause may be the conservative approach for using AOP constructs such as pointcuts and advice. From a developer's perspective, widely scoped pointcuts may ease the evolution of programs without causing pointcuts to break (this problem was observed in a previous study of fault-proneness of AO evolving AO programs [34]). Besides this, delegating the advice activation decision to the executing environment is also a facilitating strategy. However, from the tester's perspective, advanced and automated program analysis techniques are required to avoid the substantial increase in the number of test requirements to be analysed.

Table 13 Results of Shapiro-Wilk test for coverage-related metrics

		p value
Group A	$STATEMENT_{OO \leftrightarrow AO}$	0.05591
	$BRANCH_{OO \leftrightarrow AO}$	0.75190
Group B	$STATEMENT_{AO \leftrightarrow OO}$	0.57470
	$BRANCH_{AO \leftrightarrow OO}$	0.57750

Related work

This section summarises related work that addresses issues for testing AO programs (including some proposals for dealing with such issues) and studies that compare the testing of programs developed under different paradigms. Note that sections 'Related work on structural-based testing of AO programs' and 'Related work on mutation-based testing' have summarised more specific-related research (namely, related to structural and mutation testing of AO programs).

Ceccato et al. [3] discussed the difficulties for testing AO programs in contrast with OO programs. They argued that if aspects could be tested in isolation, AO testing should be easier than OO testing. According to them, code that implements a crosscutting concern is typically spread over several modules in OO systems, thus hardening test design and evaluation. At some extent, our findings with respect to the quality of test sets applied to OO and AO implementations go against these observations. The results indicate lower quality (in terms of code coverage) in the AO paradigm when concern scattering grows even with the execution of systematically developed test sets. Ceccato et al. also proposed a testing strategy to integrate base code and aspects incrementally. However, they did

Table 14 Results of *t* test for coverage-related metrics

	(Null) hypotheses test	*p* value
H5	$STATEMENT_{OO \leftrightarrow AO} = STATEMENT_{AO \leftrightarrow OO}$	0.43660
H6	$BRANCH_{OO \leftrightarrow AO} = BRANCH_{AO \leftrightarrow OO}$	0.67740

Table 15 Statement and branch coverage for Question Database

Application Name	% Covered Statem.	# Statem.	# Covered/ Missing Statem.	% Covered Branches	# Branches	# Covered/ Missing Branches
`Group A: OO - AO`						
1. TimeOO	72.2	2329	1681/648	44.6	112	50/62
TimeAO	25.8	7971	2059/5912	3.8	1028	39/989
2. LoggingOO	46.3	605	280/325	23.1	26	6/20
LoggingAO	22.4	2015	451/1564	4.8	228	11/217
Average OO	59.2			33.9		
Average AO	24.1			4.3		
`Group A: OO - AO`						
1. ConnectionAO	11.3	3384	381/3003	2.7	413	11/402
ConnectionOO	24.9	977	243/734	15.4	26	4/22
2. ExceptionAO	5.5	7500	409/7091	2.7	308	8/300
ExceptionOO	16.2	2199	357/1842	5.0	126	6/120
Average AO	8.4			2.7		
Average OO	20.6			10.2		

not report any kind of evaluation of their strategy as we did in the previous sections of this paper.

Zhao and Alexander [76] proposed an approach to test AspectJ AO programs as OO programs. Based on a decompilation process, AspectJ applications can be tested as ordinary Java applications using conventional approaches. Although this may ease the tests, it may impose other obstacles specially when a fault is detected in the decompiled code. In such case, identifying the fault in the original—i.e. aspectual—code may become unfeasible due to code transformations that occur during the forwards and backwards compilation/weaving processes. Differently, in this paper we summarised a set of testing approaches that are directly applied to AspectJ programs, without requiring any decompilation step.

Xie and Zhao [77] discussed existing solutions for AO testing such as test input generation, test selection and runtime checking, mostly developed by the authors. For instance, their tools support automatic test generation based on compiled AspectJ aspects (i.e. classes as bytecodes). They also discussed unit and integration testing of aspects using wrapping mechanisms, control flow- and data flow-based testing focused on early versions of AspectJ and mutation testing applied to code obtained from refactoring aspects into ordinary Java classes. Differently from our work, Xie and Zhao did not present assessment studies neither selected examples extracted from practical evaluation.

With respect to test reuse and evaluation across paradigms, Prado et al. [78] and Campanha et al. [79] compared procedural and OO programming using a set of programs from the data structures domain (e.g. queues, stacks and lists). The two pieces of research focus on structural and mutation testing, respectively. The results of Prado et al. study show that there is no evidence for the existence of differences in cost and strength between

```
1  public aspect ExceptionLogging {
2
3    pointcut printStackTrace(): execution(* *.*(..)) && !within(
         ExceptionLogging);
4
5    declare soft: Exception: withincode(* *.*(..)) && !within(
         ExceptionLogging);
6
7    void around() : execution(void *.*(..)) && !within(ExceptionLogging){
8       try{
9          proceed();
10      }
11      catch (Exception ex){
12         System.err.println("Exception: "+ex.getCause());
13      }
14 }
```

Fig. 10 Example of conservative (weak) pointcut of the Question Database application

procedural and OO paradigms. This is similar to our results for small-sized OO and AO applications (details in section 'Structural coverage yielded by test sets across paradigms'). The results of Campanha et al. study, applied to the same domain and programs, show that both cost and the strength of the mutation testing are higher in programs implemented in the procedural paradigm than in the OO paradigm. No comparison with our results is possible, given that we did not apply mutation testing to assess the quality of reused test sets.

Final remarks, limitations and research directions

A report recently published summarised the contributions of the Brazilian community to the 'world' of AO Software Development [80]. For testing, five key challenges are listed: (1) identifying new potential problems; (2) defining proper underlying models; (3) customising existing test selection criteria and/or defining new ones; (4) providing adequate tool support; and (5) experimenting and assessing the approaches. We can add another key challenge to this group: (6) reuse of test sets to validate AO-based software refactorings.

In spite of the challenges addressed by our research (mainly challenges 1–4 and 6), a major open issue enumerated by Kulesza et al. [80] concerns the lack of experimental studies to assess the usefulness and feasibility of AO testing approaches, as well as the generalisation of results. With respect to this, results of the preliminary studies presented in sections 'Structural-based viewpoint analysis', 'Mutation-based viewpoint analysis' and 'Reuse-centred viewpoint analysis' represent only initial evaluation stage. Other studies that address AO systems larger than the ones used in the preliminary evaluation, as well as larger samples, are indeed necessary, though not available for the time being. For instance, for medium-sized AO systems, we have estimated the effort to cover structural requirements derived from the pointcut-based approach based on a theoretical analysis [16]. Besides this, we have also roughly estimated the cost of mutation testing in terms of number of mutants for medium-sized AO systems [26]. However, only by creating adequate test suites for such systems one shall be able to draw stronger conclusions about the feasibility and usefulness of AO-specific test selection criteria.

We highlight that this limitation is general in regard to research on AO testing and, at some extent, to some other research on AO software development [53, 81, 82]. Overall, other research on AO testing addressed only small-sized applications [10, 13, 15, 23, 45]. Just a few studies and approaches that may be related to testing (e.g. characterisation of bug patterns for exception handling [6] and AO refactoring supported by regression testing [2]) have handled larger AO systems, though with a different focus if compared to the evaluation we presented in section 'Reuse-centred viewpoint analysis'. In other cases, testing approaches are partially applied to larger systems; for example, as in the work of Parizi et al. [22], who applied a subset of our mutation operators [12] and limited the number of generated mutants per program.

As future work, we are planning cross-comparison studies considering test selection criteria of different techniques within the AO context. This shall enable us to empirically establish a subsume relation for the investigated criteria and to define incremental testing strategies. We also intend to target AO systems larger than the ones typically analysed in current research. The motivation is that designing a test case to exercise a large program path that includes integrated units, or analysing a mutant that has wide impact on join point quantification, is very likely to require effort and complexity that cannot be easily quantified only in terms of the number of test cases or the number of test requirements.

Other research initiatives from our group include enlarging our application sets to reproduce the studies that compare effort and quality of test suites developed for implementations in different paradigms and checking the ability of adapted test sets from one paradigm to another to reveal faults simulated by mutants. To do so, we can apply mutation operators incrementally, starting from unit mutation operators towards AOP-specific ones.

Endnotes

[1] http://www.eclipse.org/aspectj/—accessed on 23/07/2015.

[2] http://caesarj.org/—accessed on 23/07/2015.

[3] For more details of the fault classification and examples of faulty scenarios, the reader may refer to the work of Ferrari et al. [7]

[4] By non-trivial faults, we mean faults that are not easily revealed with an existing test set, be it systematically developed or not.

[5] It is likely that a test case designed to cover a fault modelled by a traditional (e.g. unit-level) mutation operator may also reveal a different, perhaps AOP-specific fault. However, it has been empirically shown [61] that context-specific test sets (e.g. for unit testing) may have reduced ability to reveal faults in a different context (e.g. at the integration level).

[6] The inter-class mutation operators for Java [59] pose a similar challenge: mutations of inheritance and polymorphism elements also require broad analyses of the compiled application.

[7] http://www.eclipse.org/ajdt/—accessed in 23/07/2015.

[8] The testing process and criterion application was supported by the Proteum/AJ tool [21]. More details can be found in a previous paper [26].

[9]OO implementations have only classes, while AO counterparts have both classes and aspects.

[10]In group A, concerns are time (security procedure that locks the screen after a given time without mouse movement or any pressed key) and logging. In group B, concerns are exception logging (raised exceptions are displayed to the user) and database connection control.

[11]http://meldmerge.org/—accessed on 23/07/2015.

[12]http://www.r-project.org/—accessed on 30/07/2015

[13]http://www.eclemma.org/—accessed on 23/07/2015.

[14]https://www.eclipse.org/aspectj/doc/next/devguide/ajc-ref.html—accessed on 23/07/2015.

[15]Also known as *weak pointcuts* [9, 13].

Competing interests
The authors declare that they have no competing interests.

Author's contributions
FCF developed mutation-based testing approaches and experimental evaluation. BBPC developed structural-based integration testing approaches and experimental evaluation. TGL and JTSL performed cross-paradigm test set reuse studies. OALL developed structural-based unit and integration testing approaches and experimental evaluation. JCM developed mutation-based and structural-based testing approaches. PCM developed structural-based testing approaches. All authors drafted, read and approved the final manuscript.

Acknowledgements
We thank the financial support received from CNPq (Universal Grant 485235/2013-7) and CAPES.

Author details
[1]Computing Department, Federal University of São Carlos, Rod. Washington Luis, km 235, 13565-905 São Carlos, SP, Brazil. [2]Informatics Department, Pontifical Catholic University of Rio de Janeiro, Rua Marquês de São Vicente, 225 RDC, 22451-900 Rio de Janeiro, RJ, Brazil. [3]Institute of Science and Technology, Federal University of São Paulo, Rua Talim, 330, 12231-280 São José dos Campos, SP, Brazil. [4]Computer Systems Department, University of São Paulo, Avenida Trabalhador São-carlense, 400, 13566-590 São Carlos, SP, Brazil.

Received: 1 August 2014 Accepted: 2 November 2015
Published online: 20 November 2015

References

1. Alexander RT, Bieman JM, Andrews AA (2004) Towards the systematic testing of aspect-oriented programs. Tech. Report CS-04-105, Dept. of Computer Science, Colorado State University, Fort Collins/Colorado - USA
2. van Deursen A, Marin M, Moonen L (2005) A systematic aspect-oriented refactoring and testing strategy, and its application to JHotDraw. Tech.Report SEN-R0507, Stichting Centrum voor Wiskundeen Informatica, Amsterdam - The Netherlands
3. Ceccato M, Tonella P, Ricca F (2005) Is AOP code easier or harder to test than OOP code? In: Proceedings of the 1st workshop on testing aspect oriented programs (WTAOP)—held in conjunction with AOSD, Chicago/IL - USA
4. Bækken JS, Alexander RT (2006) A candidate fault model for aspectj pointcuts. In: Proceedings of the 17th international symposium on software reliability engineering (ISSRE). IEEE Computer Society, Raleigh/NC - USA. pp 169–178
5. Zhang S, Zhao J (2007) On identifying bug patterns in aspect-oriented programs. In: Proceedings of the 31st annual international computer software and applications conference (COMPSAC). IEEE Computer Society, Beijing - China. pp 431–438
6. Coelho R, Rashid A, Garcia A, Ferrari F, Cacho N, Kulesza U, von Staa A, Lucena C (2008) Assessing the impact of aspects on exception flows: an exploratory study. In: Proceedings of the 22nd European conference on object-oriented programming (ECOOP). Springer, Paphos - Cyprus. pp 207–2345142
7. Ferrari FC, Burrows R, Lemos OAL, Garcia A, Maldonado JC (2010) Characterising faults in aspect-oriented programs: Towards filling the gap between theory and practice. In: Proceedings of the 24th Brazilian symposium on software engineering (SBES). IEEE Computer Society, Salvador/BA - Brazil. pp 50–59
8. Zhao J (2003) Data-flow-based unit testing of aspect-oriented programs. In: Proceedings of the 27th annual IEEE international computer software and applications conference (COMPSAC). IEEE Computer Society, Dallas/Texas - USA. pp 188–197
9. Mortensen M, Alexander RT (2005) An approach for adequate testing of AspectJ programs. In: Proceedings of the 1st workshop on testing aspect oriented programs (WTAOP)—held in conjunction with AOSD, Chicago/IL - USA
10. Xu D, Xu W (2006) State-based incremental testing of aspect-oriented programs. In: Proceedings of the 5th international conference on aspect-oriented software development (AOSD). ACM Press, Bonn - Germany. pp 180–189
11. Lemos OAL, Vincenzi AMR, Maldonado JC, Masiero PC (2007) Control and data flow structural testing criteria for aspect-oriented programs. J Syst Softw 80(6):862–882
12. Ferrari FC, Maldonado JC, Rashid A (2008) Mutation testing for aspect-oriented programs. In: Proceedings of the 1st international conference on software testing, verification and validation (ICST). IEEE, Lillehammer - Norway. pp 52–61
13. Anbalagan P, Xie T (2008) Automated generation of pointcut mutants for testing pointcuts in AspectJ programs. In: Proceedings of the 19th international symposium on software reliability engineering (ISSRE). IEEE Computer Society, Seattle/WA - USA. pp 239–248
14. Lemos OAL, Franchin IG, Masiero PC (2009) Integration testing of object-oriented and aspect-oriented programs: a structural pairwise approach for Java. Sci Comput Program 74(10):861–878
15. Wedyan F, Ghosh S (2010) A dataflow testing approach for aspect-oriented programs. In: Proceedings of the 12th IEEE international high assurance systems engineering symposium (HASE). IEEE Computer Society, San Jose/CA - USA. pp 64–73
16. Lemos OAL, Masiero PC (2011) A pointcut-based coverage analysis approach for aspect-oriented programs. Inf Sci 181(13):2721–2746
17. Wedyan F, Ghosh S (2012) On generating mutants for AspectJ programs. Inf Softw Technol 54(8):900–914
18. Omar E, Ghosh S (2012) An exploratory study of higher order mutation testing in aspect-oriented programming. In: Proceedings of the 23rd international symposium on software reliability engineering (ISSRE). IEEE Computer Society, Dallas/TX - USA. pp 1–10
19. Anbalagan P, Xie T (2006) Efficient mutant generation for mutation testing of pointcuts in aspect-oriented programs. In: Proceedings of the 2nd workshop on mutation analysis (mutation)—held in conjunction with ISSRE. Kluwer Academic Publishers, Raleigh/NC -USA. pp 51–56
20. Delamare R, Baudry B, Le Traon Y (2009) AjMutator: A tool for the mutation analysis of aspectj pointcut descriptors. In: Proceedings of the 4th international workshop on mutation analysis (mutation). IEEE, Denver/CO - USA. pp 200–204
21. Ferrari FC, Nakagawa EY, Rashid A, Maldonado JC (2010) Automating the mutation testing of aspect-oriented Java programs. In: Proceedings of the 5th ICSE international workshop on automation of software test (AST). ACM Press, Cape Town - South Africa. pp 51–58
22. Parizi RM, Ghani AA, Lee SP (2015) Automated test generation technique for aspectual features in AspectJ. Inf Softw Technol 57:463–493
23. Delamare R, Baudry B, Ghosh S, Le Traon Y (2009) A test-driven approach to developing pointcut descriptors in AspectJ. In: Proceedings of the 2nd international conference on software testing, verification and validation (ICST). IEEE Computer Society, Denver/CO - USA. pp 376–385
24. Cafeo BBP, Masiero PC (2011) Contextual integration testing of object-oriented and aspect-oriented programs: a structural approach for Java and AspectJ. In: Proceedings of the 25th Brazilian symposium on software engineering (SBES). IEEE Computer Society, São Paulo/SP - Brazil. pp 214–223
25. Mahajan M, Kumar S, Porwal R (2012) Applying genetic algorithm to increase the efficiency of a data flow-based test data generation approach. SIGSOFT Software Engineering Notes 37(5):1–5

26. Ferrari FC, Rashid A, Maldonado JC (2013) Towards the practical mutation testing of AspectJ programs. Sci Comput Program 78(9):1639–1662
27. Levin TG, Ferrari FC (2014) Is it difficult to test aspect-oriented software? Preliminary empirical evidence based on functional tests. In: Proceedings of the 11th workshop on software modularity (WMod). Brazilian Computer Society, Maceio/AL - Brazil
28. Lacerda JTS, Ferrari FC (2014) Towards the establishment of a sufficient set of mutation operators for AspectJ programs. In: Proceedings of the 8th Brazilian workshop on systematic and automated software testing (SAST). Brazilian Computer Society, Maceio/AL - Brazil
29. Wedyan F, Ghosh S, Vijayasarathy LR (2015) An approach and tool for measurement of state variable based data-flow test coverage for aspect-oriented programs. Inf Softw Technol 59:233–254
30. Muñoz F, Baudry B, Delamare R, Traon YL (2009) Inquiring the usage of aspect-oriented programming: an empirical study. In: Proceedings of the 25th international conference on software maintenance (ICSM). IEEE Computer Society, Edmonton/AB - Canada. pp 137–146
31. Rashid A, Cottenier T, Greenwood P, Chitchyan R, Meunier R, Coelho R, Südholt M, Joosen W (2010) Aspect-oriented software development in practice: tales from AOSD-Europe. IEEE Comput 43(2):19–26
32. Kiczales G, Irwin J, Lamping J, Loingtier JM, Lopes C, Maeda C, Menhdhekar A (1997) Aspect-oriented programming. In: Proceedings of the 11th European conference on object-oriented programming (ECOOP). Springer, Jyväskylä - Finland. pp 220–2421241
33. Mortensen M, Ghosh S, Bieman JM (2008) A test driven approach for aspectualizing legacy software using mock systems. Inf Softw Technol 50(7-8):621–640
34. Ferrari FC, Burrows R, Lemos OAL, Garcia A, Figueiredo E, Cacho N, Lopes F, Temudo N, Silva L, Soares S, Rashid A, Masiero P, Batista T, Maldonado JC (2010) An exploratory study of fault-proneness in evolving aspect-oriented programs. In: Proceedings of the 32nd international conference on software engineering (ICSE). ACM Press, Cape Town - South Africa. pp 65–74
35. Alves P, Santos A, Figueiredo E, Ferrari FC (2011) How do programmers learn AOP? An exploratory study of recurring mistakes. In: Proceedings of the 5th Latin American workshop on aspect-oriented software development (LA-WASP). Brazilian Computer Society, São Paulo/SP - Brazil. pp 65–74
36. Alves P, Figueiredo E, Ferrari FC (2014) Avoiding code pitfalls in aspect-oriented programming. In: Proceedings of the 18th Brazilian symposium on programming languages (SBLP). Springer, Maceió/AL - Brazil
37. Ferrari FC, Cafeo BBP, Lemos OAL, Maldonado JC, Masiero PC (2013) Difficulties for testing aspect-oriented programs: a report based on practical experience on structural and mutation testing. In: Proceedings of the 7th Latin American workshop on aspect-oriented software development (LA-WASP). Brazilian Computer Society, Brasília/DF - Brazil. pp 12–17
38. Myers GJ, Sandler C, Badgett T, Thomas TM (2004) The art of software testing. 2nd edn. John Wiley & Sons, Hoboken/NJ - USA
39. Rapps S, Weyuker EJ (1982) Data flow analysis techniques for program test data selection. In: Proceedings of the 6th international conference on software engineering (ICSE). IEEE Computer Society, Tokio - Japan. pp 272–278
40. Morell LJ (1990) A theory of fault-based testing. IEEE Trans Softw Eng 16(8):844–857
41. DeMillo RA, Lipton RJ, Sayward FG (1978) Hints on test data selection: help for the practicing programmer. IEEE Comput 11(4):34–43
42. Mathur AP (2007) Foundations of software testing. Addison-Wesley Professional, Toronto, Canada
43. Dijkstra EW (1976) A discipline of programming. Prentice-Hall, Englewood Cliffs/NJ - USA
44. Filman RE, Friedman D (2004) Aspect-oriented programming is quantification and obliviousness. In: Filman RE, Elrad T, Clarke S, Akşit M (eds). Aspect-oriented software development. Addison-Wesley, Boston. pp 21–35. Chap. 2
45. Bernardi ML, Lucca GAD (2007) Testing aspect oriented programs: an approach based on the coverage of the interactions among advices and methods. In: Proceedings of the 6th international conference on quality of information and communications technology (QUATIC). IEEE Computer Society, Lisbon - Portugal. pp 65–76
46. Neves V, Lemos OAL, Masiero PC (2009) Structural integration testing at level 1 of object- and aspect-oriented programs. In: Proceedings of the 3rd Latin American workshop on aspect-oriented software development (LA-WASP). Brazilian Computer Society, Fortaleza/CE - Brazil. pp 31–38. (in Portuguese)
47. Linnenkugel U, Müllerburg M (1990) Test data selection criteria for (software) integration testing. In: First international conference on systems integration. IEEE Computer Society, Morristown/NJ - USA. pp 709–717
48. Harrold MJ, Rothermel G (1994) Performing data flow testing on classes. In: Proceedings of the 2nd ACM SIGSOFT symposium on foundations of software engineering (FSE). ACM Press, New Orleans/LA - USA. pp 154–163
49. Vincenzi AMR, Delamaro ME, Maldonado JC, Wong WE (2006) Establishing structural testing criteria for java bytecode. Software: practice and experience 36(14):1513–1541
50. Ferrari FC (2010) A contribution to the fault-based testing of aspect-oriented software. PhD thesis, Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo (ICMC/USP), São Carlos/SP - Brasil
51. Zhao J (2002) Tool support for unit testing of aspect-oriented software. In: Workshop on tools for aspect-oriented software development—held in conjunction with OOPSLA, Seattle/WA - USA
52. Xie T, Zhao J (2006) A framework and tool supports for generating test inputs of AspectJ programs. In: Proceedings of the 5th international conference on aspect-oriented software development (AOSD). ACM Press, Bonn - Germany. pp 190–201
53. Xu G, Rountev A (2007) Regression test selection for AspectJ software. In: Proceedings of the 29th international conference on software engineering (ICSE). IEEE Computer Society, Minneapolis/MN - USA. pp 65–74
54. Lemos OAL, Ferrari FC, Masiero PC, Lopes CV (2006) Testing aspect-oriented programming pointcut descriptors. In: Proceedings of the 2nd workshop on testing aspect oriented programs (WTAOP)—held in conjunction with ISSTA. ACM Press, Portland/Maine - USA. pp 33–38
55. Agrawal H, DeMillo RA, Hathaway R, Hsu W, Hsu W, Krauser EW, Martin RJ, Mathur AP, Spafford EH (1989) Design of mutant operators for the C programming language. Technical Report SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette/IN - USA
56. Budd TA (1980) Mutation analysis of program test data. PhD thesis, Graduate School, Yale University, New Haven, CT - USA
57. Delamaro ME, Maldonado JC, Mathur AP (2001) Interface mutation: an approach for integration testing. IEEE Trans Softw Eng 27(3):228–247
58. Vincenzi AMR (2004) Object-oriented: definition, implementation and analysis of validation and testing resources. PhD thesis, ICMC/USP, São Carlos, SP - Brazil(in Portuguese)
59. Ma YS, Kwon YR, Offutt J (2002) Inter-class mutation operators for Java. In: Proceedings of the 13th international symposium on software reliability engineering (ISSRE). IEEE Computer Society Press, Annapolis/MD - USA. pp 352–366
60. Ma YS, Harrold MJ, Kwon YR (2006) Evaluation of mutation testing for object-oriented programs. In: Proceedings of the 28th international conference on software engineering (ICSE). ACM Press, Shanghai - China. pp 869–872
61. Vincenzi AMR (1998) Resources for the establishment of testing strategies based on the mutation technique. Master's thesis, ICMC/USP, São Carlos/SP - Brazil (in Portuguese)
62. Bodkin R, Laddad R (2005) Enterprise aspect-oriented programming. In: Tutorials of EclipseCon 2005. Online, Burlingame/CA - USA. http://www.eclipsecon.org/2005/presentations/EclipseCon2005_EnterpriseAspectJTutorial9.pdf - accessed on 23/07/2015
63. Hilsdale E, Hugunin J (2004) Advice weaving in AspectJ. In: Proceedings of the 3rd international conference on aspect-oriented software development (AOSD). ACM Press, Lancaster - UK. pp 26–35
64. Barbosa EF, Maldonado JC, Vincenzi AMR (2001) Toward the determination of sufficient mutant operators for C. The Journal of Software Testing, Verification and Reliability 11(2):113–136
65. Leme FG, Ferrari FC, Maldonado JC, Rashid A (2015) Multi-level mutation testing of Java and AspectJ programs supported by the ProteumAJv2 tool. In: Proceedings of the 6th Brazilian conference on software: theory and practice (CBSoft—tools session). (to appear). Brazilian Computer Society, Belo Horizonte/MG - Brazil

66. Linkman S, Vincenzi AMR, Maldonado JC (2003) An evaluation of systematic functional testing using mutation testing. In: Proceedings of the 7th international conference on empirical assessment in software engineering (EASE). Institution of Electrical Engineers, Keele - UK. pp 1–15
67. Hannemann J, Kiczales G (2002) Design pattern implementation in Java and AspectJ. In: Proceedings of the 17th ACM SIGPLAN conference on object-oriented programming, systems, languages, and applications (OOPSLA). ACM Press, Seattle/WA - USA. pp 161–173
68. Prechelt L, Unger B, Tichy WF, Brössler P, Votta LG (2001) A controlled experiment in maintenance comparing design patterns to simpler solutions. IEEE Trans Softw Eng 27(12):1134–1144
69. Bartsch M Empirical assessment of aspect-oriented programming and coupling measurement in aspect-oriented systems. PhD thesis, School of Systems Engineering - University of Reading, Reading - UK
70. Liu CH, Chang CW (2008) A state-based testing approach for aspect-oriented programming. J Inf Sci Eng 24(1):11–31
71. Chagas JDE, Oliveira MVG (2009) Object-oriented programming versus aspect-oriented programming. A comparative case study through a bank of questions. Technical report, Federal University of Sergipe, São Cristóvão - Brazil (in Portuguese)
72. (2014) The Eclipse Foundation: AspectJ Documentation. Online. http://www.eclipse.org/aspectj/docs.php - accessed on 23/07/2015
73. Nagappan N, Ball T (2005) Use of relative code churn measures to predict system defect density. In: Proceedings of the 27th international conference on software engineering (ICSE). IEEE Computer Society, St. Louis/MO - USA. pp 284–292
74. Meyer B (1988) Object-oriented software construction. Prentice-Hall, Upper Saddle River/NJ - USA
75. Shull F, Singer J, Sjøberg DIK (2007) Guide to advanced empirical software engineering. Springer, Secaucus, NJ, USA
76. Zhao C, Alexander RT (2007) Testing aspect-oriented programs as object-oriented programs. In: Proceedings of the 3rd workshop on testing aspect oriented programs (WTAOP). ACM Press, Vancouver - Canada. pp 23–27
77. Xie T, Zhao J (2007) Perspectives on automated testing of aspect-oriented programs. In: Proceedings of the 3rd workshop on testing aspect oriented programs (WTAOP). ACM Press, Vancouver/British Columbia - Canada. pp 7–12
78. Prado MP, Souza SRS, Maldonado JC (2010) Results of a study of characterization and evaluation of structural testing criteria between procedural and OO paradigms. In: Proceedings of the 7th experimental software engineering Latin American workshop (ESELAW), Goiânia/GO - Brasil. pp 90–99
79. Campanha DN, Souza SRS, Maldonado JC (2010) Mutation testing in procedural and object-oriented paradigms: an evaluation of data structure programs. In: Proceedings of the 24th Brazilian symposium on software sngineering (SBES). IEEE Computer Society, Salvador/BA - Brazil. pp 90–99
80. Kulesza U, Soares S, Chavez C, Castor Filho F, Borba P, Lucena C, Masiero P, Sant'Anna C, Ferrari FC, Alves V, Coelho R, Figueiredo E, Pires P, Delicato F, Piveta E, Silva C, Camargo V, Braga R, Leite J, Lemos O, Mendonça N, Batista T, Bonifácio R, Cacho N, Silva L, von Staa A, Silveira F, Valente MT, Alencar F, Castro J, et al. (2013) The crosscutting impact of the AOSD Brazilian research community. J Syst Softw 86(4):905–933
81. Rinard M, Salcianu A, Bugrara S (2004) A classification system and analysis for aspect-oriented programs. In: Proceedings of the 12th ACM SIGSOFT international symposium on foundations of software engineering (FSE). ACM Press, Newport Beach/CA - USA. pp 147–158
82. Burrows R, Taïani F, Garcia A, Ferrari FC (2011) Reasoning about faults in aspect-oriented programs: a metrics-based evaluation. In: Proceedings of the 19th international conference on program comprehension (ICPC). IEEE Computer Society, Kingston/ON - Canada. pp 131–140