Difficulties for Testing Aspect-Oriented Programs: A Report based on Practical Experience on Structural and Mutation Testing

Fabiano C. Ferrari*, Bruno B. P. Cafeo†, Otávio A. L. Lemos‡, José C. Maldonado§, Paulo C. Masiero§
*Computing Department – Federal University of São Carlos (UFSCar) – São Carlos – Brazil
Email: fabiano@dc.ufscar.br
†Informatics Department – Pontifical Catholic University of Rio de Janeiro (PUC-Rio) – Rio de Janeiro – Brazil
Email: bcafeo@inf.puc-rio.br
‡Department of Science and Technology – Federal University of São Paulo (UNIFESP) – S. J. Campos – Brazil
Email: otavio.lemos@unifesp.br
§Computer Systems Department – University of São Paulo (USP) – São Carlos – Brazil
Email: {jcmaldon, masiero}@icmc.usp.br

***Abstract*—Since the first discussions of new challenges posed by Aspect-Oriented Programming to software testing, the real difficulties of testing AO programs have not been properly analysed. Despite the customisation of traditional techniques such as structural and mutation testing, there is still a lack of discussions on how hard is to apply them to (even ordinary) AO programs based on practical experience. This papers deals with this issue. It discusses the difficulties of testing AO programs from two perspectives: structural-based testing and fault-based testing. 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. The conclusion is that building test models for AO programs, as well as deriving and covering test requirements, are not straightforward as is has been for procedural and object-oriented programs. Examples are provided to support the discussions. Current limitations are raised and directions for future research are pointed out.**

I. Introduction

In 2004, Alexander et al. [1] first discussed the challenges posed by Aspect-Oriented Programming (AOP) from the software testing perspective. 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, the authors 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, researchers have been mainly concerned with: (i) the characterisation of fault types and bug patterns [2, 3, 4, 5, 6]; and (ii) the definition of underlying test models, including test selection criteria [7, 8, 9, 10, 11, 12, 13, 14]. In particular, structural-based and mutation-based testing have been on focus by several research initiatives [7, 9, 10, 12, 15, 14, 13, 16, 17].

Despite this variety of approaches for testing AO software, too little has been reported about the tricks 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.

In previous research, we investigated the fault-proneness of AO programs based on faults identified during the testing of real-world AO applications [18]. 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. However, in those exploratory studies we have not applied any AO-specific testing approach; instead, we have used test sets built upon the object-oriented (OO) versions of the applications, and then used such test sets to evaluate the AO counterparts with some eventual test set evolution.

In this paper, we revisit our previous research results and discuss some tricks of testing AO programs from two perspectives: structural-based testing (Section III) and fault-based testing (Section IV). We analyse the impact of using AOP mechanisms on the testability of programs in terms of (i) the definition of the underlying models, (ii) the derivation of test requirements, and (iii) the coverage of the requirements. Note that Sections III and IV are split according these three items. The points presented in this paper rely on our practical experience of establishing customised approaches to test AO programs and on observations we have made in exploratory assessment of them. In the next section we start by describing related research and in Section V we summarise our discussions and point out future research directions.

II. Related Work

Ceccato et al. [4] discuss the difficulties for testing AO programs in contrast with OO programs. They argue that if aspects could be tested in isolation, AO testing should be easier

than OO testing. According to them, code that implements crosscutting concerns is spread over several modules in OO systems, thus hardening test design and evaluation. The authors also propose a testing strategy to integrate base code and aspect incrementally. However, they have not reported any kind of evaluation of their strategy as we do in the next sections of this paper.

Zhao and Alexander [19] also propose an approach to test AspectJ AO programs as OO programs. Based on a decompilation process, Java code tested 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 infeasible due code transformations that occur during the forwards and backwards compilation/weaving processes. Differently, in this paper we summarise a set of testing approaches that are directly applied to AspectJ programs, without requiring any decompilation step.

Xie and Zhao [20] discuss 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 discuss unit and integration testing of aspects using wrapping mechanisms, control flow- and data flow-based testing based 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 do not present evaluation studies neither select examples extracted from practical evaluation.

III. Structural-Based Viewpoint Analysis

A. Creating an Underlying Model

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. A *control-flow graph* (CFG) is generally used to represent the control flow of a program, where 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. For data flow-based testing, the *def-use graph* extends the CFG with information about the definitions and uses of variables [21].

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, overriding and recursion. 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 models by simply adding nodes and edges to represent the integration of some aspectual behaviour with the base program [7, 22, 14]. This is a problem 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.

The generation of the underlying model for AO programs requires a more sophisticated approach. To ameliorate the aforementioned problem, in our work [9, 12, 23, 16] we use the Java bytecode to generate the underlying model for programs written in Java and AspectJ. It takes advantage of the AspectJ weaving process to extract static information of two different programming languages from one unified representation (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.

B. Deriving Test Requirements

To better analyse the issues of deriving test requirements in AO programs, we summarise some research where structural testing criteria were proposed for procedural and OO programs. Afterwards, we describe adapted procedural and OO criteria to AO programs and contrast a specific AO approach with AO adapted criteria 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 [21], and have been evolved to address the integration level [24]. The underlying models explicitly show the internal logic of units and the data interactions when either unit or integration testing are on focus.

For OO programs, control flow and data flow criteria are evolutions of criteria defined for procedural programs. For instance, Harrold and Rothermel [25] 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 [25].

Inspired by Harrold and Rothermel's criteria, Vincenzi et al. [26] presented a set of testing criteria based on both control flow and data flow for unit (*i.e.* method) testing. Vincenzi et al.'s approach relies on Java bytecode analysis and is automated by the *JaBUTi* tool.

Structural requirements for AO programs: In our research [9], we developed an approach for unit testing of AO programs considering a unit to be a method or an advice. We proposed a model to represent the control flow of a unit and the join points where advices are going to be activated. 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 [12]. 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.

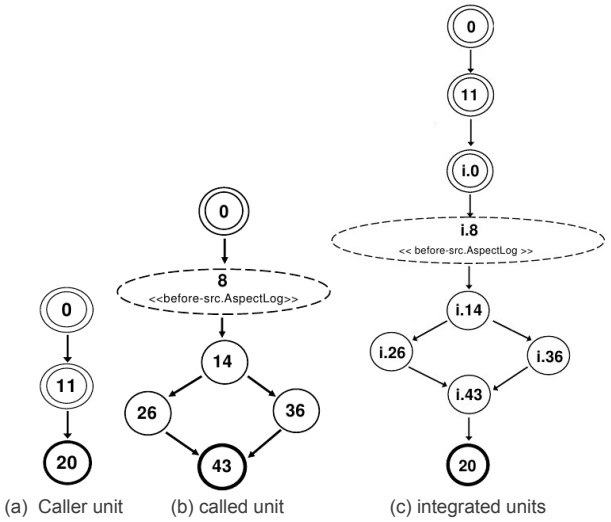

Fig. 1. Example of an integrated CFG for the pairwise approach [12].

Neves et al. [23] 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 [16] of the approaches presented by ourselves [9, 12] and by Neves et al. [23]. We enlarged 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. We also proposed control flow and data flow criteria for the proposed approach. We highlight that the representation model we proposed relies on Java bytecode analysis, likewise our previous and Neves et al. [23]'s approaches, and also represents crosscutting nodes using a special type of node.

We also proposed another approach to the integration testing of AO programs [13]. It characterises the whole execution context for a given piece of advice in a model that represents the execution flow from the aspect perspective [13]. A set of control flow and data flow criteria is proposed to require the execution of paths related to base code–advice integration.

C. 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 [12], multi-level integration testing [16] and pointcut-based integration testing [13]. A summary of the results is depicted in Table I.

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

Analysing Table I 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. 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.

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. An example where the underlying model helps with the understanding of the dynamic behaviour can be found in Figure 2.

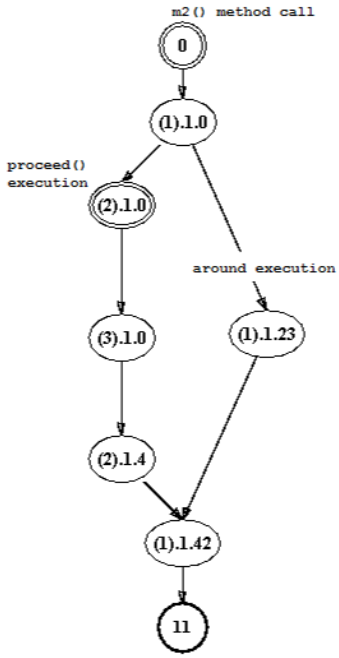

Fig. 2. CFG of an around advice with a proceed command.

The figure shows the CFG of the multi-level approach of the method *m1*. The *around* advice will be executed replacing the join point which is the call of method *m2*. Moreover, within the execution of the advice, there is a possibility of the execution of the proceed command which returns the execution flow to the join point. The CFG shown in Figure 2 tries to represent such execution 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.

TABLE I
RESULTS OF EVALUATION STUDY OF STRUCTURAL-BASED TESTING APPROACHES.

Application & basic metrics				Pairwise [12]			Multi-level integration [16]				Pointcut-based integration [13]		
	#C	#A	#u	#u.TCs	#ad.TCs	%ad.TCs	#u.TCs	Max 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

Legend:
#C: # of classes #A: # of aspects #u: # of units #u.TCs: # of tests for units #ad.TC: # of tests added to cover criteria %ad.TC: % of tests added to cover criteria

IV. MUTATION-BASED VIEWPOINT ANALYSIS

A. Creating an Underlying Model

Fault-based testing relies on fault models and fault taxonomies – that is, sets of prespecified faults [27]. They guide the selection of test cases that reveal those faults. Fault models and taxonomies can be devised from a combination of historical data, researchers' and practitioners' expertise, and specific programming paradigm concepts and technologies.

For AOP, faults taxonomies are mostly based on the pointcut–advice–ITD (intertype declaration) model implemented in AspectJ. We identified and grouped together several fault types for AO software that have been described by other researchers [1, 2, 3, 4]. Additionally, we included new fault types that can occur in programs written in AspectJ [10, 6].

The taxonomy encompasses 26 different fault types distributed over four categories, Category F1 includes eight pointcut-related fault types; for instance, incorrect join point quantification and incorrect pointcut composition rules. Category F2 includes nine fault types that regard ITD- and declare-like expressions. Examples are improper class member introduction and incorrect aspect instantiation rules. Category F3 includes six types of faults related to advice definition and implementation; for example, incorrect advice logic and incorrect advice-pointcut binding. Finally, F4 includes three faults types rooted in the base program. Examples are code evolution that causes pointcuts to break and duplicated crosscutting code.

We used the taxonomy to classify 104 faults documented from 3 medium-sized AO systems [6]. Overall, it has shown to be complete in terms of fault categories. It also helped us to characterise recurring faulty implementation scenarios that should be checked during the development of AO software.

B. Deriving Test Requirements

A largely explored fault-based criterion is the mutation testing [28]. Based on a fault taxonomy, mutation operators are designed to insert faults into a program, creating slightly modified versions of the program. These are called *mutants* and are used to evaluate the ability of the tests to reveal those artificial faults. Only mutants that behave differently from the original program are considered for computing test coverage.

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. [29] 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 [30]. Although the number C-based mutation operators is much larger than the number Fortran-based ones, Agrawal et al. explain that their operators are either customisations or extensions of the latter, however considering the C's specificities [29].

Delamaro et al. [31] addressed the mutation testing of procedural programs at the integration level. The authors proposed the Interface Mutation criterion, which focuses on communication variables and encompasses a set of 33 mutation operators for C programs.

In 2004, Vincenzi [32] analysed the applicability of these two sets of C-based operators in the context of object-oriented 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. [33] 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 [34].

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 [33] which still require to assessed 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. 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 [10]. The operators address instances of several fault types (18 in total) described in the taxonomy mentioned in Section IV-A. In particular, the operators simulate instances of faults within the first three categories (F1, F2 and F3). 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 [35]. We applied the operators to 12 small-sized AspectJ programs and ran the non-equivalent mutants on a functional-based test set. Table II summarises the study results. It includes: some metrics for the systems (*e.g.* the number of aspects, pointcuts and advices); 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 8–10 in the table), we can observe that changes applied to pointcuts 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, 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.

C. Covering and Analysing Test Requirements

According to the results presented in Table II, the operators were able to introduce non-trivial faults into the systems. In total, 39 mutants remained alive after the test set execution in 7 out of 12 systems. The main point here regarded the analysis of such 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[1]. This is due to the quantification and obliviousness properties [36] that are realised by AOP constructs such as pointcuts, advices and `declare`-like expressions.

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) {
```

Fig. 3. Example of an equivalent mutant of the MusicOnline application.

The code excerpts shown in Figure 3 illustrate 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) [10] for the MusicOnline system, which consists in an online music service presented by Bodkin and Laddad [37]. 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. The conclusion is that, even though a mutation can impact on the quantification of join points, the behaviour of the woven application may remain the same.

Mutations such as the one shown in Figure 3 require 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. On the other hand, as shown in Table II, many mutants were automatically set as equivalent[2]. They are all pointcut-related mutants, and the automatic detection of the equivalent ones is based on the analysis of *join point static shadows* [39]. If two pointcut capture the same set of join points, they are considered equivalent, despite the dynamic residues left in the base code during the weaving process.

V. Discussion and Final Remarks

A report recently published summarised the contributions of the Brazilian community to the "world" of AO Software Development [40]. 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.

In spite of the challenges addressed by our research (mainly challenges 1–4), a major open issue enumerated by Kulesza et al. [40] concerns the lack of experimental studies to assess the usefulness and feasibility of AO testing approaches. With respect to this, the results of the preliminary studies presented in Sections III and IV represent only initial evaluation stage. Other studies that address AO systems larger than the ones used in the preliminary evaluation are strongly necessary, though not available for the time being. For instance, for larger AO systems, we have estimated the effort to cover structural requirements derived from the pointcut-based approach based on a theoretical analysis [13]. Besides this, we have also roughly estimated the cost of mutation testing in terms of number of mutants for larger systems [35]. However, only by creating adequate test suites for such large systems one shall be able to draw 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. Overall, other research initiatives address only small-sized applications [8, 22, 11, 15, 14]. Only a few studies and approaches that may be related to testing (*e.g.* characterisation of bug patterns for exception handling [5] and AO refactoring supported by regression testing [2]) have handled larger AO systems, not only "toy" examples.

As future work, we are planning cross-comparison studies considering test selection criteria of different techniques within

[1]The inter-class mutation operators for Java [33] pose a similar challenge: mutations of inheritance and polymorphism elements also require broad analyses of the compiled application.

[2]The testing process and criterion application was supported by the *Proteum/AJ* tool [38]. More details can be found in a previous paper [35].

TABLE II
RESULTS OF EVALUATION STUDY OF MUTATION-BASED TESTING.

Application	Classes[1]	Aspects	Pointcuts	Advices	declare-like expressions	Mut. G1	Mut. G2	Mut. G3	Total	Equiv. Autom.	Equiv. Manual	Anom.	Alive	Killed by original TS	Added TCs
1. BankingSystem	9	6	11	7	1	108	2	26	136	68	–	18	50	50	–
2. Telecom	6	3	6	7	1	82	2	27	111	46	10	12	53	31	4
3. ProdLine	8	8	10	10	4	158	0	41	199	125	–	16	58	58	–
4. FactorialOptimiser	1	1	2	3	0	14	0	15	29	8	1	6	15	14	–
5. MusicOnline	7	2	4	3	0	47	0	10	57	25	2	5	27	22	2
6. VendingMachine	1	3	6	6	1	82	2	29	113	58	13	8	47	23	5
7. PointBoundsChecker	1	1	4	4	0	46	0	24	70	32	–	10	28	28	–
8. StackManager	4	3	3	3	0	34	0	11	45	24	–	0	21	21	–
9. PointShadowManager	2	1	3	3	0	38	0	12	50	25	5	4	21	13	2
10. Math	1	1	1	1	0	16	0	4	20	13	2	0	7	4	1
11. AuthSystem	3	2	2	2	0	45	0	7	52	28	1	3	21	17	2
12. SeqGen	8	4	3	3	2	33	0	7	40	19	8	3	18	4	3
TOTAL	51	35	55	52	9	703	6	213	922	471	42	85	366	285	19

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

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 former evaluations. 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 that cannot be easily quantified only in terms of the number of test cases or the number of test requirements.

Acknowledgements: The authors thank the financial support received from FAPESP and CAPES.

REFERENCES

[1] R. T. Alexander, J. M. Bieman, and A. A. Andrews, "Towards the systematic testing of aspect-oriented programs," Colorado State University, Tech Report CS-04-105, 2004.

[2] A. van Deursen, M. Marin, and L. Moonen, "A systematic aspect-oriented refactoring and testing strategy, and its application to JHotDraw," Stichting Centrum voor Wiskundeen Informatica, Tech.Report SEN-R0507, 2005.

[3] J. S. Bækken and R. T. Alexander, "A candidate fault model for aspectj pointcuts," in *ISSRE'06*, 2006, pp. 169–178.

[4] M. Ceccato, P. Tonella, and F. Ricca, "Is AOP code easier or harder to test than OOP code?" in *WTAOP'05*, 2005.

[5] R. Coelho et al., "Assessing the impact of aspects on exception flows: An exploratory study," in *ECOOP'08*. Springer, 2008, pp. 207–234.

[6] F. C. Ferrari et al., "Characterising faults in aspect-oriented programs: Towards filling the gap between theory and practice," in *SBES'10*. IEEE Computer Society, 2010, pp. 50–59.

[7] J. Zhao, "Data-flow-based unit testing of aspect-oriented programs," in *COMPSAC'03*. IEEE Computer Society, 2003, pp. 188–197.

[8] D. Xu and W. Xu, "State-based incremental testing of aspect-oriented programs," in *AOSD'06*. ACM Press, 2006, pp. 180–189.

[9] O. A. L. Lemos et al., "Control and data flow structural testing criteria for aspect-oriented programs," *Journal of Systems and Software*, vol. 80, no. 6, pp. 862–882, 2007.

[10] F. C. Ferrari, J. C. Maldonado, and A. Rashid, "Mutation testing for aspect-oriented programs," in *ICST'08*. IEEE, 2008, pp. 52–61.

[11] P. Anbalagan and T. Xie, "Automated generation of pointcut mutants for testing pointcuts in AspectJ programs," in *ISSRE'08*. IEEE Computer Society, 2008, pp. 239–248.

[12] O. A. L. Lemos, I. G. Franchin, and P. C. Masiero, "Integration testing of object-oriented and aspect-oriented programs: A structural pairwise approach for Java," *Science of Computer Programming*, vol. 74, no. 10, pp. 861–878, 2009.

[13] O. A. L. Lemos and P. C. Masiero, "A pointcut-based coverage analysis approach for aspect-oriented programs," *Information Sciences*, vol. 181, no. 13, pp. 2721–2746, 2011.

[14] F. Wedyan and S. Ghosh, "A dataflow testing approach for aspect-oriented programs," in *HASE'10*. IEEE, 2010, pp. 64–73.

[15] R. Delamare, B. Baudry, S. Ghosh, and Y. Le Traon, "A test-driven approach to developing pointcut descriptors in AspectJ," in *ICST'09*. IEEE Computer Society, 2009, pp. 376–385.

[16] B. B. P. Cafeo and P. C. Masiero, "Contextual integration testing of object-oriented and aspect-oriented programs: A structural approach for Java and AspectJ," in *SBES'11*. IEEE, 2011, pp. 214–223.

[17] F. Wedyan and S. Ghosh, "On generating mutants for aspectj programs," *Information and Software Technology*, vol. 54, no. 8, pp. 900–914, 2012.

[18] F. C. Ferrari et al., "An exploratory study of fault-proneness in evolving aspect-oriented programs," in *ICSE'10*. ACM Press, 2010, pp. 65–74.

[19] C. Zhao and R. T. Alexander, "Testing aspect-oriented programs as object-oriented programs," in *WTAOP'07*. ACM, 2007, pp. 23–27.

[20] T. Xie and J. Zhao, "Perspectives on automated testing of aspect-oriented programs," in *WTAOP'07*. ACM Press, 2007, pp. 7–12.

[21] S. Rapps and E. J. Weyuker, "Data flow analysis techniques for program test data selection," in *ICSE'82*. IEEE, 1982, pp. 272–278.

[22] M. L. Bernardi and G. A. D. Lucca, "Testing aspect oriented programs: an approach based on the coverage of the interactions among advices and methods," in *QUATIC'07*. IEEE Computer Society, 2007, pp. 65–76.

[23] V. Neves, O. A. L. Lemos, and P. C. Masiero, "Structural integration testing at level 1 of object- and aspect-oriented programs," in *LA-WASP'09*. Brazilian Computer Society, 2009, pp. 31–38, (in Portuguese).

[24] U. Linnenkugel and M. Müllerburg, "Test data selection criteria for (software) integration testing," in *ICSI'90*, 1990, pp. 709–717.

[25] M. J. Harrold and G. Rothermel, "Performing data flow testing on classes," in *FSE'94*. ACM Press, 1994, pp. 154–163.

[26] A. M. R. Vincenzi, M. E. Delamaro, J. C. Maldonado, and W. E. Wong, "Establishing structural testing criteria for java bytecode," *Software: Practice and Experience*, vol. 36, no. 14, pp. 1513–1541, 2006.

[27] L. J. Morell, "A theory of fault-based testing," *IEEE Transactions on Software Engineering*, vol. 16, no. 8, pp. 844–857, 1990.

[28] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *IEEE Computer*, vol. 11, no. 4, pp. 34–43, 1978.

[29] H. Agrawal et al., "Design of mutant operators for the C programming language," Purdue University, Tech Report SERC-TR41-P, 1989.

[30] T. A. Budd, "Mutation analysis of program test data," Ph.D. dissertation, Graduate School, Yale University, New Haven, CT - USA, 1980.

[31] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur, "Interface Mutation: An approach for integration testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 3, pp. 228–247, 2001.

[32] A. M. R. Vincenzi, "Object-oriented: Definition, implementation and analysis of validation and testing resources," Ph.D. dissertation, ICMC/USP, São Carlos, SP - Brazil, 2004, (in Portuguese).

[33] Y. S. Ma, Y. R. Kwon, and J. Offutt, "Inter-class mutation operators for Java," in *ISSRE'02*. IEEE Computer Society, 2002, pp. 352–366.

[34] Y.-S. Ma, M. J. Harrold, and Y.-R. Kwon, "Evaluation of mutation testing for object-oriented programs," in *ICSE'06*, 2006, pp. 869–872.

[35] F. C. Ferrari, A. Rashid, and J. C. Maldonado, "Towards the practical mutation testing of AspectJ programs," *Science of Computer Programming*, vol. 78, no. 9, pp. 1639–1662, 2013.

[36] R. E. Filman and D. Friedman, "Aspect-oriented programming is quantification and obliviousness," in *Aspect-Oriented Software Development*. Boston: Addison-Wesley, 2004, ch. 2, pp. 21–35.

[37] R. Bodkin and R. Laddad, "Enterprise aspect-oriented programming," in *Tutorials of EclipseCon 2005*, Burlingame/CA - USA, 2005.

[38] F. C. Ferrari, E. Y. Nakagawa, A. Rashid, and J. C. Maldonado, "Automating the mutation testing of aspect-oriented Java programs," in *AST'10*. ACM Press, 2010, pp. 51–58.

[39] E. Hilsdale and J. Hugunin, "Advice weaving in AspectJ," in *AOSD'04*. ACM Press, 2004, pp. 26–35.

[40] U. Kulesza et al., "The crosscutting impact of the AOSD Brazilian research community," *The Journal of Systems and Software*, vol. 86, no. 4, pp. 905–933, 2013.