A contribution to the fault-based testing of aspect-oriented software

Fabiano Cutigi Ferrari

A contribution to the fault-based testing of aspect-oriented software

Fabiano Cutigi Ferrari

Orientador: ***Prof. Dr. José Carlos Maldonado***

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação - ICMC-USP, como parte dos requisitos para obtenção do título de Doutor em Ciências - Ciências de Computação e Matemática Computacional.

USP – São Carlos
Outubro de 2010

Acknowledgements

Initially, I would like to thank the Instituto de Ciências Matemáticas e de Computação and its members for having supported me to conclude my PhD research. A also thank my supervisor, Professor José Carlos Maldonado, for his perspicacity during the whole course of my work, always pointing out key issues and directions that should be followed.

Thanks to Professor Awais Rashid, from the University of Lancaster, for his supervision and financial support during my visit to that university, and to Yvonne Rigby for her cordial welcome and support as well.

I specially thank Professor Alessandro Garcia for the motivation and strong collaboration that has started at Lancaster and lasts until nowadays.

Lovely thanks to Valéria, my source of inspiration, who has comprehended my absence specially during the final episodes and highest pressure times. To my parents Ivani and Antônio, and to my brother Cristiano, for their unconditional support and for having welcomed me amongst all my adventures across the world.

I am thankful for my friends and competent contributors: Otávio Lemos and Rachel Burrows (several nights awake near submission deadlines!), Marco Graciotto, Rodrigo Fraxino, André Endo, Vinicius Durelli, Eduardo Figueiredo, Nelio Cacho, Roberta Coelho, Elisa Nakagawa and Paulo Masiero.

I would also like to say thanks to my friends from LabES and ICMC, who have borne a few bad mood days along all these years ☺; some friends of very short term, others much more "insistent"; some already professionals in the market, others still in the fight. I will not write a list of names otherwise I would certainly be unfair. Therefore, thank you all!

Thanks also to my friends from Lancaster and PUC-Rio, for the warm welcome and pleasant stay during my visits. Again, thank you all!

Finally, I would like to thank the financial support received from the Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), from the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES), and from the AOSD-Europe Project.

Agradecimentos

Inicialmente, agradeço ao Instituto de Ciências Matemáticas e de Computação e seus membros por terem possibilitado a conclusão deste trabalho de doutorado. Agradeço também ao meu orientador, Prof. José Carlos Maldonado, por sua perspicácia ao longo de todo o trabalho, sempre destacando pontos-chave e apontando direções a serem seguidas.

Agradeço ao Professor Awais Rashid, da Universidade de Lancaster, pela supervisão e apoio financeiro durante a minha visita a essa universidade. E à Yvonne Rigby, pela cordial recepção em Lancaster e apoio durante minha visita.

Faço aqui um agradecimento especial ao Professor Alessandro Garcia, pela motivação e intensa colaboração iniciada em Lancaster e que se estendeu até os dias atuais.

Agradeço à Valéria, minha fonte de inspiração, que soube compreender os momentos de ausência, principalmente nas fases finais e de maior pressão do trabalho. Aos meus pais, Ivani e Antônio, e ao meu irmão, Cristiano, pelo constante apoio e por sempre me acolherem entre minhas várias aventuras pelo mundo afora.

Aos amigos e competente colaboradores: Otávio Lemos e Rachel Burrows (várias noites em claro em dias de submissão!), Marco Graciotto, Rodrigo Fraxino, André Endo, Vinicius Durelli, Eduardo Figueiredo, Nelio Cacho, Roberta Coelho, Elisa Nakagawa e Paulo Masiero.

Aos meus amigos do LabES e do ICMC, que agüentaram alguns poucos dias de mau humor durante esses anos ☺; alguns de rápida passagem, outros bem mais "insistentes"; alguns já profissionais em atividade, outros ainda na luta. Não citarei nomes, pois certamente cometeria a injustiça do esquecimento. Portanto, obrigado a todos!

Aos amigos de Lancaster e da PUC-Rio, que me receberam muito bem durante minhas visitas. Novamente, obrigado a todos!

Por fim, agradeço pelo apoio financeiro recebido da Fundação de Amparo à Pesquisa do Estado de São Paulo (FAPESP), da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior (CAPES). e do Projeto AOSD-Europe.

Declaration of Original Authorship and List of Publications

I confirm that this dissertation has not been submitted in support of an application for another degree at this or any other teaching or research institution. It is the result of my own work and the use of all material from other sources has been properly and fully acknowledged. Research done in collaboration is also clearly indicated.

Excerpts of this dissertation have been either published or submitted for the appreciation of editorial boards of journals, conferences and workshops, according to the list of publications presented below. My contributions to each publication are noted. Acceptance rates of conferences are also indicated.

Journal and Conference Papers

- ***Ferrari, F. C.***; Rashid, A.; and Maldonado, J. C.: *"Towards the Practical Mutation Testing of Aspect-Oriented Java Programs"*, (currently under evaluation).

  Journal: Science of Computer Programming.

  Level of contribution: High – the PhD candidate is the main investigator and conducted the work together with his contributors.

- ***Ferrari, F. C.***; Burrows, R.; Lemos, O. A. L.; Garcia, A.; Figueiredo, E.; Cacho, N.; Lopes, F.; Temudo, N.; Silva, L.; Soares, S.; Rashid, A.; Masiero, P.; Batista, T.; and Maldonado, J. C.: *"An Exploratory Study of Fault-Proneness in Evolving Aspect-Oriented Programs"*.

  Event: 32^{nd} International Conference on Software Engineering (ICSE'10).

  Acceptance rate: 14%

  Level of contribution: High – the PhD candidate is the main investigator and led a large team of contributors during all phases of the work.

- ***Ferrari, F. C.***; Burrows, R.; Lemos, O. A. L.; Garcia, A.; and Maldonado, J. C.: *"Characterising Faults in Aspect-Oriented Programs: Towards Filling the Gap between Theory and Practice"*.

  Event: 24^{th} Brazilian Symposium on Software Engineering (SBES'10).

  Acceptance rate: 22%

  Level of contribution: High – the PhD candidate is the main investigator and led the data collection and analysis as well as the paper writing.

- ***Ferrari, F. C.***; Maldonado, J. C.; and Rashid, A.: *"Mutation Testing for Aspect-Oriented Programs"*.

  Event: 1^{st} International Conference on Software Testing, Verification and Validation (ICST'08).

  Acceptance rate: 25%

  Level of contribution: High – the PhD candidate is the main investigator and conducted the work together with his contributors.

- Coelho, R.; Rashid, A.; Garcia, A.; ***Ferrari, F. C.***; Cacho, N.; Kulesza, U.; von Staa, A.; and Lucena, C.: *"Assessing the Impact of Aspects on Exception Flows: An Exploratory Study"*.

  Event: 22^{nd} European Conference on Object-Oriented Programming (ECOOP'08).

  Acceptance rate: 20%

  Level of contribution: Medium – the PhD candidate helped in the data collection and analysis as well as in the paper writing.

- Nakagawa, E. Y.; Simão, A. S.; ***Ferrari, F. C.***; and Maldonado, J. C.: *"Towards a Reference Architecture for Software Testing Tools"*.

  Event: 19^{th} International Conference on Software Engineering and Knowledge Engineering (SEKE'07).

  Acceptance rate: 43%

  Level of contribution: Low – the PhD candidate helped in the definition of the core architecture concepts and in the elaboration of parts of the text.

Workshop Papers

- ***Ferrari, F. C.***; Nakagawa, E. Y.; Rashid, A.; and Maldonado, J. C.: *"Automating the Mutation Testing of Aspect-Oriented Java Programs"*.

  Event: 5^{th} ICSE International Workshop Automation of Software Test (AST'10).

Level of contribution: High – the PhD candidate developed the tool with support from his contributors, specially regarding the definition of requirements and architecture. He also led the paper writing.

- ***Ferrari, F. C.***; Höhn, E. N.; and Maldonado, J. C.: *"Testing Aspect-Oriented Software: Evolution and Collaboration through the Years"*.

  Event: 3^{rd} Latin American Workshop on Aspect-Oriented Software Development (LAWASP'09) – held in conjunction with SBES'09.

  Level of contribution: High – the PhD candidate is the main investigator and led the work and the paper writing.

- ***Ferrari, F. C.***; and Maldonado, J. C.: *"Experimenting with a Multi-Iteration Systematic Review in Software Engineering"*.

  Event: 5^{th} Experimental Software Engineering Latin America Workshop (ESELAW'08).

  Level of contribution: High – the PhD candidate is the main investigator and conducted the work together with his adviser.

- Lemos, O. A. L.; ***Ferrari, F. C.***; Masiero, P. C.; and Lopes, C. V.: *"Testing Aspect-Oriented Programming Pointcut Descriptors"*.

  Event: 2^{nd} Workshop on Testing Aspect Oriented Programs (WTAOP'06) - held in conjunction with ISSTA'06.

  Level of contribution: Medium – the PhD candidate helped in the approach definition (in particular, the mutation-based testing phase) as well as in the paper writing.

- ***Ferrari, F. C.***; and Maldonado, J. C.: *"A Systematic Review on Aspect-Oriented Software Testing"* (in Portuguese).

  Event: 3^{rd} Brazilian Workshop on Aspect-Oriented Software Development (WASP'06) - held in conjunction with SBES'06.

  Level of contribution: High – the PhD candidate is the main investigator and conducted the work together with his adviser.

Book Chapter

- Masiero, P. C.; Lemos, O. A. L.; ***Ferrari, F. C.***; and Maldonado, J. C.: *"Testing Object and Aspect-Oriented Programs: Theory and Practice"* (in Portuguese).

  Book: Atualizações em Informática. Breitman, K. and Anido, R., eds. Editora PUC-Rio, Rio de Janeiro, Brazil, 2006.

Level of contribution: High – the PhD candidate helped in the text writing and structuring, as well as in identifying sound examples to demonstrate the application of the selected testing approaches.

Technical Report

- ***Ferrari, F. C.***; and Maldonado, J. C.: *"Aspect-Oriented Software Testing: A Systematic Review"* (in Portuguese).

  Institution: ICMC/USP, 2007.

  Level of contribution: High – the PhD candidate is the main investigator and conducted the work together with his adviser.

Other Related Publications

- Burrows, R.; ***Ferrari, F. C.***; Lemos, O. A. L.; Garcia, A.; and Taïani, F.: *"The Impact of Coupling on the Fault-Proneness of Aspect-Oriented Programs: An Empirical Study"*.

  Event: 21^{st} International Symposium on Software Reliability Engineering (ISSRE'10).

  Acceptance rate: 32%

  Level of contribution: Medium – the PhD candidate helped in the data collection and analysis as well as in the paper writing.

- Burrows, R.; ***Ferrari, F. C.***; Garcia, A.; and Taïani, F.: *"An Empirical Evaluation of Coupling Metrics on Aspect-Oriented Programs"*.

  Event: ICSE Workshop on Emerging Trends in Software Metrics (WETSoM'10).

  Level of contribution: Medium – the PhD candidate helped in the data collection and analysis as well as in the paper writing.

- Coelho, R.; Lemos, O. A. L.; ***Ferrari, F. C.***; Masiero, P. C.; and von Staa, A.: *"On the Robustness Assessment of Aspect-Oriented Programs"*.

  Event: 3^{rd} Workshop on Assessment of Contemporary Modularization Techniques (ACoM) - held in conjunction with OOPSLA'09.

  Level of contribution: Medium – the PhD candidate helped in the definition of a testing approach as well as in the paper writing.

- Figueiredo, E.; Cacho, N.; SantAnna, C.; Monteiro, M.; Kulesza, U.; Garcia, A.; Soares, S.; ***Ferrari, F. C.***; Khan, S.; Castor Filho, F.; and Dantas, F.: *"Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability"*.

  Event: 30^{th} International Conference on Software Engineering (ICSE'08).

  Acceptance rate: 15%

  Level of contribution: Low – the PhD candidate helped in the data collection and in the elaboration of parts of the text.

Abstract

Aspect-Oriented Programming (AOP) is a contemporary software development technique that strongly relies on the Separation of Concerns principle. It aims to tackle software modularisation problems by introducing the aspect as a new implementation unit to encapsulate behaviour required to realise the so-called crosscutting concerns. Despite the benefits that may be achieved with AOP, its implementation mechanisms represent new potential sources of faults that should be handled during the testing phase. In this context, mutation testing is a widely investigated fault-based test selection criterion that can help to demonstrate the absence of prespecified faults in the software. It is believed to be an adequate tool to deal with testing-related specificities of contemporary programming techniques such as AOP. However, to date, the few initiatives for customising the mutation testing for aspect-oriented (AO) programs show either limited coverage with respect to the types of simulated faults, or a need for both adequate tool support and proper evaluation. This thesis tackles these limitations by defining a comprehensive mutation-based testing approach for AO programs written in the AspectJ language. It starts with a fault-proneness investigation in order to define a fault taxonomy for AO software. Such taxonomy encompasses a range of fault types and underlay the definition of a set of mutation operators for AO programs. Automated tool support is also provided. A series of quantitative studies show that the proposed fault taxonomy is able to categorise faults identified from several available AO systems. Moreover, the studies show that the mutation operators are able to simulate faults that may not be revealed by pre-existing, non-mutation-based test suites. Furthermore, the effort required to augment the test suites to provide adequate coverage of mutants does not tend to overwhelm the testers. This provides evidence of the feasibility of the proposed approach and represents a step towards the practical fault-based testing of AO programs.

Keywords: Software testing, Aspect-Oriented Programming, mutation testing, fault taxonomy, fault characterisation.

Resumo

A Programação Orientada a Aspectos (POA) é uma técnica contemporânea de desenvolvimento de software fortemente baseada no princípio da separação de interesses. Ela tem como objetivo tratar de problemas de modularização de software por meio da introdução do aspecto como uma nova unidade de implementação que encapsula comportamento relacionado aos interesses transversais do software. A despeito dos benefícios que podem ser alcançados com o uso da POA, seus mecanismos de implementação representam novas potenciais fontes de defeitos que devem ser tratados durante a fase de teste de software. Nesse contexto, o teste de mutação consiste em um critério de seleção de testes baseado em defeitos que tem sido bastante investigado para demonstrar a ausência de defeitos pré-especificados no software. Acredita-se que o teste de mutação seja uma ferramenta adequada para lidar com as particularidades de técnicas de programação contemporâneas como a POA. Entretanto, até o presente momento, as poucas iniciativas para adaptar o teste de mutação para o contexto de programas orientados a aspectos (OA) apresentam cobertura limitada em relação aos tipos de defeitos simulados, ou ainda requerem adequado apoio automatizado e avaliações. Esta tese visa a mitigar essas limitações por meio da definição de uma abordagem abrangente de teste de mutação para programas OA escritos na linguagem AspectJ. A tese inicia como uma investigação da propensão a defeitos de programas OA e define uma taxonomia de defeitos para tais programas. A taxonomia inclui uma variedade de tipos de defeitos e serviu como base para a definição de um conjunto de operadores de mutação para programas OA. Suporte automatizado para a aplicação dos operadores também foi disponibilizado. Uma série de estudos quantitativos mostra que a taxonomia de defeitos proposta é suficiente para classificar defeitos encontrados em vários sistemas OA. Os estudos também mostram que os operadores de mutação propostos são capazes de simular defeitos que podem não ser relevados por conjuntos de teste pré-existentes, não derivados para cobrir mutantes. Além disso, observou-se que o esforço requerido para evoluir tais conjuntos de teste de forma a torná-los adequados para os requisitos gerados pelos operadores

de mutação não tendem a sobrecarregar os testadores envolvidos. Dessa forma, geraram-se evidências de que o teste de mutação é factível para programas OA, representando um passo no sentido da sua aplicação prática nesse contexto.

Palavras-chave: Teste de software, Programação Orientada a Aspectos, teste de mutação, taxonomia de defeitos, caracterização de defeitos.

Table of Contents

1 Introduction **1**
- 1.1 Problem Statement and Justification for the Research . . . 3
- 1.2 Objectives and Research Methodology . . . 4
- 1.3 Thesis Outline and Summary of Contributions . . . 5

2 Background **9**
- 2.1 Foundations of Aspect-Oriented Programming . . . 9
  - 2.1.1 Programming Languages and other Supporting Technologies . . . 14
- 2.2 Foundations of Software Testing . . . 20
  - 2.2.1 Basic Terminology . . . 21
  - 2.2.2 Testing Techniques and Criteria . . . 23
  - 2.2.3 Test Evaluation and Comparison amongst Criteria . . . 30
  - 2.2.4 Test Automation . . . 32
- 2.3 Testing of Aspect-Oriented Software . . . 33
  - 2.3.1 The Systematic Mapping Study Protocol and Process . . . 33
  - 2.3.2 The Systematic Mapping Study Results . . . 35
  - 2.3.3 Fault Taxonomies for AO Software . . . 36
  - 2.3.4 AO Testing Approaches . . . 40
  - 2.3.5 Tool Support for AO Testing . . . 49
- 2.4 Final Remarks . . . 56

3 Evaluating the Fault-Proneness of Aspect-Oriented Programs **57**
- 3.1 A Study of the Fault-Proneness of AO Programs . . . 58
  - 3.1.1 Goals and Method . . . 59
  - 3.1.2 Results . . . 60
- 3.2 Defining and Evaluating a Fault Taxonomy for AO Programs . . . 63
  - 3.2.1 Goals and Method . . . 64
  - 3.2.2 Results . . . 64
- 3.3 Summary of Contributions and Limitations . . . 68

4 Designing Mutation Operators for Aspect-Oriented Programs 71
4.1 Mutation Operators for AspectJ Programs . . . 72
4.1.1 Mutation Operators versus Fault Types . . . 72
4.1.2 Preliminary Cost Analysis . . . 74
4.2 Generalisation of the Fault Taxonomy . . . 75
4.3 Summary of Contributions and Limitations . . . 76

5 Automating the Mutation Testing of Aspect-Oriented Programs 79
5.1 Requirements for Mutation Tools . . . 80
5.2 The Architecture of Proteum/AJ . . . 81
5.3 The Main Functionalities of Proteum/AJ . . . 83
5.4 Implementation Details . . . 84
5.4.1 Core Modules . . . 85
5.5 Summary of Contributions and Limitations . . . 87

6 Evaluating the Proposed Mutation Testing Approach 89
6.1 First Study: Evaluating the Usefulness and Required Effort . . . 90
6.1.1 Target Applications . . . 91
6.1.2 Building the Initial Test Sets . . . 93
6.1.3 Applying Mutant Analysis to the Target Applications . . . 95
6.1.4 Analysis of the Results . . . 97
6.1.5 Additional Comments on the Mutant Analysis Step . . . 99
6.2 Second Study: Estimating the Cost of the Approach with Larger Systems . 100
6.2.1 Target Systems . . . 100
6.2.2 Generating Mutants for the Target Systems . . . 101
6.2.3 Contrasting the Results with the First Study . . . 102
6.3 Study Limitations . . . 104
6.4 Final Remarks . . . 105

7 Conclusions 107
7.1 Revisiting the Thesis Contributions . . . 108
7.1.1 Theoretical Definitions . . . 108
7.1.2 Implementation of Automated Support . . . 109
7.1.3 Evaluation Studies . . . 109
7.2 Limitations and Future Work . . . 110
7.2.1 Possible Research Directions . . . 111

References 113

A Paper: An Exploratory Study of Fault-Proneness in Evolving Aspect-Oriented Programs 135

B Paper: Characterising Faults in Aspect-Oriented Programs: Towards Filling the Gap between Theory and Practice 147

C Paper: Mutation Testing for Aspect-Oriented Programs 159

D Paper: Automating the Mutation Testing of Aspect-Oriented Java Programs **171**

E Paper: Towards the Practical Mutation Testing of Aspect-Oriented Java Programs **181**

List of Figures

2.1 Example of crosscutting concern . . . 11
2.2 Basic elements of an AOP supporting system . . . 12
2.3 A simple graphical editor written in AspectJ . . . 17
2.4 Example of context exposure in AspectJ. . . . 19
2.5 Example of static crosscutting in AspectJ . . . 20
2.6 Example of a DUG . . . 26
2.7 Subsume relation amongst structural-based test selection criteria . . . 31
2.8 The process for systematic literature review updates . . . 35
2.9 Lemos et al.'s PCD-related fault types . . . 39
2.10 Example of an AODU graph . . . 42
2.11 Examples of PWDU and PCDU graphs . . . 44
2.12 Example of an ASM before and after weaving the aspects into the base application . . . 46
2.13 Example of mutants generated by Anbalagan and Xie's framework . . . 48
2.14 The test generation process implemented in the *Aspectra* framework . . . 50
2.15 The test process supported by *JaBUTi/AJ* . . . 51
2.16 An AODU graph generated by *JaBUTi/AJ* . . . 52
2.17 A *JaBUTi/AJ* coverage report screen . . . 53
2.18 A *JaBUTi/AJ* pairwise-based test requirement selection screen . . . 53
2.19 The test process supported by *AjMutator* . . . 54

3.1 Fault distribution according to the fault taxonomy for AO software . . . 67
3.2 Example of fault related to arbitrary advice execution order . . . 68

5.1 Proteum/AJ architecture . . . 82
5.2 Proteum/AJ execution flow . . . 83
5.3 Proteum/AJ core modules . . . 86

6.1 Source code of the `debit` method (`AccountSimpleImpl` class). . . . 94
6.2 Coverage yielded by SFT test sets. . . . 98
6.3 Effort required to derive the T_M test sets. . . . 99
6.4 Example of an equivalent mutant of the MusicOnline application. . . . 100

6.5 Percentages of equivalent and anomalous mutants for the target systems. . 104

List of Tables

2.1 Advice types in AspectJ . . . 18
2.2 Main constructs for static crosscutting in AspectJ . . . 19
2.3 Advice precedence rules in AspectJ . . . 20
2.4 Systematic Mapping Study of AO testing: final selection. . . . 37
2.5 Systematic Mapping Study of AO testing: overlapping results. . . . 38
2.6 Summary of features present in tools for AO testing. . . . 55

3.1 Fault distribution per system . . . 61
3.2 Faults associated with obliviousness . . . 61
3.3 Correlation between AOP mechanism usage and faults per concern/release 63
3.4 Faults related to PCDs . . . 65
3.5 Faults related to ITDs and declare-like expressions . . . 65
3.6 Faults related to advices . . . 66
3.7 Faults related to the base program . . . 66

4.1 Mutation operators for PCDs . . . 73
4.2 Mutation operators for ITDs and declare-like expressions . . . 73
4.3 Mutation operators advices . . . 73
4.4 Relationship between mutation operators and fault types for AO software . 74
4.5 TSD and HW mutants . . . 75
4.6 Mutant average per element type . . . 75
4.7 Relationship between AOP technologies and fault types . . . 76

5.1 Requirements fulfilled by tools for mutation testing of AO programs . . . 80
5.2 Proteum/AJ implementation effort . . . 85

6.1 Values of metrics for the selected applications (first study). . . . 91
6.2 Equivalence classes and boundary values for the `debit` method. . . . 94
6.3 Functional-based test requirements and respective adequate test sets. . . . 95
6.4 Mutants generated for the 12 target applications. . . . 96
6.5 Mutation testing results for the 12 target applications. . . . 97
6.6 Values of metrics for the selected applications (second study). . . . 101

6.7 Percentages of equivalent mutants generated for the target systems. 102
6.8 Percentages of anomalous mutants generated for the target systems. 103

Abbreviations and Acronyms

AO	-	Aspect-Oriented
AODU	-	Aspect-Oriented Def-Use
AOP	-	Aspect-Oriented Programming
AOSD	-	Aspect-Oriented Software Development
ASM	-	Aspectual State Model
CFG	-	Control Flow Graph
DUG	-	Def-Use Graph
FREE	-	Flattened Regular Expression
FSM	-	Finite State Machine
EJP	-	Explicit Join Point
ITD	-	Intertype declaration
JP	-	Join point
JVM	-	Java Virtual Machine
OO	-	Object-oriented
PCD	-	Pointcut descriptor
PCDU	-	PointCut-based Def-Use
PWDU	-	PairWise Def-Use
SFT	-	Systematic Functional Testing
SLR	-	Systematic Literature Review
SMS	-	Systematic Mapping Study
SoC	-	Separation of concerns
SQA	-	Software Quality Assurance
V&V	-	Verification and Validation
XPI	-	Crosscut Programming Interface

Chapter

1

Introduction

Producing high quality software has become a pursuit of the industry since software started playing a central role in the last decades, irrespective of the application domain. To obtain high quality products, Software Quality Assurance (SQA) activities are undertaken along the development process (Sommerville, 2007, p. 642-644). Under the SQA umbrella, Verification and Validation (V&V) activities are carried out to assure the software behaves as it was specified to. In this context, software testing – which consists in one of the V&V activities – is fundamental in order to reveal faults that, despite the adoption of rigorous development practices, still remain in the products.

When we consider the advances in the Software Engineering field, we can notice that a possible mean of improving software quality regards the application of contemporary development approaches that are intended to cope with the increasing software complexity. For example, the object-oriented (OO) paradigm benefits from the object-based problem decomposition (Booch, 1994, p. 16-20) in order to handle complex software design and implementation. It is expected to enhance external software quality attributes (e.g. evolvability and extensibility) and, in turn, to make software more reusable. In general, reusing a piece of software that has already been tested within its original context will consequently result in less faults being introduced in a newly derived software, as empirically demonstrated in previous research (Mohagheghi et al., 2004).

In spite of the benefits that can be achieved with the adoption of the OO paradigm, its typical implementation units – i.e. the classes – may not properly modularise some stakeholders' concerns and non-functional requirements that appear either interwoven with other concerns or requirements, or scattered over several modules in the software (Elrad et al., 2001a). Typical examples of such concerns[1] – the so-called *crosscutting concerns* (Kiczales et al., 1997) – are distribution, caching, concurrency, business rules and certain design patterns.

Aspect-Oriented Programming (AOP) is a contemporary development technique that aims to tackle this software modularisation issue (Kiczales et al., 1997). It introduced the *aspect* as a new conceptual implementation unit that ideally encapsulates all behaviour required to realise a given crosscutting concern. Once the aspects are implemented, they are combined with the other system modules – the *base modules* or *base code* – in a process called *aspect weaving* (Kiczales et al., 1997). This combination process produces the executable system that fully realises the expected behaviour. The benefits that are likely to be observed in aspect-oriented (AO) programs range from improved modularity itself to enhanced maintainability (Mortensen et al., 2010), higher reusability (Laddad, 2003a) and facilitated software evolution (Coady and Kiczales, 2003).

As of the proposition of innovative software development techniques, however, new challenges with respect to software testing are also introduced. As a consequence, researchers and practitioners continuously endeavour to figure out means of reducing the number of faults in the resulting products. For example, underlying OO concepts such as information hiding, class hierarchies and polymorphism pose new challenges for the testing activity (Binder, 1999, p. 69-97). Hence, specific approaches have been proposed to support the testing of OO program accordingly (Chen et al., 1998; Harrold and Rothermel, 1994; Ma et al., 2002; Turner and Robson, 1993; Vincenzi et al., 2006a).

When AOP is concerned, its underlying concepts such as quantification and obliviousness (Filman and Friedman, 2004) as well as its basic programming constructs like pointcuts and advices (Kiczales et al., 2001b) also represent new potential sources of software faults (Alexander et al., 2004). Furthermore, the combination of aspects and base code yields new subtle interactions amongst the software modules, which can lead to a higher number of faults in the resulting programs (Burrows et al., 2010b).

Similarly to OO programming, once again it is necessary to investigate the existing testing approaches and spot the required adaptations and innovations to provide adequate

[1]Without loss of generality, we henceforth use the term *concern* to refer to any stakeholders' concern and functional/non-functional requirement that can impact on the design and maintenance of program modules (Robillard and Murphy, 2007).

support for the testing of AO software. This thesis addresses this problem, evaluating the fault-proneness of AO programs and proposing a fault-based testing approach for this kind of programs. The remaining of this introductory chapter characterises the investigated problem (Section 1.1), defines the research objectives and methodology (Section 1.2), and summarises the achieved contributions and the document organisation (Section 1.3).

1.1 Problem Statement and Justification for the Research

Software faults have been widely studied by researchers over the years (Basili and Perricone, 1984; Endress, 1978; Offutt et al., 2001; Ostrand and Weyuker, 1984; Voas et al., 1991). Both theoretical and empirical characterisation of faults are important because they are the basis for the definition of *fault taxonomies* that can be used, for instance, for the evaluation of existing testing approaches with respect to their ability in revealing the faults described in the taxonomy. Furthermore, they provide guidance for the definition of testing approaches that can support fault detection within specific software application domains.

In this context, the Mutant Analysis (DeMillo et al., 1978), also known as *mutation testing*, has been largely explored as a fault-based test selection criterion that relies on recurring mistakes made by programmers during the software development. It develops upon well-characterised fault taxonomies and simulates faults by means of mutation operators (DeMillo et al., 1978). Mutation testing enables the evaluation of the software itself (when faults are revealed during the test process) (Mathur, 2007, p. 533-536) as well as the test data (whether it is sensitive enough to reveal the simulated faults) (DeMillo et al., 1978; Mathur, 2007, p. 512). Furthermore, mutation testing has been shown to be an important tool for the assessment of other testing approaches within the Experimental Software Engineering context (Andrews et al., 2005; Do and Rothermel, 2006).

When it comes to AOP, the results of a systematic mapping study[2] (Ferrari et al., 2009) reveal that considerable research effort has been spent in AOP-specific fault characterisation (Alexander et al., 2004; Bækken, 2006; Ceccato et al., 2005; Coelho et al., 2008a; Eaddy et al., 2007; Ferrari et al., 2008; Lemos et al., 2006; van Deursen et al., 2005; Zhang and Zhao, 2007) and some testing approaches for AO software based on mutation testing (Anbalagan and Xie, 2008; Delamare et al., 2009a; Lemos et al., 2006; Mortensen and Alexander, 2005). However, existing research on fault characterisation is either mostly based on programming language features and researchers' expertise or limited to specific

[2]More details can be found in Chapter 2, Section 2.3.

programming mechanisms. Moreover, it still lacks evaluation with respect to their ability in classifying faults uncovered in real software development scenarios.

The current mutation-based testing approaches for AO software suffer from similar limitations as does research on fault taxonomies. In general, the approaches do not fully cover the set of main AOP mechanisms neither support an ordinary testing process that goes from the derivation of test requirements up to the evaluation of the achieved results after the test execution. Besides that, such approaches have not been properly evaluated regarding, for instance, application costs, effectiveness and provided tool support.

To conclude, it is noticeable that more research is needed towards the practical fault-based testing of AO programs and, consequently, to help it evolve from the state of the art to the state of the practice. Developing well-founded and practical testing approaches and demonstrating their effectiveness are essential to promote the technology transfer to the industry, as emphasised by Harrold around one decade ago (Harrold, 2000).

1.2 Objectives and Research Methodology

According to the research gap characterised in the previous section, the general research question to be investigated in this thesis is that whether or not one can apply fault-based testing to AO software in an effective way and at practicable costs. Based on this general research question, we defined the following main objectives:

- *Definition of a comprehensive fault taxonomy for AO software*: we aim to investigate the fault-proneness of Aspect-Oriented Programming with respect to its underlying properties, harmful implementation mechanisms and recurring faulty scenarios. Based on our observations as well as on characteristics of the available AOP supporting technologies, we aim to define a comprehensive fault taxonomy which shall be able to categorise the variety of faults that can occur in AO software.

- *Definition of a fault-based testing approach for AO programs*: mutation testing has been empirically shown to be one of the strongest test selection criteria when compared to other widely investigated approaches like control flow- and data flow-based testing (Frankl et al., 1997; Mathur and Wong, 1994; Mathur, 2007, p. 503; Li et al., 2009). Therefore, we aim to explore the mutation testing to devise a fault-based testing approach for AO software. The fault taxonomy for AO software is going to guide the design of mutation operators for program written in AspectJ (Kiczales et al., 2001b), which is the most representative AOP supporting language currently available. We highlight that mutation operators are strongly dependent on the software

technology they are designed for (e.g. specification and implementation languages), fact that has motivated us to select AspectJ as our target language.

- *Automation of the proposed testing approach*: software testing strongly relies on automated support to enable its systematic application in real software projects (Harrold, 2000). Without this, testing may be costly, error-prone and limited to small programs (Vincenzi et al., 2006a). Therefore, we aim at automating the proposed mutation operators, hence providing a tool that supports a mutation-based testing approach for AO programs. To implement such a tool, we can leverage previous knowledge of our research group at University of São Paulo, Brazil, on developing a family of tools that support specification and program testing based on mutation (Maldonado et al., 2000).

- *Evaluation of the proposed fault taxonomy and testing approach*: we intend to evaluate the proposed fault taxonomy and the mutation operators for AO software based on data gathered from varied sources. These goals can be accomplished by documenting and categorising faults from representative AO applications with respect to the range of employed AOP mechanisms. Furthermore, these applications shall allow us to evaluate the proposed mutation operators with respect to their usefulness, application cost and effort required to cover the derived test requirements.

The next section summarises the contributions of this thesis according to the aforementioned objectives. It also presents the organisation of this doctoral dissertation according to its chapter structuring.

1.3 Thesis Outline and Summary of Contributions

The contributions of this thesis are organised as a collection of papers either published or under current evaluation. Chapter 2 is an exception to this, given that it brings the necessary background information and characterises the state of the art with respect to testing of AO software. Each of the remaining chapters presents an overview of the objectives and contributions of the respective paper. The full contents of the papers are reproduced in Appendices A–E. We call the readers' attention to the fact that there are occurrences of duplicated text excerpts specially in regard to the introductory and background sections of the papers. However, the main benefit is that the contributions are reproduced exactly as they have been published or submitted for evaluation. Therefore, we organised this dissertation as following described.

Chapter 2 brings an overview of the background theory and concepts required for the comprehension of the research presented herein. The chapter describes the fundamentals of AOP and the basic concepts of software testing, including the commonly used terminology. The chapter also summarises the results of a systematic mapping study that characterises the state of the art with respect to research on testing of AO software. This study has been updated several times over the last few years (Ferrari et al., 2009; Ferrari and Maldonado, 2006, 2007) and the results presented in Chapter 2 include the most recent developments in the field.

Chapter 3 describes the results of two studies that investigate the fault-proneness and fault characterisation of AO programs. The first study was published in the Proceedings of the 32^{nd} International Conference on Software Engineering (Ferrari et al., 2010a) and its full contents are presented in Appendix A. It analyses (i) the impact of the obliviousness property (Filman and Friedman, 2004) on the fault-proneness of evolving AO programs; and (ii) the differences amongst the fault-proneness of the main AOP mechanisms in the context of such programs. The key results show that obliviousness did impact the correctness of the evaluated AO systems, therefore corroborating with recent trends on AOP approaches that reduce obliviousness in favour of higher program comprehension (Griswold et al., 2006; Hoffman and Eugster, 2007; Kiczales and Mezini, 2005). Moreover, we found out that the main AOP mechanisms present similar fault-proneness when we consider both the overall system and concern-specific implementations. These results motivate the investigation of testing approaches that focus on other AOP mechanisms beyond pointcut expressions.

The second study described in Chapter 3 was published in the Proceedings of the 24^{th} Brazilian Symposium on Software Engineering (Ferrari et al., 2010b) and is reproduced in Appendix B. It extends our previous research on fault-proneness of AO programs (Ferrari et al., 2010a) in three different ways: (i) it describes a fault taxonomy for AO software which was initially introduced in our previous work (Ferrari et al., 2008); (ii) it evaluates the fault taxonomy through the categorisation of the fault set analysed in our previous paper (Ferrari et al., 2010a); and (iii) it characterises the most recurring faulty implementation scenarios observed in the analysed systems. The results confirm the ability of the fault taxonomy in categorising faults identified in the analysed systems. Moreover, they provide hints on harmful implementation scenarios that should be either avoided or verified by developers during the development phase.

Chapter 4 defines a set of mutation operators that underlie the fault-based testing of AspectJ programs. The operators address a variety of AOP mechanisms by simulating a range of fault types included in the fault taxonomy for AO software. Chapter 4 also

evaluates the generalisation of the fault taxonomy to other AOP supporting technologies beyond AspectJ. We noticed that most of the fault types can occur in programs developed with the investigated technologies, thus enabling the reuse of knowledge embodied into our mutation-based testing approach. The paper that includes the results described in Chapter 4 was published in the Proceedings of the 1^{st} International Conference on Software Testing, Verification and Validation (Ferrari et al., 2008) and its full contents are presented in Appendix C.

A tool named *Proteum/AJ* that automates the mutation testing of AspectJ programs is described in Chapter 5. *Proteum/AJ* supports the required steps of mutation testing, namely program execution, mutant generation, mutant execution and mutant analysis (DeMillo et al., 1978). It is also able to automatically identify equivalent mutants produced by a subset of the mutation operators. The paper that describes the tool was published in the Proceedings of the 5^{th} International Workshop on Automation of Software Test (Ferrari et al., 2010c). The paper can be found in Appendix D.

Chapter 6 presents the results of two studies that involve the application of the mutation testing approach proposed along this doctoral work. The studies aim to demonstrate the feasibility of applying mutation testing to AO systems of varied sizes and complexity. The results show that the mutation operators can be applied to AO programs at a feasible cost, not requiring large effort with respect to the number of additional test cases in order to build adequate test suites to cover the produced mutants. This chapter partially reproduces a paper submitted to the Science of Computer Programming journal. More specifically, it reproduces Sections 7, 8 and 9 of the referred paper, which describe the evaluation studies. The full paper can be found in Appendix E.

Finally, Chapter 7 concludes this dissertation. It revisits the investigated problem and summarises the achieved contributions, limitations and future work.

Chapter

2

Background

This chapter brings a literature review of the subjects that underlie the research developed in this thesis. It starts describing the core concepts of Aspect-Oriented Programming in Section 2.1. In the sequence, Section 2.2 introduces the basic terminology of software testing along with an overview of the main testing techniques and criteria for test case selection. Section 2.3 characterises the state of the art with respect to testing of aspect-oriented software according to the results of a systematic survey of the topic. Section 2.4 concludes this chapter identifying the research opportunities and outlining the contributions that are described in the subsequent chapters and appendices.

2.1 Foundations of Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) (Kiczales et al., 1997) arose in the late 90's as a possible solution to problems regarding software modularisation. Its was mainly motivated by the fact that traditional software development approaches, like procedural and object-oriented programming, could not satisfactorily cope with concerns that are spread across or tangled with other concerns in the software. Classical examples of such concerns, the so-called *crosscutting concerns* (Kiczales et al., 1997), are logging, exception handling, concurrency and certain design patterns.

AOP is strongly based on the idea of separation of concerns (SoC), which claims that computer systems are better developed if their several concerns are specified and implemented separately. This idea was already supported by Dijkstra (1976, p. 211-212) in the 70's, who stated that tasks are better accomplished when they are handled independently. These tasks can be mapped to software concerns in general, and can address both functional requirements (e.g. business rules) and non-functional requirements (e.g. synchronisation or transaction management) (Baniassad and Clarke, 2004).

Despite the initial emphasis on the implementation phase, AOP concepts have been lifted up to higher levels of software abstraction, originating the Aspect-Oriented Software Development (AOSD) approach (Filman et al., 2004). Aspects have started being considered in varied phases of the software life cycle, from requirements engineering (Araújo and Moreira, 2003; Araújo et al., 2002; Chitchyan et al., 2007; Rashid et al., 2002) to analysis and design (Baniassad and Clarke, 2004; Chavez, 2004).

Amongst the main concepts related to AOP, we can highlight the idea of *base code* and *crosscutting concerns* (Kiczales et al., 1997), the *weaving* process (Kiczales et al., 1997), *obliviousness* and *quantification* (Filman and Friedman, 2004), and *aspects* themselves together with their internal elements (Kiczales et al., 2001b, 1997). Each of them is described in the sequence.

Base Code and Crosscutting Concerns

Robillard and Murphy (2007) define a ***concern*** as *"anything a stakeholder may want to consider as a conceptual unit, including features, non-functional requirements, and design idioms"*. In this way, a concern may be any consideration that can impact the implementation of a program, ranging from a single variable within a program to coarse-grained features like business rules or data persistence. In the context of AOP, a concern is typically handled as a coarse-grained feature that can be modularised within well-defined implementation units. In this way, concerns are split into two categories: non-crosscutting concerns and crosscutting concerns. The non-crosscutting concerns compose the ***base code*** of an application and comprise the set of functionalities that can be modularised within conventional implementation units (e.g. classes and data structures). *Base modules*, *base classes* and *core functionalities* are used as synonyms of *base code*.

Crosscutting concerns, on the other hand, are concerns that cannot be properly modularised within conventional units (Kiczales et al., 1997). As a consequence, code that realises a crosscutting concern usually appears scattered over several modules in a system. Moreover, such code tends to be tangled with other concern implementations, thus hardening software evolution and maintenance.

Figure 2.1 shows an example of a crosscutting concern. The class diagram on the left-hand side represents an OO solution for a simple editor for graphical elements, which enables the user to draw points and lines on a computer display. The `Point` and `Line` classes implement methods for retrieving point coordinates (e.g. `getX` and `getP1`) as well for updating them (e.g. `setY` and `moveBy`). Every time an updating-related method is invoked, the system needs to update the objects shown on the display. We can observe that the *Display updating* concern *crosscuts* other elements in the system, i.e. the `Point` and `Line` classes. Using AOP, we can have a solution as depicted in the right-hand side of Figure 2.1; instead of inserting code related to the *Display updating* concern into `Point` and `Line`, such code can be encapsulated into an independent unit – the `DisplayUpdating` aspect – so that it will be activated by the aspect when required, i.e. when the display needs to be updated[1].

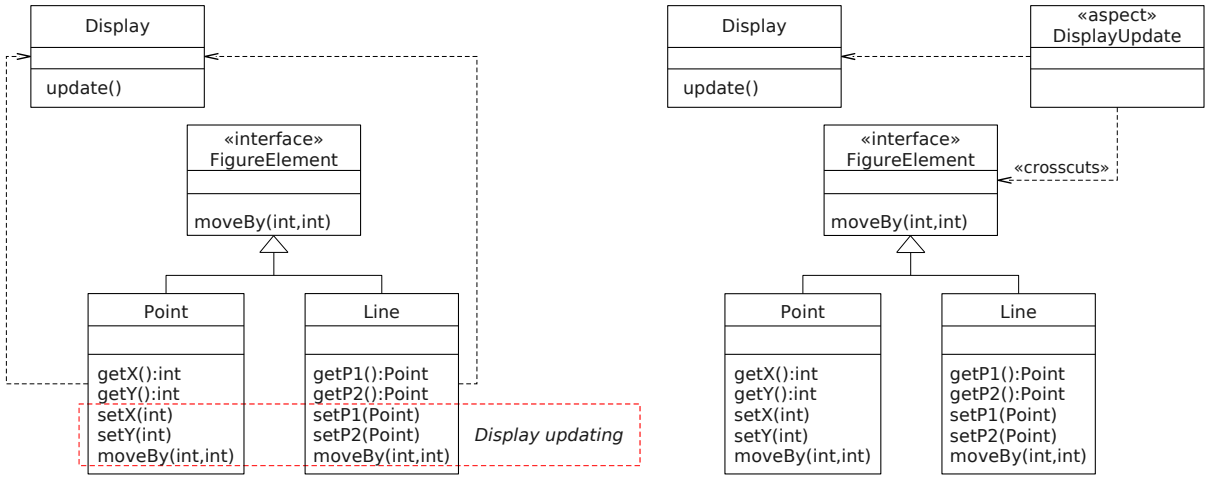


Figure 2.1: Example of crosscutting concern – partially adapted from Kiczales et al. (2001a).

Aspects, Join Points, Pointcuts and Advices

While describing the example of an editor for graphical elements (Figure 2.1), we stated that the display updating concern can be modularised within an independent unit and executed at appropriate times. To enable this, AOP introduced the ***aspect*** as a new conceptual implementation unit to modularise crosscutting concerns.

[1]In Figure 2.1(b), the ≪`crosscuts`≫ stereotype is used to indicate the modules affected by the `DisplayUpdate` aspect.

Aspects have the ability of implicitly modifying the behaviour of a program at specific points during its execution. Each of these points is called a ***join point*** (JP) (Kiczales et al., 2001b), and is characterised as a well-defined point in a program execution in which aspects can insert additional behaviour as well as replacing the existing one.

An important matter in AOP regards the identification of join points in a program. According to Elrad et al. (2001b), AOP supporting technologies must implement a model that enables the software engineer to precisely identify join points in which aspect-related behaviour will be executed. This join point model is realised by means of *pointcut descriptors* (PCD), or simply ***pointcuts*** (Kiczales et al., 2001b). A pointcut is a semantics- or language-based matching expression that identifies a set of join points that share some common characteristic (e.g. based on properties or naming conventions).

Once a join point is identified using a PCD during the program execution, behaviour that is encapsulated within method-like constructs named ***advices*** starts to run. Advices can be of different types depending on the supporting technology. In general, advices can be defined to run at three different moments when a join point is reached: before, after or in place of it, as implemented in the AspectJ language (Kiczales et al., 2001b) (more details in Section 2.1.1).

The Weaving Process

AOP enhances the modularisation of crosscutting concerns through the use of aspects that are implemented separately from the base code. However, base code and aspects need to be combined in order to work together to fulfil the requirements they are intended to. This combination step is called ***weaving process*** (Kiczales et al., 1997) and is performed by a tool called *aspect weaver*. This process is depicted by Figure 2.2. The aspect weaver receives the base code and aspects as input, and produces as output the woven application that includes both the base code functionalities and the aspectual behaviour.

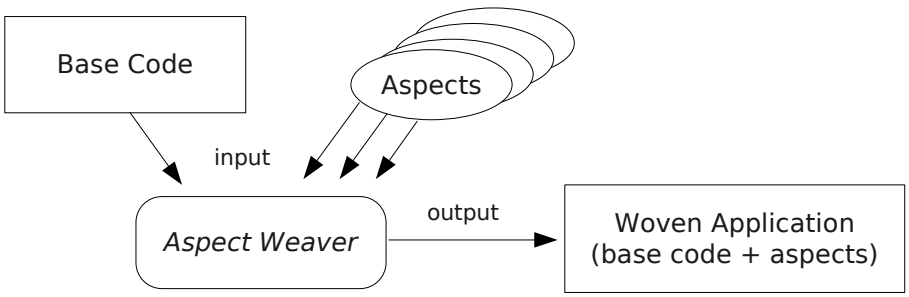


Figure 2.2: Basic elements of an AOP supporting system – adapted from Kiczales et al. (1997).

In general, three different approaches can be employed in the weaving process (Popovici et al., 2002): (i) *compile-time weaving*, in which base code and aspects are merged together to generate a new application that includes all functionalities; (ii) *load-time weaving*, which weaves based code and aspects at the moment the base modules (e.g. classes) are loaded into the executing environment (e.g. a Java Virtual Machine); and (iii) *runtime weaving*, also called dynamic weaving, in which aspects are dynamically invoked during the system execution. AspectJ, which is further described in Section 2.1.1 , implements the two first weaving approaches, i.e. compile-time and load-time weaving.

Quantification and Obliviousness

Filman and Friedman (2004), two pioneers in AOP research, advocate that *quantification* and *obliviousness* are two fundamental properties of AOP. ***Quantification*** refers to the ability of declaratively selecting sets of points via pointcuts in the execution of a program. ***Obliviousness***, on the other hand, implies that the developers of core functionality need not be aware of, anticipate or design code to be advised by aspects. The relationship between these two properties is defined by Filman and Friedman (2004) as follows: *"AOP can be understood as the desire to make quantified statements about the behaviour of programs, and to have these quantifications hold over programs written by oblivious programmers"*.

These two properties are realised by the join point models. In general, such models enables the software engineer to quantify join points using PCDs at different levels of abstraction (e.g. design- or source code-level), and to execute aspectual behaviour that the base code has no previous knowledge.

A relevant note regarding the obliviousness property is that, in the early phases of research on AOP, it was considered to be a mandatory property for AO software development (Filman and Friedman, 2000). However, Sullivan et al. (2005) show the results of a case study where the oblivious implementation suffers from several problems regarding software evolution. Indeed, recent results show that obliviousness may negatively impact the fault-proneness of AO programs (Ferrari et al., 2010a). In this sense, recent methods and languages for AOP can help to ameliorate this problem. Examples of such approaches are aspect-aware interfaces (Kiczales and Mezini, 2005), Crosscut Programming Interfaces (XPIs) (Griswold et al., 2006) and Explicit Join Points (EJPs) (Hoffman and Eugster, 2007). Although they reduce the obliviousness among system modules, these approaches help to improve the program comprehension by making aspect-base interaction more explicit.

2.1.1 Programming Languages and other Supporting Technologies

AOP has motivated the development of several supporting technologies. In particular, languages and frameworks have received special attention, providing AOP capabilities for existing languages such as Java, C, C++, C#, Smalltalk and PHP. In general, AOP languages and frameworks implement the four basic elements required for an AOP supporting system (Elrad et al., 2001b): a join point model, a pointcut language, an implementation unit that encapsulates the pointcuts and advices, and a weaver.

We next provide a brief overview of some of these languages. Considering the generally higher focus on Java-based AOP than on other languages, the overview is split into two categories: AOP in Java and AOP in other languages.

AOP in Java

AspectWerkz (Bonér and Vasseur, 2005), JBossAOP (Burke and Brock, 2003) and SpringAOP (Johnson et al., 2007) are examples of frameworks that support AOP in Java. In AspectWerkz, aspects can be implemented as regular Java classes as well as within XML files. In both cases, join points are identified using XML-based specifications, and the AspectWerkz API provides means for base code and aspect weaving at either compile-time, load-time or runtime.

Similarly to AspectWerkz, JBossAOP supports the implementation of crosscutting behaviour as regular Java classes that must implement a specific interface, and join points are selected via XML specifications. SpringAOP, on the other hand, is part of the Spring Framework (Johnson et al., 2007) and supports the implementation of advices as well as pointcuts in regular Java classes, without requiring any additional compilation step (i.e. the weaving process).

AspectJ (Kiczales et al., 2001b) is a Java-based general purpose language to support AOP. It was developed by the AOP proponents and is currently maintained by the Eclipse Foundation (The Eclipse Foundation, 2010a). In this thesis, special emphasis is given to AspectJ due to two main reasons: firstly, it is the most investigated and used AOP language and has been the basis for the development of several other AOP languages. Secondly, together with Java, AspectJ underlies our research, whose results are presented in the remaining chapters of this dissertation. Therefore, AspectJ is more extensively described in Section 2.1.1.1.

CaesarJ (Mezini and Ostermann, 2003) is a Java-like language that focuses on software reuse. The language combines AspectJ-like AOP constructs with enhanced object-oriented modularisation mechanisms (Aracic et al., 2006; Mezini and Ostermann, 2003). In partic-

ular, CaesarJ's join point and advice models are very similar to the models implemented in AspectJ.

AOP in other Languages

Several other programming languages have been extended in order to support AOP, either object-oriented, procedural or script-based. We next list some of them, but the set of available languages is certainly larger than the one presented in the sequence.

In regard to C-like languages, Gal et al. (2001) proposed the AspectC++ language in extension to C++. It implements join point and advice models that are very similar to AspectJ, however with limitations with regard to primitive pointcut designators (see Section 2.1.1.1 for more details about AspectJ pointcut designators). Likewise C and C# languages were extended to support AOP, resulting in the extensions AspectC (Gong et al., 2006) and AspectC# (Kim, 2002), respectively. In all cases, a language-specific weaver is responsible for combining the base code and aspects into an executable system.

Other language extension initiatives can also be identified. Hirschfeld (2001) designed AspectS to support AOP in Smalltalk within the Squeak environment[2]. Another example is the AOPHP language (Stamey et al., 2005) that supports AOP in PHP. Both languages' join point and advice models are similar to the ones AspectJ implements.

2.1.1.1 The AspectJ Language

AspectJ is a general purpose AOP language that extends Java by introducing a series of novel constructs. It was originally developed in the Xerox Palo Alto Research Center and is currently maintained by the Eclipse Foundation (The Eclipse Foundation, 2010a). Note that AspectJ is defined as superset of Java, which means that any Java program can be compiled using a standard AspectJ compiler (The AspectJ Team, 2005b).

The documentation of AspectJ release 1.2.1 was used as a basis for the language description presented in this section (The AspectJ Team, 2003). Such version is Java 2 compliant. Additional details of AspectJ that address Java 5 features (e.g. generics, annotations and enumerated types) can be obtained in the AspectJ 5 Developers Notebook (The AspectJ Team, 2005a).

In AspectJ programs, crosscutting code can be implemented in either static or dynamic ways (Kiczales et al., 2001b). Dynamic crosscutting, on the one hand, is realised by the use of pointcuts and advices, which activate aspectual behaviour when join points are reached during the program execution. On the other hand, static crosscutting consists

[2]Available at `http://www.squeak.org/` - last accessed on 17/03/2010.

of structural modifications of modules that compose the base code. These modifications are achieved by the so-called *intertype declarations* (ITDs) or simply *introductions* (The AspectJ Team, 2003). Examples of intertype declarations are the introduction of a new member (e.g. an attribute or a method) into a base module or a change in the class' inheritance (e.g. a newly declared superclass).

Join point shadows: The compilation of an AspectJ program results in standard Java bytecodes (Hilsdale and Hugunin, 2004). Each aspect is converted into a Java class at bytecode level. Advices implemented within an aspect result in static methods (also known as *advice-methods*) in the respective class file. Their parameter lists are formed by information obtained from the join points that activate the advice executions. In regard to the base code, this is modified in order to introduce the static *join point shadows*, which consist of the sets of implicit calls to advice-methods (Hilsdale and Hugunin, 2004). Such calls are inserted into the base code during the compilation process. The advice execution itself may only be resolved at runtime, that is, not all join points included in a joint point static shadow will in fact trigger the advice execution.

Figure 2.3 shows an example of AspectJ code. It consists in a possible implementation for the simple graphical editor early introduced in Figure 2.1. This example is used in the sequence to describe some of the main programming constructs available in AspectJ.

AspectJ PCDs: As explained earlier in Section 2.1, a join point is a well-defined point in a program execution. In the AspectJ join point model, candidate join points can be, for example, a method call, a method execution, an attribute value update, an object initialisation or an exception handler. An advice execution is also a candidate join point.

An AspectJ PCD is an expression that matches certain characteristics of the base program, thus selecting a (possibly empty) set of join points. For example, in Figure 2.3, a PCD named `update` is defined in lines 39–41. The right-hand side of the PCD expression defines the set of intended join points to be selected. For this, AspectJ offers a set of predefined pointcuts, called *primitive pointcut designators* (The AspectJ Team, 2003). Besides, the developer can benefit from a set of special characters named *wildcards*. The wildcards can be used for defining pattern-based PCDs that may select sets of join points with common characteristics.

The `update` PCD defined in lines 39–41 includes two `call` primitive pointcut designators and the "`*`", "`+`" and "`..`" wildcards. The join points captured by this PCD include: (i) calls to the `moveBy` method that belongs to the `Point` class; and (ii) calls to methods whose names start with `set` and are implemented by the `FigureElement` as well

```
1  interface FigureElement {
2          public void moveBy(int x,int y);
3  }
4
5  class Point implements FigureElement {
6          private int x,y;
7          public Point(int x, int y) { this.x = x; this.y = y; }
8          public void setX(int x) { this.x = x; }
9          public void setY(int y) { this.y = y; }
10         public int getX() { return this.x; }
11         public int getY() { return this.y; }
12
13         public void moveBy(int x, int y) {
14                 this.x += x;  this.y += y;
15         }
16 }
17
18 class Line implements FigureElement {
19         private Point p1, p2;
20         public Line(Point p1, Point p2) { this.p1 = p1; this.p2 = p2; }
21         public void setP1(Point p1) { this.p1 = p1; }
22         public void setP2(Point p2) { this.p2 = p2; }
23         public Point getP1() { return this.p1; }
24         public Point getP2() { return this.p2; }
25
26         public void moveBy(int x, int y) {
27                 this.p1.moveBy(x,y);
28                 this.p2.moveBy(x,y);
29         }
30 }
31
32 class Display {
33         public static void update() {
34                 /* commands to update the screen */
35         }
36 }
37
38 aspect DiplayUpdate {
39         public pointcut update():
40                 call(public void Point.moveBy(int,int)) ||
41                 call(public void FigureElement+.set*(..));
42         after() returning: update() {
43                 Display.update();
44         }
45 }
46
47 public class MainEditor {
48         public static void main(String[] args) {
49                 Point point1 = new Point(1,2);
50                 Point point2 = new Point(6,7);
51                 Line line = new Line(point1,point2);
52                 line.moveBy(5,5);
53         }
54 }
```

Figure 2.3: A simple graphical editor written in AspectJ.

as by any of its subclasses. In the latter case, "`*`" enables the generalisation of method names, whereas the "`+`" wildcard defines that subclasses of `FigureElement` should also be included in the set of analysed classes. Besides, the "`..`" wildcard states any matched method call should be included, regardless its number of parameters and their types. Note

that logical operators (i.e. AND ("`&&`"), OR ("`||`") and NOT ("`!`")) can be used to build compound PCDs, as we can observe in the `update` PCD.

AspectJ Advices: AspectJ provides three different types of advices: `before`, `after` and `around`. While the two first have self-explanatory names, `around` advices wrap the execution of the original join points when these are reached during the program execution. `around` gives the developer the option of also running the behaviour implemented in the join point by providing the `proceed` command.

The three types of advices are summarised in Table 2.1. Note that `after` has two additional forms – `after returning` and `after throwing`– which are also described in the table.

Table 2.1: Advice types in AspectJ.

Advice type	Description
`before`	Runs before the normal execution of the join point.
`after`	Runs after the normal execution of the join point.
`after returning`	Runs after the normal execution of the join point, that is, when the execution of the join point does not throw any exception.
`after throwing`	Runs after the abnormal execution of the join point, that is, when an exception is thrown but not handled during the join point execution.
`around`	Runs in place of the join point. Optionally, it can trigger the execution of the join point with the `proceed` command.

Context Exposure in AspectJ: In addition to selecting join points in the base program, a PCD can also capture context information and make it available to be handled inside the advices. For example, the `DiplayUpdateExposingContext` aspect presented in Figure 2.4 is a slightly modified version of the `DisplayUpdate` shown in Figure 2.3. The `update` PCD now includes the `target` primitive pointcut designator, which captures join points where the target object (e.g. in a method call) is of the same type as declared in the PCD's left-hand side.

In the example displayed in Figure 2.4, objects of type `FigureElement` are bound to the `fig` variable, which can be handled inside the advice that runs after the selected join points. Other examples of context information that can be exposed are method parameters, the caller object and return values.

```
1 aspect DiplayUpdateExposingContext {
2         public pointcut update(FigureElement fig):
3                 (call(public void Point.moveBy(int,int)) ||
4                 call(public void FigureElement+.set*(..))) && target(fig);
5         after(FigureElement fig) returning: update(fig) {
6                 Display.update();
7                 // commands to handle the target object
8         }
9 }
```

Figure 2.4: Example of context exposure in AspectJ.

Static Crosscutting in AspectJ: Static crosscutting in AspectJ programs is supported by ITDs and other `declare`-like constructs (The AspectJ Team, 2003). They are summarised in Table 2.2.

Table 2.2: Main constructs for static crosscutting in AspectJ.

Construct	Description
ITDs	Allow the insertion of members (attributes and methods) into the base modules.
`declare parents`	Changes the type hierarchy of the system by specifying that a base class extends another one or implements a specific interface.
`declare soft`	Specifies that an exception, if thrown at a join point, is converted to an unchecked exception.
`declare precedence`	Defines the advice execution order when a join point is advised by two or more advices.
`declare error/warning`	Signals error or warning messages when specific join points are matched by a PCD.

The code printed in Figure 2.5 is an example of static crosscutting implemented in AspectJ. In the `TransformationsAspect`, line 6 states that `FigureElement` objects must implement the `IRotate` interface, which declares the `rotate` method's signature. In the sequence, lines 8–10 and 12–14 introduce the `rotate` method into the two subclasses of `FigureElement`, i.e. `Point` and `Line`. In this way, the required interface implementation for `FigureElement` objects is fulfilled by means of ITD members.

The arbitrary execution order of advices that share common join points is pointed out as one of the fault sources in AO programs (Alexander et al., 2004). In AspectJ, the execution order can be defined with the `declare precedence` expression. We use an example to describe how the language handles the execution of two advices implemented in different aspects but that share a common join point. Be `A` and `B` two aspects whose precedence is defined as:

```
declare precedence:  A, B;
```

```
1  interface IRotate {
2          public void rotate(int axis, int degrees);
3  }
4
5  aspect TransformationsAspect {
6          declare parents: FigureElement implements IRotate;
7
8          public void Point.rotate(int axis, int degrees) {
9                  // commands for rotating a Point object
10         }
11
12         public void Line.rotate(int axis, int degrees) {
13                 // commands for rotating a Line object
14         }
15 }
```

Figure 2.5: Example of static crosscutting in AspectJ.

Be `adviceA` and `adviceB` two advices of the same type implemented in `A` and `B`, respectively. The execution precedence rules for `adviceA` and `adviceB` are summarised in Table 2.3.

Table 2.3: Advice precedence rules in AspectJ.

Advice type	Execution order
`before`	`adviceA` runs before `adviceB`.
`after`	`adviceA` runs after `adviceB`.
`around`	`adviceA` runs and, if it invokes the `proceed` command, then `adviceB` runs.

2.2 Foundations of Software Testing

The main goal of software testing is revealing faults in software products. According to Myers et al. (2004, p. 1-2, p. 18), software testers should not only make sure a program in fact does what is was designed to do, but also make sure the program does not do what it is not expected to do. They define testing as *"the process of executing a program with the intent of finding errors"*. Therefore, a test is considered successful when it reveals one or more faults in the analysed program.

In general, it is not possible to prove a program is correct due to the absence of faults (Myers et al., 2004, p. 43). However, when tests are performed in a systematic and rigorous fashion, it helps to increase the stakeholders' confidence that the software behaves as expected (Harrold, 2000; Weyuker, 1996).

In the remaining of this section, we introduce the basic terminology and concepts related to software testing. Besides, we describe the more widely investigated techniques

that are typically applied to programming paradigms such as procedural and OO programming.

2.2.1 Basic Terminology

Fault, error and failure: In this dissertation, we adopted definitions for *fault*, *error* and *failure* that are aligned with the IEEE 610.12-1990 standard (IEEE, 1990), as follows:

- A ***fault*** is a difference between the actually implemented software product and the product that is assumed to be correct. An example of a fault is an incorrect step, process or data definition.
- An ***error*** is characterised by an internal state of the software product that differs from the correct state, usually due to the execution of a fault in such product; and
- A ***failure*** is the external manifestation of an error, that is, it is an error that goes across the system boundaries and becomes visible to the users.

Testing phases: The testing activity is traditionally divided into the four phases that are presented in the sequence (Myers et al., 2004, p. 91-118, p. 129-130; Pressman, 2005, p. 362-379). Note that other alternative classifications may split these phases into two or more sub-phases, or even group two or more of them into a single phase (Beizer, 1990, p. 21-22; Sommerville, 2007, p. 538).

- ***Unit testing:*** the main goal of this phase is to test each software unit in isolation, with the intent of revealing faults related to the implemented logic.
- ***Integration testing:*** this phase aims to identify problems related to the interface amongst the units of a piece of software. For example, it tests the elements involved in the communication between two units that have been previously tested in isolation at the unit testing phase.
- ***Validation testing:*** it is performed after the integration testing and aims to reveal faults related to coarse-grained functions and performance that do not conform to the specification.
- ***System testing:*** in this phase, tests take into account all elements involved in the software execution (e.g. hardware, stakeholders and database). The main goal is to assess the software in terms of its general functionalities and performance.

In this dissertation, a ***unit*** is considered a software portion that cannot be subdivided into smaller parts, in accordance with the IEEE 610.12-1990 standard (IEEE, 1990). For instance, a unit can be a function in a procedural program, a method in an OO program or an advice in an AO program. A ***module***, on the other hand, is a collection of inter-related units (e.g. a class or an aspect) that interact through well-defined interfaces (e.g. a method call) or through indirect invocations (e.g. an advice triggering).

Note that the adopted definition for *unit* may impact on the definition of the testing phases. For example, Vincenzi (2004) also assumes the method as the smaller portion of code in his approach for testing OO programs. In doing so, he defines that the unit testing phase includes only intra-method evaluations, and that integration testing includes inter-method, intra-class and inter-class levels. However, if he had taken the classes as the units of an OO system, the unit testing phase would encompass intra-method, inter-method and intra-class levels, whilst integration testing would include only the inter-class level. Nevertheless, the following basic tasks need to be carried out in each testing phase (Sommerville, 2007, p. 539; Myers et al., 2004, p. 145-147): (1) test planning; (2) test case design; (3) test execution; and (4) test evaluation. These tasks should be performed along the full software development process, from the planning to the maintenance phase (Sommerville, 2007, p. 81; Pressman, 2005, p. 356).

Test case, testing criterion, testing technique: Formally, a ***test case*** t can be defined as a tuple $(d, O(d))$, where d represents an element from a specific input domain D, $d \in D$, and $O(d)$ is the expected output when the software is executed having d as the input. In other words, a test case consists of a value (or range of values) that should be provided to the software and the output that is expected after the software execution on that value or range of values (Myers et al., 2004, p. 14). Once a test case is executed, one needs to observe the program behaviour to decide whether it is correct or not. The entity that performs such check, be it the tester or an automated tool, is known as the ***oracle*** (Mathur, 2007, p. 27).

With respect to the input domain D, dealing with its size poses a great challenge for researchers and practitioners. The difficulty resides in the fact that the number of elements in D, if considered as a whole, may be very large or even infinite for certain scenarios. Therefore, one needs to find ways of systematically defining subsets of D, hence reducing the number of test cases required to achieve adequate test quality. A solution for this is the definition of ***testing criteria***, which consist of sets of rules for defining subsets of D (Frankl and Weyuker, 2000). A test criterion defines elements from a program (or any other software product) that should be exercised during the software

execution, thus guiding the software engineer throughout the test case designing process. The elements that are required to be exercised according to a testing criterion are called ***test requirements***. A test requirement can be, for example, a specific path in the logic of a unit, or a functionality obtained from the software specification. Note that some test requirements may be impossible to be exercised, e.g. when the program semantics hinders the design of a test case to cover some associated test requirement (Offutt and Pan, 1997). Such test requirements are considered ***unfeasible*** and may incur additional effort to be identified during the testing activity.

Each software criterion is associated with a particular ***testing technique***. The several existing techniques differ from each other based on the underlying software artefacts they require to derive the test requirements. The testing criteria, on the other hand, are used as guidelines that constrain the amount of test requirements. The main testing techniques and their associated criteria are described in the next section.

2.2.2 Testing Techniques and Criteria

One or more testing techniques can be applied in a given testing phase. Amongst the most widely investigated techniques, we can highlight *functional testing*, *structural testing*, *state-based testing*[3] and *fault-based testing*. Each of them relies on underlying artefacts upon which the associated criteria define the test requirements. For example, functional testing requires specification documents (e.g. user requirements), state-based testing requires state models (e.g. UML Statecharts) and fault-based testing requires well-characterised fault types (e.g. fault taxonomies). Besides that, these techniques have been explored in the context of testing of formal software specifications such as Finite State Machines (Fabbri et al., 1994), Statecharts (Fabbri et al., 1999), SDL Specifications (Sugeta et al., 2004) and Coloured Petri Nets (Simão et al., 2003).

2.2.2.1 Functional Testing

The functional testing technique is based on software specification documents to derive the test requirements. It is also known as *black-box* testing (Myers et al., 2004, p. 9), since it does not take into account internal implementation details to be applied; basically, only functional requirements are considered. It is traditionally used to demonstrate that the

[3]Due to the behavioural property of objects in OO systems, the state-based testing can be classified as a testing technique (Binder, 1999), despite the possibility of applying different techniques to state-based system representations (e.g. fault-based testing (Fabbri et al., 1994, 1999)).

software behaves accordingly by observing that the inputs are accepted by the software and the outputs are produced as specified.

Performing functional testing comprises two main steps: (1) identifying the functions the software should perform; and (2) designing test cases that check whether such functions are accomplished accordingly or not. These steps can be carried out by applying the following criteria:

- ***Equivalence Partitioning:*** it consists of splitting the input and output domains into classes of data, both valid and invalid, according to the specification of the targeted functionality (Myers et al., 2004, p. 52). Each class is said to be an *equivalence class*. In the sequence, a minimum set of test cases is designed containing at least one representative element for each equivalence class.

- ***Boundary-Value Analysis:*** it extends the Equivalence Partitioning criterion by requiring test cases composed by elements that are at the boundaries of each equivalence class (Myers et al., 2004, p. 59). It is motivated by the fact that several faults are related to the boundary input or output values of equivalence classes.

- ***Systematic Functional Testing:*** it aggregates Equivalence Partitioning and Boundary-Value Analysis into a single criterion (Linkman et al., 2003). In general terms, it requires that two test cases be created to exercise each equivalence class in order to uncover faults that might keep "hidden" with the execution of a single test case for a given equivalence class.

2.2.2.2 Structural Testing

Differently from the functional testing technique, structural testing is based on internal implementation details of the software. This technique – also called *white-box testing* – is concerned with the coverage degree of the program logic yielded by the tests (Myers et al., 2004, p. 44).

Usually, structural-based testing criteria utilise a program representation called *control flow graph* (CFG), or *program graph*, to derive the test requirements. An example[4] of CFG is presented in Figure 2.6. Each node in a CFG represents a disjoint block of sequential statements. The execution of the first statement in a block implies the execution of all subsequent ones in that block. Besides, each statement has a single predecessor as well as a

[4]Note that the information added to nodes and edges of the CFG depicted in Figure 2.6 is used to build the *Def-Use Graph* which is described in the sequence. The CFG itself is composed only by nodes and edges.

single successor, except the first, which can possibly have more than one predecessor, and the last node, which may have more than one successor (Rapps and Weyuker, 1982). In a CFG node, the label represents its first statement, e.g. the line of code of this statement in the source code.

The example depicted in Figure 2.6 was extracted from Masiero et al. (2006b). On the left-hand side, we can observe a Java method for validating some language-specific identifiers. The CFG that represents this method is shown on the right-hand side. We can observe that a CFG can have additional information about each variable being defined and/or used in each graph element. In this case, the graph is also known as *Def-Use Graph* (Rapps and Weyuker, 1982), or simply DUG. In a DUG, variables have two different use-related notations: computation-use (*c-use*) or predicate-use (*p-use*). While a c-use of a variable is associated with a node in a DUG, p-uses are associated with edges.

For example, the node #3 in Figure 2.6 includes the definition of the `achar` and `valid_id` variables. This node also includes the c-use of the `achar` and `s` variables. On the other hand, the `s` and `valid_id` variables are used in a predicate within the node #3 (statement "`if (s.length() == 1 && valid_id)`"), thus being associated with its outgoing edges.

Several testing criteria based on the CFG and on the DUG have been proposed to date. Examples of such criteria are briefly described as follows:

Control Flow-based Criteria

- ***All-nodes:*** it requires the execution of every node of a unit represented in a CFG (Myers et al., 2004, p. 44-45; Rapps and Weyuker, 1982).

- ***All-edges:*** it requires the traversal of all edges represented in a CFG (Myers et al., 2004, p. 45-52; Rapps and Weyuker, 1982).

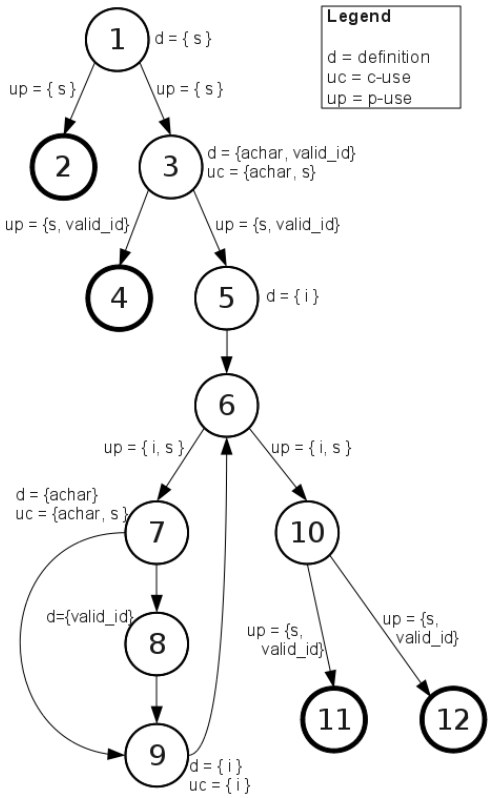
- ***All-paths:*** it requires the traversal of all paths in a given CFG (Myers et al., 2004, p. 11; Rapps and Weyuker, 1982). According to Myers et al. (2004, p. 13), this criteria is impracticable since it may be impossible to achieve adequate test sets.

Data Flow-based Criteria

- ***All-defs:*** it requires that each variable defined in a DUG be exercised by at least one test case that reaches either a c-use or a p-use of this variable before it is redefined (Rapps and Weyuker, 1982).

```
            public static boolean verify(String s)
            {
/* 01 */      if (s == null || s.length() == 0)
/* 02 */        return false;
/* 03 */      char achar;
/* 03 */      boolean valid_id;
/* 03 */      valid_id = true;
/* 03 */      achar = s.charAt(0);
/* 03 */      valid_id = valid_s(achar);
/* 03 */      if (s.length() == 1 && valid_id)
/* 04 */        return true;
/* 05 */      int i = 1;
/* 06 */      while (i < s.length()) {
/* 07 */        achar = s.charAt(i);
/* 07 */        if (!valid_f(achar))
/* 08 */          valid_id = false;
/* 09 */        i++;
/* 09 */      }
/* 10 */      if (valid_id && (s.length() <= 6))
/* 11 */         return true;
/* 12 */      return false;
/* 12 */    }
```

a. Source code



b. *Def-Use Graph*

Figure 2.6: Example of a DUG– adapted from Masiero et al. (2006b).

- ***All-uses:*** it requires that all c-uses and p-uses of each variable defined in the DUG be exercised by test cases that reach such uses, always through paths where the variable is not redefined (Rapps and Weyuker, 1982). ***All-c-uses*** and ***All-p-uses*** are variants of the All-uses criterion (Rapps and Weyuker, 1982).

- ***All-potential-uses:*** it requires that all paths between a variable definition and the other reachable nodes in the DUG be exercised by test cases, always through paths where the variable is not redefined (Maldonado, 1991).

Complexity-based Criteria

- ***McCabe criterion***: builds on the cyclomatic complexity of the CFG to derive the test requirements. It requires that each linearly independent path in the CFG be exercised by the test set (McCabe, 1976).

2.2.2.3 State-Based Testing

State-based testing consists in deriving test requirements based on the dynamic behaviour of the software. It is typically applied in testing of OO systems, which represents the dynamic behaviour of classes that compose the system using Finite State Machines (FSM) (Turner and Robson, 1993). In state-based testing, a FSM is used to represent the possible states an object can take. The state of an object is generally characterised by its set of attribute values at a given moment during the software execution. Domain partitioning techniques can be employed to reduce both the amount of values and the combination among them.

A FSM can include either a single class or a set of inter-related classes. Once the state model is available, a transition tree can be derived from it in order to represent all possible transitions among states. Testing criteria, which are based on specific subsets of transitions, can then be applied in order to systematically guide the creation of test cases. Examples of criteria proposed by Offutt et al. (2003) are:

- ***Transition coverage:*** it requires that all transitions among states in a FSM be exercised by the test set.
- ***Full predicate coverage:*** it requires that every transition that depends on a predicate to be taken in the FSM be evaluated true and false by the test set.
- ***Transition-pair coverage:*** it requires that every pair of adjacent transitions in a FSM be exercised in sequence by at least one test in the test set.
- ***Complete sequence:*** it requires that every meaningful sequence of transitions in a FSM be exercised by the test set. Such sequences should be chosen based on the expertise of the testing engineer.

The state-based testing technique has also been investigated in the context of AO software (Badri et al., 2005; Xie and Zhao, 2006; Xu and Xu, 2006a,b). Several testing criteria have been proposed, focusing particularly on state transitions that are affected by aspectual behaviour. Some of these approaches are described later in this chapter.

2.2.2.4 Fault-Based 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 (Morell, 1990). Two examples of fault-based testing criteria are *Error Seeding* and *Mutant Analysis*, which are following described.

- ***Error Seeding:*** it consists in randomly inserting (i.e. *seeding*) a pre-defined number of faults into the software. In the sequence, the faulty program is run on the test data in order to reveal faults, from which the rate of real and artificial faults is calculated (Mills, 1972) apud (Budd, 1980).

- ***Mutant Analysis:*** originally proposed by DeMillo et al. (1978), it consists in creating several slightly modified versions of a program. The intent is to simulate faults commonly introduced into the programs and to check if the test data is sensitive enough to reveal these faults. Due to its relevance to this thesis, the Mutant Analysis criterion is better described in the sequence.

The Mutant Analysis Criterion

A recent survey undertaken by Jia and Harman (2010) showed that the Mutant Analysis criterion – or simply *mutation testing* – has been extensively and increasingly investigated in the last three decades. Several approaches address mutation testing at several levels of software abstraction, ranging for formal specification to source code level testing.

The basic idea behind mutation testing consists in creating several versions of the original program, each one containing a simple fault. These modified versions are called *mutants* and are all expected to behave differently from the original program (DeMillo et al., 1978). In other words, any mutant that is executed against the test data should produce a different output when compared to the execution of the original program.

Mutation testing is underlain by the *Competent Programmer* and the *Coupling Effect* hypotheses (DeMillo et al., 1978). The first states that any program P that is ready to be tested is correct or nearly correct. Therefore, it assumes that any fault artificially introduced into P should be detected by the current test set. The second hypothesis, on the other hand, states that test data which distinguishes P from its variant P' containing a simple fault is also able to distinguish variants of P that include complex faults. In other words, complex faults are coupled to simple faults, according to empirical principles (DeMillo et al., 1978). Together, these hypotheses state that a program under test contains only small syntactic faults and that complex faults result from the combination of them. Consequently, fixing the small faults will probably solve the complex ones.

In mutation testing, ***mutation operators*** encapsulate the modification rules applied to P. The criterion requires the creation of a set M of mutants of P, resulting from the

application of mutation operators to it. 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 $P(t) \neq S(t), (t \in T)$ (Mathur, 2007, 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 (DeMillo et al., 1978): (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 following formula:

$$ms(P,t) = \frac{DM(P,T)}{M(P) - EM(P)}$$

where:

ms(P,T) is the *mutation score*;
DM(P,T) is the number of mutants killed by T;
M(P) is the total number of mutants created from p; and
EM(P) is the number of mutants of P that are equivalent to P.

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 (Mathur, 2007, p. 519).

Mutation analysis beyond unit testing of programs: Originally proposed for the unit testing of programs, mutation testing has also been investigated in the context of integration testing (Delamaro et al., 2001). Moreover, it has been applied to several other software specification models such as Finite State Machines (Fabbri et al., 1994), Statecharts (Fabbri et al., 1999), Estelle (Souza et al., 1999) and coloured Petri Nets (Simão et al., 2003). As highlighted by Vincenzi et al. (2006b), this is possible because mutation testing only requires an executable software artefact to be applied.

Alternative mutation testing approaches and cost reduction techniques: several variants of the Mutant Analysis criterion have been proposed along the years. They aim to either reduce its application complexity or decrease its application cost. For example, as originally proposed by DeMillo et al. (1978), mutation testing requires the complete execution of the program and its mutants in order to decide whether the mutant is killed or not. This "conservative" approach is known as ***strong mutation*** (Marick, 1991). ***Weak mutation*** (Howden, 1982), on the other hand, requires the evaluation of the mutant state right after the execution of the modified portion of code; if the state of the original program differs from state of the mutant at the same point, the mutant is set as dead. Another variant of the criterion is the ***firm mutation*** (Woodward and Halewood, 1988), which requires the evaluation of the original program' and its mutants' state at some moment after the execution of the mutated code. Both approaches – weak and firm mutation – tend to reduce the complexity by means of simplified test case design and execution; moreover, they tend to decrease costs since complete program and mutant executions are no longer required.

Other cost reduction techniques for mutation testing focus on reducing the number of employed mutation operators. Examples are the ***Constrained Mutation*** (Mathur and Wong, 1993) and ***Selective Mutation*** (Offutt et al., 1993). They both focus on reducing the number of mutants by applying a subset of the mutation operators, although still keeping the efficacy of the criterion in regard to the quality of the derived test suites.

2.2.3 Test Evaluation and Comparison amongst Criteria

The evaluation and comparison of testing criteria can be undertaken in several different ways. These activities are useful for establishing the cost-benefit relationship of a given criterion, hence having impact on the definition of testing strategies that might be adequate within certain contexts. According to Wong (1993), three properties may be taken into consideration in theoretical and empirical studies that compare different criteria: (i) *cost*, which can be measured in several ways (e.g. the number of test requirements or the number of test cases required to cover the test requirements); (ii) *effectiveness*, which regards the fault discovery ability of a criterion given an adequate test set is available for it; and (iii) *strength*, which consists in the likelihood of satisfying a criterion C_2 using a test set that is adequate (i.e. satisfies) another criterion C_1.

The strength of a criterion can be investigated in terms of the *subsume relation* among adequacy criteria (Zhu et al., 1997). This relation can be defined as follows: Let C_1 and C_2 be two testing criteria, and let T be a test set that is C_1-*adequate*, that is, T is a test

set that covers all test requirements produced by C_1. We can state that C_1 *subsumes* C_2 if and only if T is C_2-*adequate* for all cases, i.e. all possible programs to which C_1 and C_2 can be applied.

Theoretical studies performed by Maldonado (1991) resulted in the subsume relation depicted in Figure 2.7. It extends a previous study from Rapps and Weyuker (1982) in order to include the *Potential-Uses* family criteria. The relation presented in the figure includes the structural-based criteria presented in Section 2.2.2, also considering some variants of them, e.g. *all-c-uses/some-p-uses* (Rapps and Weyuker, 1982) and *all-potential-du-paths* (Maldonado, 1991). Note that this relation considers that the CFG includes only feasible paths, otherwise the subsume relation does not hold, as observed by Frankl and Weyuker (1988).

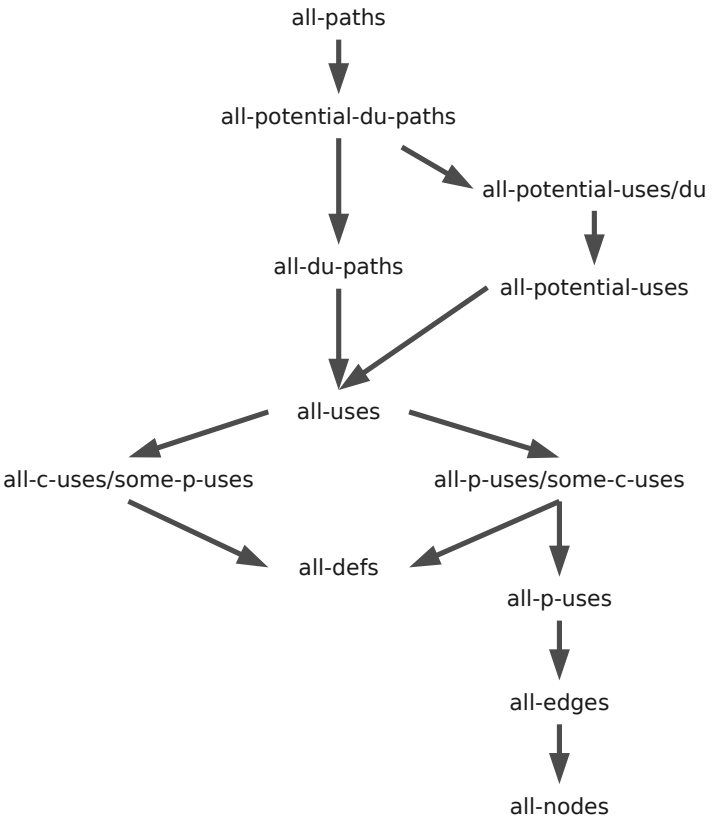


Figure 2.7: Subsume relation amongst structural-based test selection criteria – adapted from Maldonado (1991) and Rapps and Weyuker (1982).

The subsume relation amongst criteria derived from different testing techniques (e.g. structural- and fault-based) cannot be theoretically defined (Delamaro, 1997; Wong, 1993). However, empirical evidence has shown that, amongst the most investigated criteria, Mu-

tant Analysis seems to be the strongest one (Mathur and Wong, 1994; Offutt et al., 1996b; Wong, 1993) as well as the most effective criterion (Li et al., 2009; Wong and Mathur, 1995). For example, Wong (1993) and Mathur and Wong (1994) concluded that mutation testing is stronger than the *all-uses* criterion (Rapps and Weyuker, 1982), given that test sets that were adequate to Mutant Analysis have also shown to be adequate for *all-uses*. Note that it did not hold when they considered the other way around. Additional evidence in regard to it was provided by Li et al. (2009), Wong and Mathur (1995) and Offutt et al. (1996b). In another study, Souza (1996) showed that Mutant Analysis and *all-potential-uses* (Maldonado, 1991) are not comparable with respect to the strength property.

2.2.4 Test Automation

Software testing needs to be performed following a systematic and rigorous process in order to enhance the users' confidence that the software behaves as expected. Moreover, testing helps to demonstrate the software achieves expected quality attributes such as reliability and correctness (Harrold, 2000; Weyuker, 1996). To achieve these goals, however, software testing strongly relies on automated tool support. Without adequate support, testing may be costly, error-prone and limited to small programs (Vincenzi et al., 2006a).

Harrold (2000) suggests that the development of practical methods and tools to support software testing can help software engineers to create high quality products. In order to achieve this, it is required that research comprising new testing approaches with associated tool support be extensively carried out. According to Harrold, the achievements would facilitate the technology transfer to the industry.

Harrold also highlights that testing should be automated as much as possible, which would contribute to intensify its adoption throughout the software development process. Moreover, testing tools are invaluable resources for research and education, as highlighted by Horgan and Mathur (1992). However, even with current developments in the area, there is still a long way towards the systematic, real adoption of testing by the industry.

Based on the above discussion, we conclude that the quality of the tests depends not only on the availability of appropriate testing techniques and criteria, but also on automated tools that support testing along the whole development process. Moreover, testing tools also facilitate the conduction of experimental studies that enables the technology evolution and transfer to the industry.

2.3 Testing of Aspect-Oriented Software

In spite of the claimed benefits that can be achieved with the adoption of AOP, such as enhanced software modularity and maintainability (Laddad, 2003a), it poses new challenges for the SQA activities. For example, AOP-specific constructs like PCDs and advices may represent new sources of faults within programs (Alexander et al., 2004), so may the implicit interactions between aspects and base modules (Burrows et al., 2010b). These new challenges have motivated the proposal of several approaches for the testing of AO software (hereafter called *AO testing*) in the last years.

In order to characterise the state of the art in AO testing, we have been regularly updating a systematic mapping study[5] of this topic. By the time this dissertation was written, we had performed five complete search cycles and performed all steps defined in the process for such kind of study, whose details and results are presented in the following sections.

2.3.1 The Systematic Mapping Study Protocol and Process

According to Petersen et al. (2008), a systematic mapping study (SMS) provides an overview of a research area by identifying and quantifying the related available research and results. A systematic literature review (SLR), on the other hand, is defined as a rigorous, well-established approach for identifying, evaluating and interpreting all available evidence in regard to a particular topic of interest (Kitchenham, 2004). In both cases, research gaps are identified and the pieces of work of interest are called *primary studies*[6]. Obviously, the SMS stakeholders can go further and also provide qualitative analysis based on the selected set of primary studies, however the absence of such kind of analysis should not invalidate the final results.

Our SMS on AO testing has two main goals: (i) identifying testing approaches that researchers and practitioners have been investigating for AO software; and (ii) identifying fault types that are specific to AO software. We defined the following primary research

[5] The term *"systematic mapping study"* (Petersen et al., 2008) has been recently used to characterise the kind of study we present in this section, due to the limited statistical meta-analysis that is allowed with the achieved results. The original *"systematic literature review"* (SLR) term (Kitchenham, 2004), on the other hand, used to replace *"systematic mapping study"*; however, SLRs are expected to go through existing primary reports and to employ in-depth analysis usually with significant statistical rigour (Petersen et al., 2008).

[6] The term *"primary study"* has so far been used in the Evidence-based Software Engineering domain (Kitchenham et al., 2004) to describe a variety of research results, from well-founded experimental procedures to incipient research approaches. SLRs and systematic maps, on the other hand, are treated as *"secondary studies"*. Note that a secondary study may also include other secondary studies of interest.

question (PQ1) and secondary questions (SQ1, SQ2 and SQ3). For each of them, associate inclusion and exclusion criteria were also defined.

- PQ1: *Which testing techniques/criteria have been applied to AO software to date?*
- SQ1: *Which of these techniques/criteria are specific to AO software?*
- SQ2: *Which AOP-specific fault types have been described to date?*
- SQ3: *Which kinds of experimental studies have been performed in order to validate AO software testing approaches?*

Defining an adequate search string is crucial for a SMS to succeed. It relies on the expertise of the involved researchers and should be as comprehensive as possible in order to match all primary studies of interest. According to our research interests, we defined the following string:

```
(aspect-oriented software OR aspect-oriented application OR
 aspect-oriented app OR aspect-oriented program OR aop) AND
(testing OR fault OR defect OR error OR  incorrect OR  failure)
```

The selected sources of primary studies range from indexed repositories (IEEE Xplore, ACM Digital Library, ScienceDirect and SpringerLink) to general purpose search engines (Google and Scirus). The general SMS procedures (i.e. preliminary selection, final selection, data extraction and documentation) followed the Biolchini et al.'s template for SLRs (Biolchini et al., 2005). We highlight that a major difference between a SMS and a SLR is that the search for primary studies in the former is usually broader than in the latter (i.e. a larger field can be characterised), while the in-depth focus of a SLR requires more refined, evidence-based search, thus posing more restrictions for the selection of primary studies (Petersen et al., 2008). Nonetheless, we find that the general procedures earlier suggested by Biolchini et al. can be legitimately applicable for SMSs since they do not conflict with the guidelines for SMS proposed by Petersen et al.

The general procedures are summarised as follows: the preliminary selection consists of the researchers going through specific parts of the primary studies retrieved from the repositories to check whether they should be selected for full reading. In the final selection step, the researchers fully read each primary study and make the final decision of selecting or discarding it, according to the inclusion and exclusion criteria. Data of interest is then extracted and stored in customised forms. During the whole process, documentation

tasks are undertaken in order to enable auditability and replicability (Biolchini et al., 2005; Kitchenham, 2004).

While planning our first update on the original iteration, we realised that we would need to adapt the original process proposed by Biolchini et al. (2005) in order to succeed. The adaptation resulted in the process shown at the bottom part of Figure 2.8 (Ferrari and Maldonado, 2008), whereas the original process is depicted in the top part of the same figure. The adaptation mainly concerned the *Planning Update*, *Filtering Results* and *Merging Results* activities, i.e. the three rounded boxes with white background in the adapted process. In short, the adapted activities focus on adjusting the selection criteria, creating filters (e.g. year of publication) to reduce overlapping during the preliminary selection and merging the extracted data into the original dataset. Note that the remaining activities are the same as defined in the original Biolchini et al.'s process and appear with grey background.

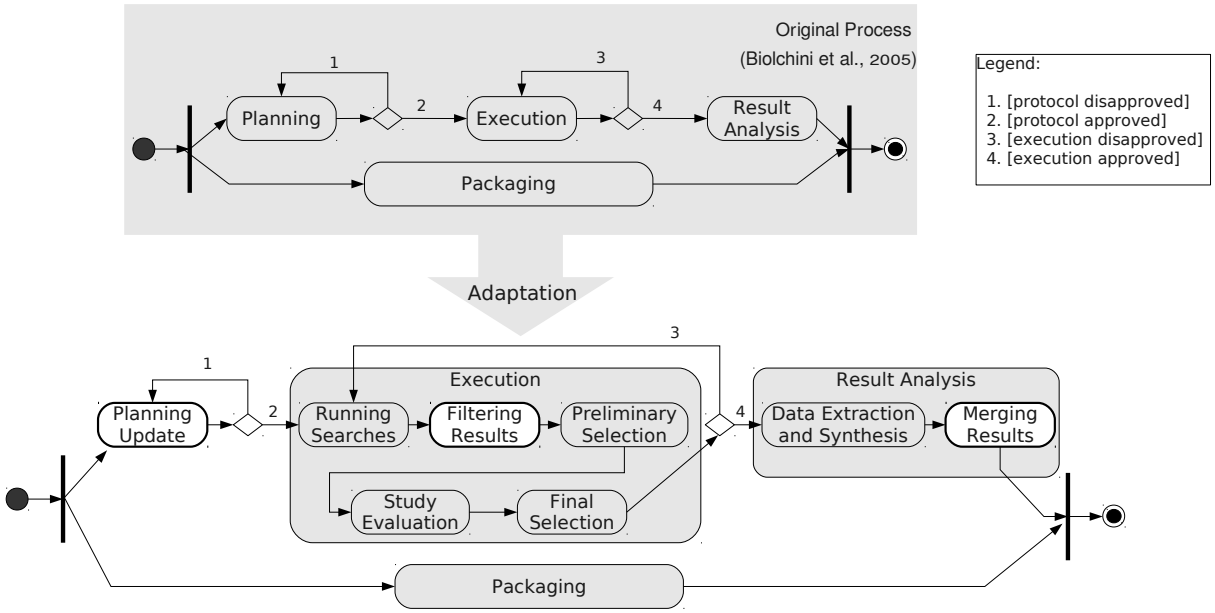


Figure 2.8: The process for systematic literature review updates – adapted from Ferrari and Maldonado (2008).

2.3.2 The Systematic Mapping Study Results

Table 2.4 summarises the final study selection after the five iterations of our SMS. It includes 34 primary studies. Note that our previous research has already documented partial results of the first four iterations (Ferrari et al., 2009; Ferrari and Maldonado,

2006, 2007). Each iteration took place with a medium interval of 12 months, having the original round taken place in July, 2006 (Ferrari and Maldonado, 2006).

According to Kitchenham (2004), the final set of selected studies should avoid overlapping of primary studies. For example, if we identify two or more pieces of work that describe the same testing approach (e.g. the latter as the evolution of the former), the final selection should include only the more recent one. This is handled during the *Merging Results* activity in the *Result Analysis* phase (see Figure 2.8). Taking this into consideration, Table 2.5 lists all overlapping results we identified along the five iterations. It results in a total of 53 studies of interest which, after the *Merging Results* step, turned to the final set of 34 items listed in Table 2.4.

We can observe that every selected study fits at least one of the research questions defined in Section 2.3.1. We highlight that even though we might consider that every testing approach which defines associated criteria should focus on at least one testing technique, Table 2.4 shows that this does not hold for all selected studies. For example, Alexander et al. (2004) and Ceccato et al. (2005) similarly define criteria that focus on testing all precedence order of concurring advices, which cannot be directly classified according to the traditional testing techniques. Nevertheless, these studies were assigned a *yes* in the "Define criteria" column.

The next sections describe the results of our SMS with focus on two of our research questions: the characterisation of fault types for AO software and the definition of AO testing approaches based on the traditional testing techniques. Such description represents relevant background for the context of this thesis and allowed us to identify the research opportunities which we explore in our research.

2.3.3 Fault Taxonomies for AO Software

As previously discussed in this chapter, the concepts and elements introduced by AOP represent new potential sources of software faults, hence posing new challenges for SQA activities such as testing and debugging. In general, software faults are artefacts that have been widely studied over the years. Among other types of study, some researchers have analysed how specific programming features can be sources of faults in software systems (Alexander et al., 2004; Coelho et al., 2008b; DeMillo and Mathur, 1995; Offutt et al., 2001). Others have empirically studied how different types of faults appear in the context of real software development projects (Basili and Perricone, 1984; Endress, 1978; Ostrand and Weyuker, 1984). Note that identifying potential sources of faults and characterising how they can occur in practice represent an important step towards the

Table 2.4: Systematic Mapping Study of AO testing: final selection.

Author(s)	Year	Round	Source	Testing technique(s)	Define criteria	Evaluation	Characterise fault types
Zhao	2003	1	IEEE	structural-based	no	n/a	no
Alexander et al.	2004	1	Google	n/a	yes	n/a	yes
Zhou et al.	2004	1	Google	n/a	yes	case study	no
Xu et al.	2004	1	Google	structural- and state models-based	no	n/a	no
Lemos et al.	2004a	1	Specialist	structural-based	yes	n/a	no
van Deursen et al.	2005	1	Google	functional- and structural-based	no	case study	yes
Ceccato et al.	2005	1	Google	n/a	yes	n/a	yes
Lesiecki	2005	1	Google	n/a	no	n/a	no
McEachen and Alexander	2005	1	ACM	n/a	no	n/a	yes
Badri et al.	2005	1	IEEE	state models-based	yes	n/a	no
Mortensen and Alexander	2005	1	Google	structural- and fault-based	yes	case study	no
Bækken	2006	2	Google	n/a	no	n/a	yes
Xu and Xu	2006a	1	ACM	state models-based	yes	n/a	yes
Xu and Xu	2006b	1	ACM	state models-based	no	case study	yes
Xie and Zhao	2006	1	ACM	structural- and state models-based	yes	case study	no
Lemos et al.	2006	1	Specialist	structural- and fault-based	no	n/a	yes
Mortensen et al.	2006	2	Google	structural-based	yes	n/a	yes
Eaddy et al.	2007	2	Google	n/a	no	n/a	yes
Zhang and Zhao	2007	2	Google	n/a	no	n/a	yes
Lemos et al.	2007	2	Elsevier	structural-based	yes	case study	no
Massicotte et al.	2007	2	Google	UML models-based	yes	n/a	yes
Xu and He	2007	2	ACM	UML models-based	yes	n/a	no
Anbalagan and Xie	2008	4	IEEE	fault-based	no	case study	no
Ferrari et al.	2008	3	IEEE	fault-based	yes	case study	yes
Coelho et al.	2008a	3	Springer	n/a	no	exploratory study	yes
Liu and Chang	2008	3	Google	state models-based	no	n/a	no
Bernardi and Di Lucca	2008	3	Google	structural-based	yes	case study	yes
Xu et al.	2008	4	Google	UML models-based	no	case study	no
Delamare et al.	2009a	4	IEEE	functional- and fault-based	no	case study	no
Babu and Krishnan	2009	4	ACM	UML models-based	no	n/a	yes
Kumar et al.	2009	4	ACM	n/a	no	n/a	yes
Lemos et al.	2009	4	Elsevier	structural-based	yes	case study	yes
Xu and Ding	2010	5	IEEE	state models-based	no	case study	yes
Lemos and Masiero	2010	5	Elsevier	structural-based	yes	case study	no

Table 2.5: Systematic Mapping Study of AO testing: overlapping results.

Final Selection	Round	Subsumed studies	Round
(Zhao, 2003)	1	(Zhao, 2002)	1
(Mortensen and Alexander, 2005)	1	(Mortensen and Alexander, 2004)	1
(Lemos et al., 2007)	2	(Lemos et al., 2004b)	1
		(Lemos et al., 2005)	1
(Xu and Xu, 2006a)	1	(Xu et al., 2005)	1
		(Xu et al., 2004)[1]	1
(Xie and Zhao, 2006)	1	(Xie et al., 2005)	1
(Massicotte et al., 2007)	2	(Massicotte et al., 2005)	1
		(Massicotte et al., 2006)	1
(Xu et al., 2008)	4	(Xu and Xu, 2005)	1
(Anbalagan and Xie, 2008)	4	(Anbalagan and Xie, 2006)	2
		(Anbalagan, 2006)	2
(Bækken, 2006)	2	(Bækken and Alexander, 2006b)	1
		(Bækken and Alexander, 2006a)	3
(Bernardi and Di Lucca, 2008)	3	(Bernardi and Lucca, 2007)	3
		(Bernardi, 2008)	3
(Lemos et al., 2009)	4	(Franchin et al., 2007)	3
(Coelho et al., 2008a)	3	(Coelho et al., 2008b)	4
(Lemos and Masiero, 2010)	5	(Lemos and Masiero, 2008a)	4
		(Lemos and Masiero, 2008b)	4
		(Lemos et al., 2006)[2]	2

[1] Xu et al.'s approach (2004) is only partially included by Xu and Xu (2006a) since the former also considers structural-based coverage measurements.

[2] Lemos et al.'s approach (2006) is only partially included by Lemos and Masiero (2010) since the former also considers mutation-based coverage measurements.

definition of fault models. Such fault models, in turn, may underlie the establishment of testing strategies that can be applied in particular software application domains.

Specifically for AOP, we identified a number of candidate fault taxonomies and bug pattern catalogues for AO software (Alexander et al., 2004; Bækken, 2006; Ceccato et al., 2005; Coelho et al., 2008a; Eaddy et al., 2007; Ferrari et al., 2008; Lemos et al., 2006; van Deursen et al., 2005; Zhang and Zhao, 2007). Note that they typically rely on programming features and languages constructs (Ferrari et al., 2008), however they still require empirical evaluation based on real software development data. Moreover, these candidate fault taxonomies are described in varied levels of details, ranging from coarse-grained characterisation (Alexander et al., 2004; Ceccato et al., 2005) to construct-specific fault type definitions (Bækken, 2006; Coelho et al., 2008a; Lemos et al., 2006).

The first initiative to define a candidate fault taxonomy for AO software was presented by Alexander et al. (2004). Initially, the authors identified possible sources of faults in AO programs. For instance, a fault may reside in a portion of the base program not affected by an aspect, or it may be related to an emergent property created by aspect-base

program interactions. Based on these sources, they proposed a high-level fault taxonomy for AspectJ-like programs that includes six types of faults: (i) incorrect strength in PCD patterns; (ii) incorrect aspect precedence; (iii) failure to establish post-conditions; (iv) failure to preserve state invariants; (v) incorrect focus on control flow; and (vi) incorrect changes in control dependencies. Alexander et al. exemplify how these fault types can occur based on a simple AspectJ application; besides, they propose high-level testing criteria to address them.

Several refinements and amendments to Alexander et al.'s taxonomy have been proposed ever since. For example, Ceccato et al. (2005) added three new fault type descriptions: (i) incorrect changes in exceptional control flow; (ii) failures due to intertype declarations; and (iii) incorrect changes in polymorphic calls. Besides that, the authors discussed how hard testing AO code is when compared to testing OO code. They suggest some adaptations to existing OO-based testing approaches to make them applicable to AO software as well, particularly when the fault types described in their taxonomy are taken into account.

Lemos et al. (2006) refined the *incorrect strength in PCD patterns* fault type into four categories of incorrect PCD: (i) selection of a superset of intended join points; (ii) selection of a subset of intended join points; (iii) selection of a wrong set of join points, which includes both intended and unintended items; and (iv) selection of a wrong set of join points, which includes only unintended items. This four PCD-related fault types are graphically represented in Figure 2.9.

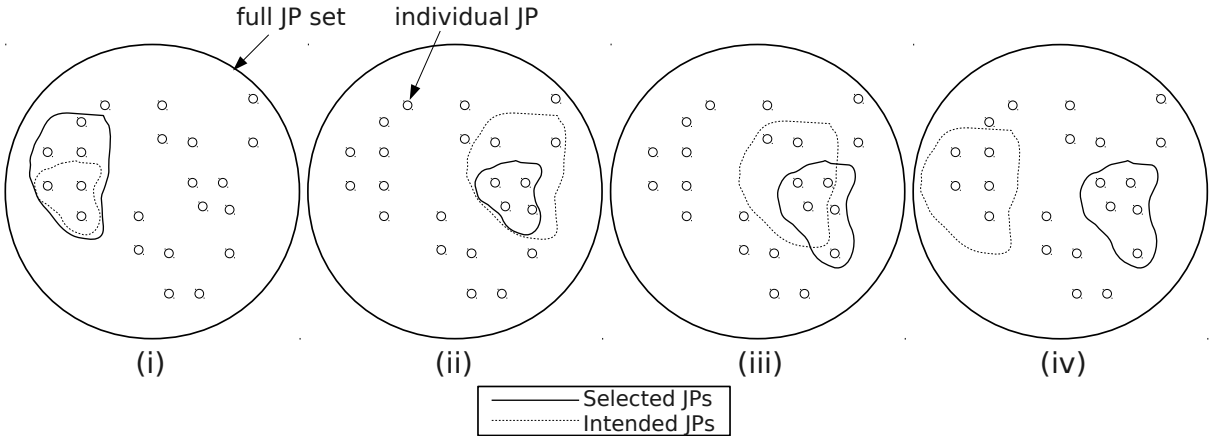


Figure 2.9: PCD-related fault types – adapted from Lemos et al. (2006).

Bækken (2006) proposed a fine-grained fault taxonomy for PCDs and advices in AspectJ programs. The author also presented a set of examples of how a faulty PCD or

advice may affect control and data dependencies in a program execution. According to his analysis of the AspectJ language, Bækken identified three kinds of errors that may result from the execution of PCD faults: (i) positive selection error, which consists of the selection of unintended join points; (ii) negative selection error, which occurs when intended join points are not picked up; and (iii) context exposure error, which occurs when a formal parameter on the left-hand side of a PCD or in an advice definition has an incorrect value, i.e., the formal parameter is bound to an incorrect variable reference or to a variable that has an incorrect value. For advice faults, on the other hand, errors are more directly related to faults that affect control and data dependencies. When some faulty statement is executed, it may result in an infection which may propagate to the output due to a change in control or data flow dependencies.

Other authors (McEachen and Alexander, 2005; van Deursen et al., 2005; Zhang and Zhao, 2007) also defined AOP-specific fault types which partially overlap Alexander et al.'s taxonomy, although also characterising additional fault types. More recently, Coelho et al. (2008a) performed an exploratory study of the impact of aspects on the flow of exceptions in AO systems. In addition, they introduced a catalogue of nine bug patterns for exception handling in AspectJ. The bug patterns are classified into three categories, according to the role aspects play in exceptional scenarios: (i) aspects as handlers; (ii) aspects as signallers; and (iii) aspects as exception softeners. All bug patterns have representatives in at least one of the AO systems subject of the study.

2.3.4 AO Testing Approaches

This section describes the AO testing approaches that have shown substantial evolution along the years. A general discussion about the evolution of AO testing and the collaboration amongst researchers was presented in previous research (Ferrari et al., 2009). The evolving approaches we next present have been documented by the authors as series of papers and articles published in relevant scientific vehicles such as international conferences and workshops as well as high quality journals.

2.3.4.1 Structural-based Testing by Lemos et al.

Unit Testing Level

Lemos et al. (2005, 2004b, 2007) developed a structural-based unit testing approach for AspectJ programs. It extends previous work undertaken within the authors' research group (Vincenzi et al., 2006a) in order to enable unit testing (methods and advices) in

isolation. The approach relies on Java bytecode analysis, thus allowing the evaluation of applications without requiring the source code.

In this unit testing approach, Lemos et al. defined the AODU (*Aspect-Oriented Def-Use*) graph for each unit that belongs to the modules under test (classes and aspects, in this case). The AODU graph extends the traditional DUG (*Def-Use Graph*) to include the *crosscutting nodes*, which consist in graph nodes that encompass control flow and data flow information about the advised join points in the base code. Given a unit under test (e.g. a class method), the crosscutting nodes indicate the join points of that unit that are affected by advices.

An example of an AODU graph is depicted in Figure 2.10. The top left part partially lists the code[7] of a Java class named `Call` and an AspectJ aspect named `Billing` that crosscuts `Call`. The AODU graph of the `Call`'s constructor method is depicted in the bottom of Figure 2.10, while the respective set of Java bytecode instructions for this constructor is listed on the right-hand side. The numbers displayed in some lines of the `Call` constructor's Java code (i.e. "4", "33-72" and so on) refer to the node labels in the graph. We can observe that the AODU is composed by the control flow structure (i.e. nodes and edges) as well data flow information. In particular, the crosscutting nodes are depicted in a customised notation: a dashed ellipse tagged with information that regards the advice that affects that point. Besides that, the uses of variables within these nodes are computed (e.g. `caller`, `receiver` and `IM` variables in nodes 33 and 78).

Based on the AODU graph as well as on the traditional All-nodes, All-edges and All-uses criteria described in Section 2.2, Lemos et al. proposed the following control and data flow-based criteria:

- ***all-crosscutting-nodes*** (All-nodes$_c$): requires that each crosscutting node of the AODU of a given unit be exercised at least once by the test set.
- ***all-crosscutting-edges*** (All-edges$_c$): requires that each edge that includes a crosscutting node in the AODU of a given unit be exercised at least once by the test set.
- ***all-crosscutting-uses*** (All-uses$_c$): requires that each def-use association for which the use occurs within a crosscutting node be exercised at least once by the test set.

For instance, based on the example presented in Figure 2.10, the All-edges$_c$ criterion requires a test set that exercises the edges (4,33) and (4,78) of the graph in order to be considered adequate with respect to this criterion for this method.

[7]This example was extracted from a system that simulates telephone calls named Telecom, which comes along with the AspectJ distribution (The Eclipse Foundation, 2010b).

```
       public class Call {
         private Customer caller, receiver;
         private Vector connections = new Vector();

0        public Call(Customer caller,
                     Customer receiver, boolean iM) {

4          this.caller = caller;
4          this.receiver = receiver;
4          Connection c;
4          if (receiver.localTo(caller)) {
33-72        c = new Local(caller, receiver, iM);
           } else {
78-117       c = new LongDistance(caller,receiver,iM);
           }
120        connections.addElement(c);
120      }
         ...
       } // end class


       public aspect Billing {
         private Customer Connection.payer;

         pointcut createConnection(Customer caller,
                  Customer receiver, boolean iM) :
              args(caller, receiver, iM) &&
              call(Connection+.new(..));

         after(Customer caller, Customer receiver,
                 boolean iM) returning (Connection c):
              createConnection(caller,receiver,iM) {

           if (receiver.getPhoneNumber().
                                  indexOf("0800")==0)
             c.payer = receiver;
           else
             c.payer = caller;
           c.payer.numPayingCalls += 1;
         }
         ...
       } // end aspect
```

```
 0 aload_0
 1 invokespecial #15 <Method Object()>
 4 aload_0
 ...
27 invokevirtual #30 <Method boolean
   localTo(telecom.Customer)>
30 ifeq 78
33 aload_1
...
52 invokespecial #34 <Method
   Local(telecom.Customer,
   telecom.Customer, boolean)>
...
69 invokevirtual #110 <Method void
   ajc$afterReturning$telecom_Billing$
   1$8a338795(
   telecom.Customer, telecom.Customer,
   boolean, telecom.Connection)>
72 nop
73 astore 4
75 goto 120
78 aload_1
...
97 invokespecial #37 <Method
   LongDistance(telecom.Customer,
   telecom.Customer, boolean)>
...
114 invokevirtual #110 <Method void
   ajc$afterReturning$telecom_Billing$
   1$8a338795(
   telecom.Customer, telecom.Customer,
   boolean, telecom.Connection)>
117 nop
118 astore 4
120 aload_0
121 getfield #20 <Field java.util.
   Vector connections>
124 aload 4
126 invokevirtual #41 <Method void
   addElement(java.lang.Object)>
129 return
```

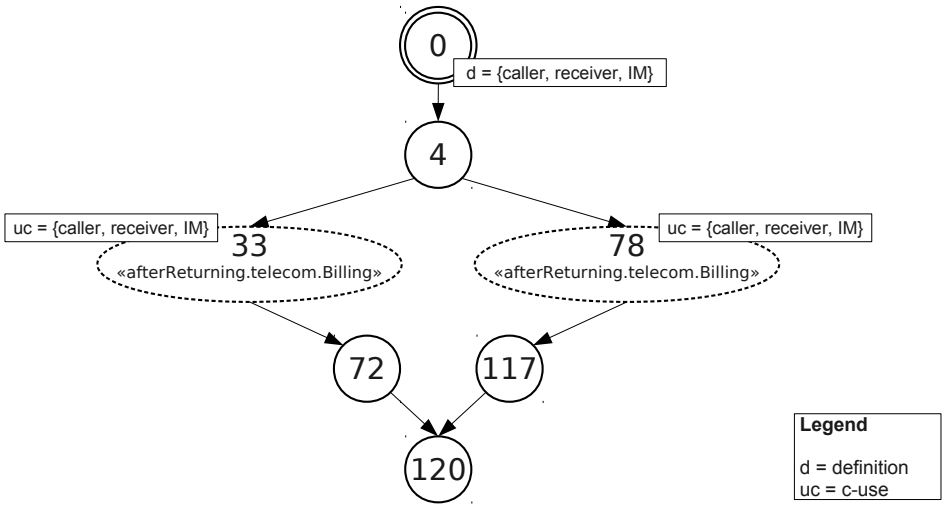


Figure 2.10: Example of an AODU graph – adapted from Lemos et al. (2007).

Pairwise- and Pointcut-based Integration Testing Levels

Franchin et al. (2007) and Lemos et al. (2009) extended their previous approach for unit testing (Lemos et al., 2007) in order to support testing at the integration level. In short, the extension consists in a pairwise-based approach in which the AODU graphs of two communicating units are combined into a single graph named PWDU (*PairWise Def-Use*). Similarly to the unit testing approach, it also relies on Java bytecode analysis to enable the graph generation.

The left-hand side of Figure 2.11 brings an example of a PWDU. The graph represents the pairwise integration of two units presented in Figure 2.10: the `Call` class' constructor method and the `after returning` advice listed for the `Billing` aspect. In fact, this advice crosscuts the constructor in two join points, where instances of `Local` and `LongDistance` classes are created. Figure 2.11 depicts the integration when the first join point is reached, i.e. when a `Local` instance is created.

Note that the PWDU graph also includes double-circled nodes to represent the call sites (e.g. method and advice calls). Besides, the prefix "i" is used to distinguish between nodes in the integrated graph. In this case, nodes prefixed with i belong to the *integrated unit* (i.e. the `after returning` advice in the example), while the non-prefixed nodes belong to the *base unit* (i.e. the class' constructor method in the same example).

Franchin et al. (2007) and Lemos et al. (2009) defined a set of control flow- and data flow-based testing criteria based on the PWDU graph which are described as follows:

- ***all-pairwise-integrated-nodes*** (All-PW-Nodes$_i$): requires that each integrated node in a PWDU graph be exercised at least once by the test set.

- ***all-pairwise-integrated-edges*** (All-PW-Edges$_i$): requires that each integrated edge of a PWDU graph be exercised at least once by the test set.

- ***all-pairwise-integrated-uses*** (All-PW-Uses$_i$): in a PWDU, it requires that each def-use association with respect to a communication variable, and whose definition occurs within the base unit while the use happens in the integrated unit and vice versa, be exercised at least once by the test set.

Note that the All-PW-Uses$_i$ criterion focuses on the set of *communication variables*, which is composed by variables that are related to the interface of the units under test (e.g. a method parameter or a global variable). Therefore, this criterion reinforces the importance of testing the integration between interacting units, after internal elements have been tested in the unit phase.

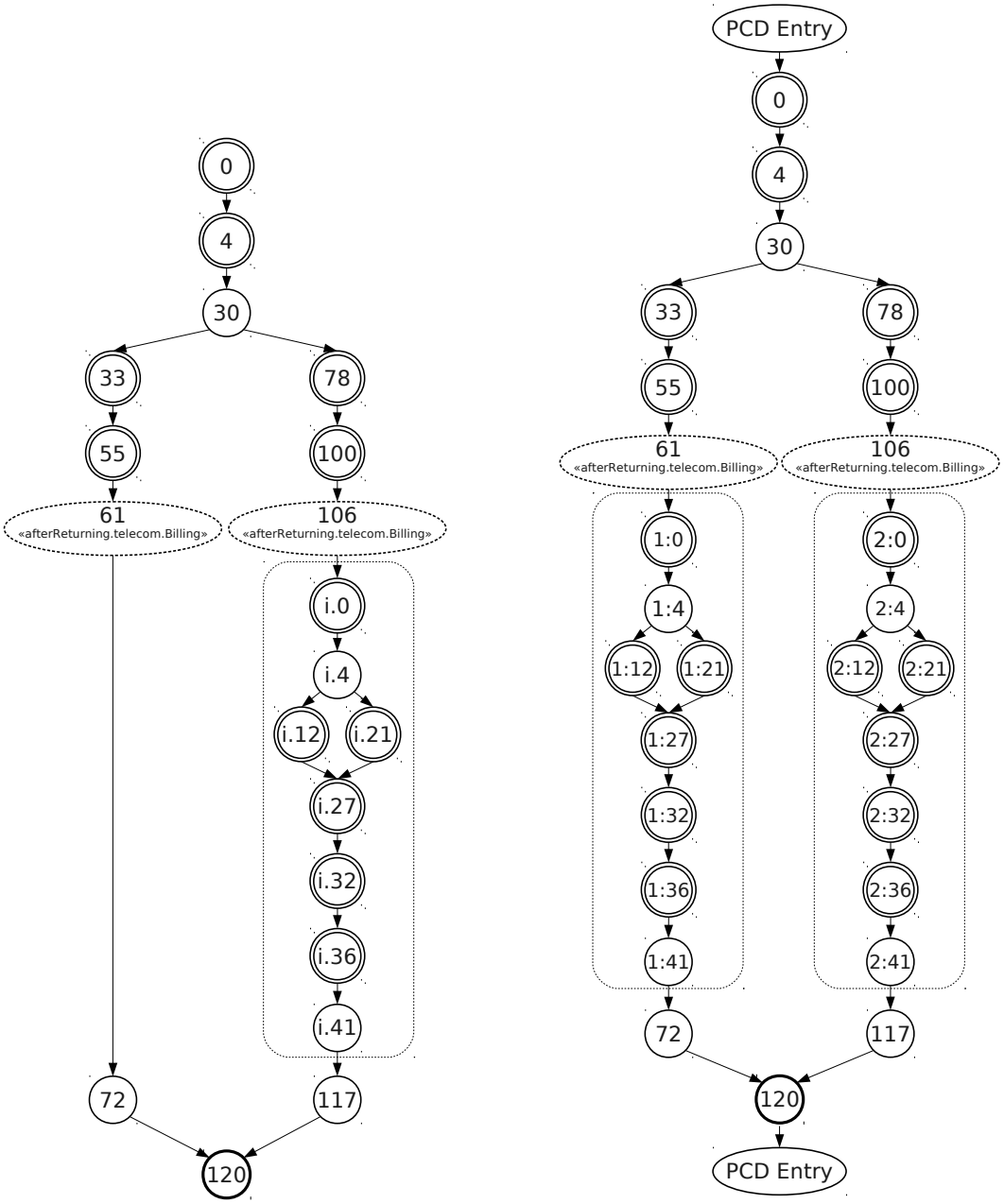


Figure 2.11: Examples of PWDU (Lemos et al., 2009) and PCDU (Lemos and Masiero, 2010) graphs.

Finally, Lemos and Masiero (2010) proposed another extension to their original unit testing approach (Lemos et al., 2007). At this time, the PCDU (*PointCut-based Def-Use*) graph is built in order to represent the whole execution context for a given piece of advice. This new graph is composed by all AODU graphs of the units affected by an advice together with the AODU of the advice repeated at each join point of the affected units. This model

provides the tester with an overall view of the advice application context and supports the design of a test set that addresses such whole context.

The right-hand side of Figure 2.11 brings an example of a PCDU. It includes the composition of the AODU graph of `after returning` advice listed in Figure 2.10 with the AODU graphs of the affected unit. Note that the AODU graph of the `Call` class' constructor method is included twice since this unit is affected in two different join points.

In a PCDU graph, numbered prefixes are used to label nodes that are owned by the advice graph and the graphs of the affected units. In the example presented in Figure 2.11, the dashed regions highlight the AODU graph of the `after returning` advice, whose nodes are prefixed with "1" and "2" since this advice crosscuts the method in two distinct join points. The remaining nodes represent the AODU graphs of the affected units (in this example, a single unit is affected in two distinct join points).

Based on the PCDU graph, Lemos and Masiero (2010) defined three additional criteria which are described as follows:

- ***all-pointcut-based-advice-nodes*** (all-pc-nodes): requires that each node that belongs to the advice in the AODU graph be exercised at least once by the test set, for all occurrences of this advice in the AODU graph.

- ***all-pointcut-based-advice-edges*** (all-pc-edges): requires that each edge that belongs to the advice in the AODU graph be exercised at least once by the test set, for all occurrences of this advice in the AODU graph.

- ***all-pointcut-based-uses*** (all-pc-uses): in a PCDU, it requires that each def-use association with respect to a communication variable, and whose definition occurs within the affected unit while the use happens in the advice and vice versa, be exercised at least once by the test set.

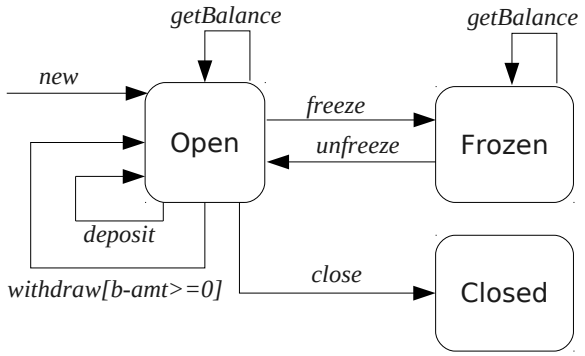
Note that the control flow-related criteria focus on testing the advice logic instead of the whole set of PCDU graph elements. The data flow-based criterion, on the other hand, focuses on communication variables, similarly to the integration testing approach earlier proposed by Lemos et al. (2009).

2.3.4.2 State Models-based Testing by Xu and Xu

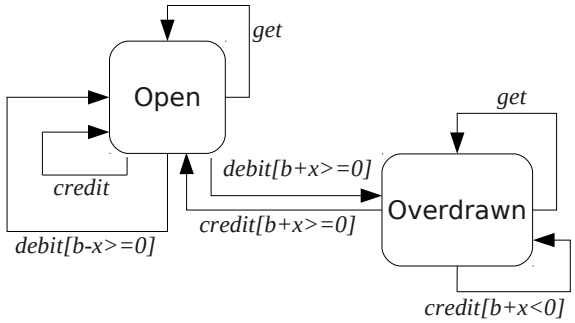
Xu and Xu (2006a) formally defined a state model for AO systems which includes states that originate from base application and also takes into account the possible modifications of the base state model (i.e. modified and added states) that result from aspectual

behaviour. The FREE (*Flattened Regular Expression*) model (Binder, 1999, p. 204-228) is used to define the *Aspectual State Model* (ASM) proposed by Xu and Xu, who defined a customised notation to represent the state transitions that can be used to derive the test requirements.

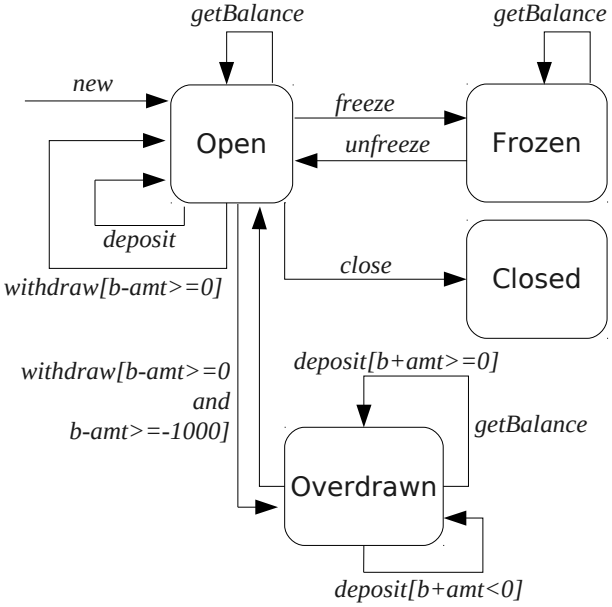
To demonstrate their approach, the authors utilise a banking management system, which is partially depicted in Figure 2.12. This figure includes: (a) the state model of the `BankAccount` class; (b) the state model that results from the `Overdraft` aspect affecting the `BankAccount` behaviour, hence yielding the additional `Overdrawn` state; and (c) the final ASM, which represents the woven state model that includes all states.



(a) State model of the `BankAccount` class.



(b) Impact of the `Overdraft` aspect on `BankAccount`.



(c) ASM of `BankAccount` combined with `Overdraft`.

Figure 2.12: Example of an ASM before (a) and after (b, c) weaving the aspects into the base application (Xu and Xu, 2006a).

A variable mapping schema that involves methods and PCDs is employed in the ASM generation. Such schema encompasses contextual information that regards the method and PCD signatures and is used to map variables amongst the several partial state models that compose the ASM. In the example presented in Figure 2.12, we can observe that a PCD named `credit` (Figure 2.12(b)) is mapped to the method named `deposit` of `BankAccount` (Figure 2.12(a)). Another example regards the `amt` variable (Figure 2.12(a)) which is mapped to the `x` variable in Figure 2.12(b).

Once the ASM is created, formal expressions that represent state sequence transitions can be used to derive the test requirements. Note that these requirements consist of state transition paths and should include both valid and invalid sequences. As an example of a negative sequence, let us consider the following expression:

```
<new, Open, withdraw[b-amt<-1000], Open>
```

This negative sequence can be realised through the following test case description: "A new account is created so there is an attempt to perform a withdraw that would result in a balance lower than -1000, however the account status would be kept as `Open`". In this case, after the withdraw, the "`b-amt<-1000`" condition holds and the account state should change to `Overdrawn`, differently from the negative test case description.

Xu and Xu proposed an 8-step procedure for test case generation based on the ASM. It starts from the base state model (i.e. only base classes are considered) then evolves towards the full ASM. In all cases, test cases are created to traverse a state transition tree that is derived from the ASM. An associated criterion requires both positive and negative paths, and can be complemented with additional criteria (e.g. Equivalence Partitioning and Boundary-Value Analysis).

2.3.4.3 Mutation-based Testing by Anbalagan and Xie

Anbalagan and Xie (2008) implemented a framework that generates and automatically detects equivalent mutant PCDs. The framework implements two mutation operators proposed by Mortensen and Alexander (2005): PCS (*Pointcut Strengthening*) and PCW (*Pointcut Weakening*). While the former narrows the PCD scope, the latter leads to a broader join point selection by the PCD.

In the Anbalagan and Xie's framework, mutants are generated in two different ways: (i) by inserting (removing) AspectJ wildcards into (from) the PCD; and (ii) by identifying and reusing naming parts of the original PCD and join points in the base code. Initially, the framework scans the base code in order to identify every possible candidate join

point. This task is supported by a third-party framework named *AJTE* (Yamazaki et al., 2005), which enables the evaluation of AspectJ programs before the weaving process. *AJTE* provides an API that allows one to represent PCDs and join points as regular Java objects. Consequently, information can be extracted from these objects, for example, the individual naming parts of a PCD or a method signature.

The next step is the generation of the mutants according to the two aforementioned ways: by inserting/removing wildcards into/from the original PCD and by building new PCDs that are composed by the naming parts identified by the *AJTE* framework. An example of the first case is shown in Figure 2.13. It includes mutants that can be produced by the PCW operator through the insertion of wildcards. We can see that mutants are exhaustively produced, for example, by replacing parts of the method pattern (`Connection.new` in the figure) with wildcards.

Original PCD:

```
pointcut createConnection(Customer caller, Customer receiver, boolean iM) :
      args(caller, receiver, iM) && call(Connection+.new(..));
```

Mutants:

```
pointcut createConnection(Customer caller, Customer receiver, boolean iM) :
     args(caller, receiver, iM) && call(*onnection+.new(..));
pointcut createConnection(Customer caller, Customer receiver, boolean iM) :
     args(caller, receiver, iM) && call(*nnection+.new(..));
pointcut createConnection(Customer caller, Customer receiver, boolean iM) :
     args(caller, receiver, iM) && call(*nection+.new(..));
pointcut createConnection(Customer caller, Customer receiver, boolean iM) :
     args(caller, receiver, iM) && call(*ection+.new(..));
pointcut createConnection(Customer caller, Customer receiver, boolean iM) :
     args(caller, receiver, iM) && call(*ction+.new(..));
pointcut createConnection(Customer caller, Customer receiver, boolean iM) :
     args(caller, receiver, iM) && call(*tion+.new(..));
// ... and so on
```

Figure 2.13: Example of mutants generated by Anbalagan and Xie (2008)'s framework.

The next step supported by the framework is the automatic identification of equivalent mutants. If a mutant PCD selects the same set of join point as does the original expression, it is automatically classified as equivalent. At this stage, the framework applies some heuristics that automatically rank the most representative mutant PCDs. If two or more mutants select the same set of join points, the one that more closely resembles the original PCD is lifted to the top of the rank. This "similarity relationship" relies on lexicographical analyses of the PCDs, given that the authors argue that effective mutant PCDs should resemble closely to the original PCD. The final output is a list of the ranked mutants.

2.3.5 Tool Support for AO Testing

In spite of the large number of AO testing approaches we identified in the systematic mapping study, they still lack adequate tool support. Several factors might contribute for this, from which we can highlight: (i) the high cost to develop robust testing tools; and (ii) the claimed benefits of AOP which are still seem with scepticism by practitioners, hence resulting in the cautious adoption of AOP by the industry (Muñoz et al., 2009). However, this sounds paradoxical given the importance of testing tools as teaching and research instruments, consequently for the real adoption of a newly introduced technology (Horgan and Mathur, 1992).

This section describes tools which automate testing approaches that rely on three different techniques: mixed state- and structural-based testing (Xie and Zhao, 2006), structural-based testing (Lemos et al., 2009; Lemos and Masiero, 2010; Lemos et al., 2007) and fault-based testing (Delamare et al., 2009b). Our choice for describing these tools was motivated by two main reasons: (i) they have been used in the most robust case studies amongst the ones identified in our systematic mapping study (Delamare et al., 2009a; Lemos et al., 2009; Lemos and Masiero, 2010; Xie and Zhao, 2006); and (ii) in general, they support the basic testing steps: derivation of test requirements, test execution and results reporting.

2.3.5.1 The Aspectra Framework for State- and Structural-based Test Generation

Xie and Zhao (2006) developed a framework called *Aspectra* that automatically generates test cases for AspectJ programs. In short, test cases are created based on state variables – i.e. class attributes – and their quality is evaluated by *Aspectra* according to two coverage measures defined by the authors. The framework classifies the execution of four types of methods within an AspectJ program: advised methods from the base modules, advices, intertype methods and public methods defined in the aspects. *Aspectra* focuses on testing the last three types, i.e. it targets behaviour that is implemented within aspects.

Aspectra implements a wrapper synthesis strategy that leverages existing tools for test case generation such as *JTest* (Parasoft, 2010) and *Rostra* (Xie et al., 2004), which produce standard *JUnit* tests that are fed to the framework. The process of test generation is depicted in Figure 2.14. In step 1, the ajc compiler weaves together the base classes and aspects and produces the respective bytecode classes. Then, in step 2 wrapper classes are created for the public and intertype methods in the base classes as well as for public, non-advice methods in the aspect classes. Advices, on the other hand, are indirectly exercised through the advised methods. In the sequence, everything (i.e. base classes,

aspects and wrappers) are woven together again in step 3 in order to ensure `call` join points are also taken into account by the test generation mechanism. Step 4 cleans-up unwanted bytecode in the wrapper classes (e.g. duplicated advice invocation due to the repeated weaving step). The clean wrapper classes are provided to test generation tools (step 5), which produce *JUnit* tests that are compiled using a standard Java compiler in step 6.

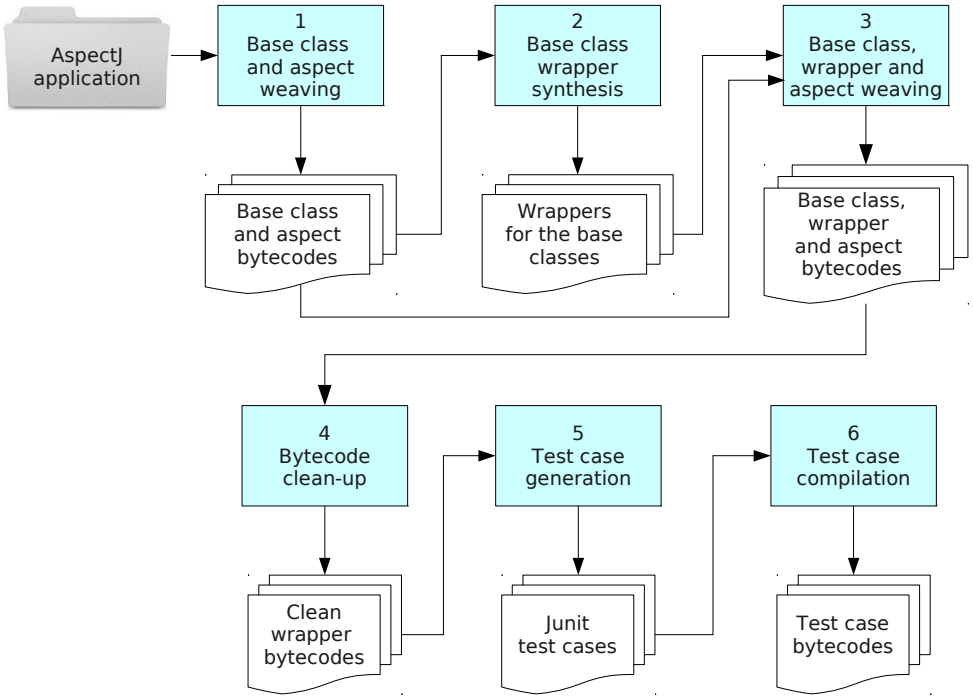


Figure 2.14: The test generation process implemented in the *Aspectra* framework.

The test case generation (step 5 in Figure 2.14) is based on the possible current object states as well as on the list of arguments that are passed to a method. The *JTest* tool uses symbolic execution to generate method arguments that will achieve structural coverage. In the sequence, the *Rostra* tool uses these values and, based on combinatorial testing, explores the state space of the receiver objects.

The two coverage criteria proposed by Xie and Zhao (2006) are *Aspectual Branch Coverage* and *Interaction Coverage*. The former requires that all branches within the aspect be exercised by the generated test set. The measurement occurs at bytecode level, therefore covering a single `around` advice[8] does not ensure all places in which this advice is activated are covered by the test set. The Interaction Coverage criterion increases the

[8]The weaving strategy implemented in the AspectJ `ajc` compiler allows the `around` advice code to be inlined in the selected join points; therefore, at the bytecode level multiple copies of the advice may appear across the woven application.

confidence on the test set given it requires that all interactions between aspect methods and advised methods be exercised by the test set. As the final output of the test execution, *Aspectra* reports the percentage of covered branches and aspect-class interactions.

2.3.5.2 The JaBUTi/AJ Series of Tools for Structural-based Testing

Lemos et al. implemented a series of tools named *JaBUTi/AJ* to support their evolving structural-based testing approach (Franchin et al., 2007; Lemos et al., 2009; Lemos and Masiero, 2010; Lemos et al., 2007). It consists of a sequence of extensions to the *JaBUTi* tool (Vincenzi et al., 2003), which was originally developed to support the evaluation of Java programs.

In general, all versions of *JaBUTi/AJ* support the several steps of a typical test session: (1) creating a test project, which includes instrumenting the program under test and computing the test requirements; (2) importing and executing test cases; and (3) calculating the test coverage. This process is depicted in Figure 2.15. It is important to highlight that *JaBUTi/AJ* computes all test requirements based on the Java bytecode information. Thus, it allows one to evaluate the test coverage obtained by a test set even if the source code of the program under test is not available during that stage.

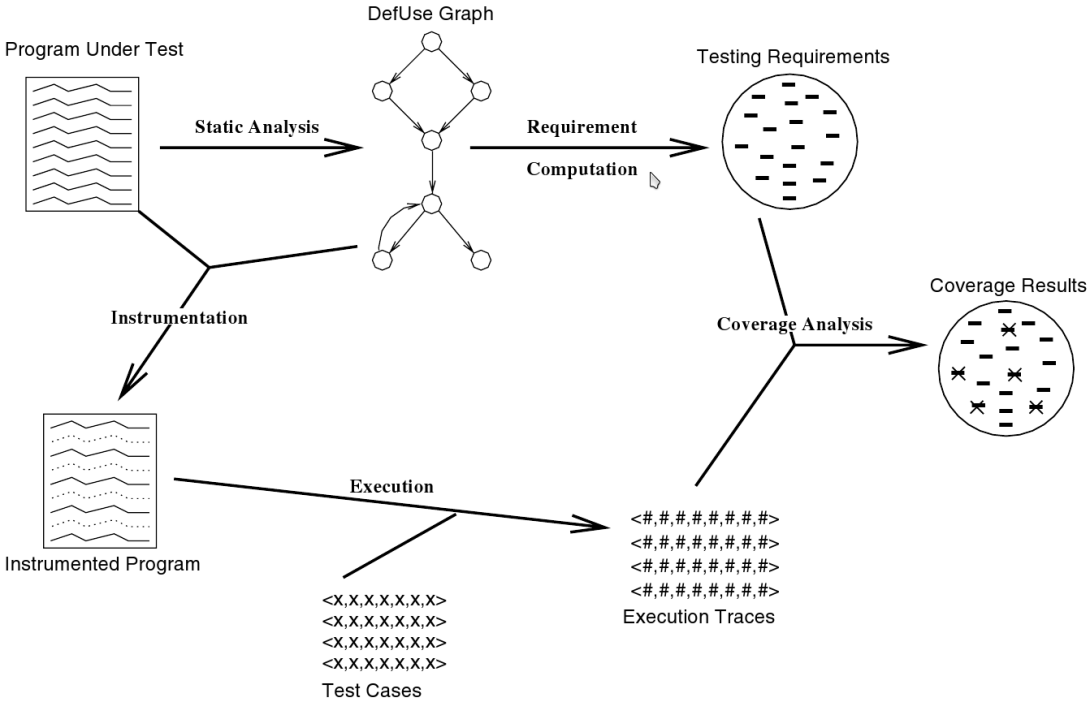


Figure 2.15: The test process supported by *JaBUTi/AJ*– adapted from Vincenzi et al. (2005) and Lemos et al. (2007).

The first version of *JaBUTi/AJ* supports the unit testing approach proposed by Lemos et al. (2007). Figure 2.16 shows a screenshot taken from the *JaBUTi/AJ*'s graphical interface. It represents an example of an AODU graph produced by the tool. The reader can notice that this is the AODU graph of the `Call` class' constructor method, which was earlier presented in Section 2.3.4.1 (Figure 2.10). The information displayed in the squared box regards the definition and use of variables in a pointed node of the graph (e.g. the node labelled with "78" as depicted in Figure 2.16).

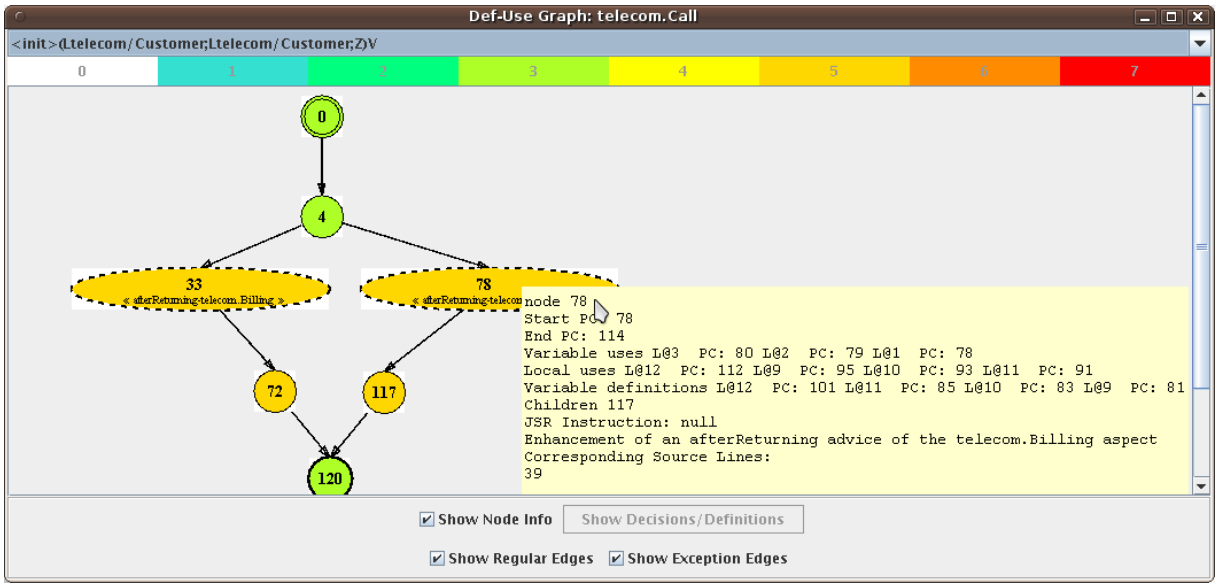


Figure 2.16: An AODU graph generated by *JaBUTi/AJ*.

Both *JaBUTi* and its extensions *JaBUTi/AJ* provide the tester with hints about which test requirements are harder to be fulfilled. This is achieved through the application of dominator and super block analysis (Agrawal, 1994), which may speed-up the definition of adequate test sets with respect to a given criterion. In short, the dominator and super block analysis enables the definition of weights for test requirements such that covering requirements with higher weights will possibly result in covering several other requirements with lower weights. In *JaBUTi/AJ*, different colours are used to represent the weight of the requirements, as we can see in Figure 2.16.

Figure 2.17 shows as example of a coverage report produced by the tool. In this example, the *all-crosscutting-nodes* (All-Nodes-c) is selected, which results in two test requirements for the `Billing` aspect and five requirements for the `Call` class. As long as *JUnit* tests are imported into the current test project, the tester can keep track of the increases in the test coverage.

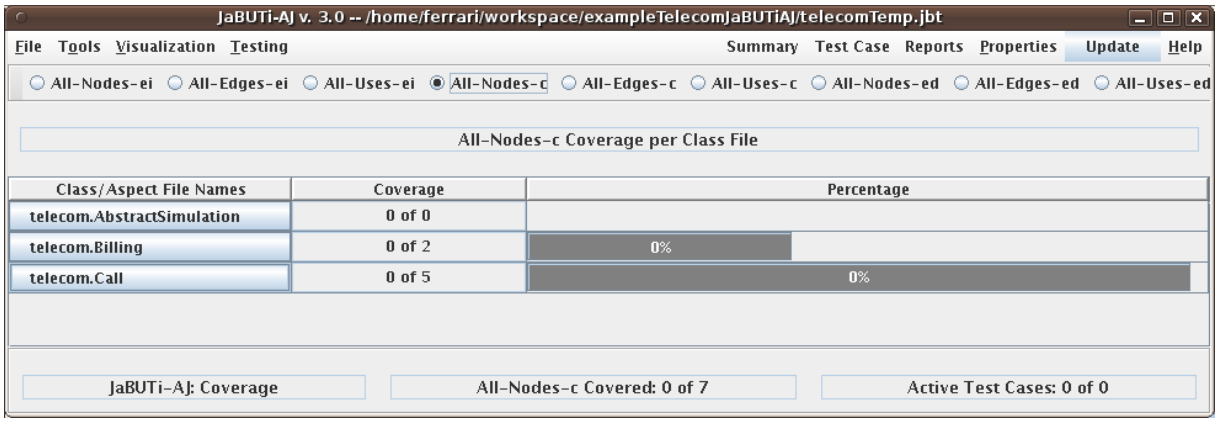


Figure 2.17: A *JaBUTi/AJ* coverage report screen.

Other *JaBUTi/AJ* extensions (Franchin et al., 2007; Lemos and Masiero, 2008a) support the pairwise- and pointcut-based testing approaches proposed by the authors. For example, Figure 2.17 shows a *JaBUTi/AJ* screen that lists the pairwise relationship between some modules that belong to an application under test. It allows the tester to select the pairs of interest, then the tool derives the test requirements based on the implemented testing criteria (see Section 2.3.4.1 for more details about the proposed criteria).

The most recent *JaBUTi/AJ* updates support the structural testing of AspectJ programs at multiple integration levels (Cafeo and Masiero, 2010; Neves et al., 2009) . They automate the creation of composed AODU graphs for a given unit and all units that are directly (Neves et al., 2009) or indirectly (Cafeo and Masiero, 2010) invoked by it. Based on these graphs, test requirements are derived according to a series of structural-based integration testing criteria.

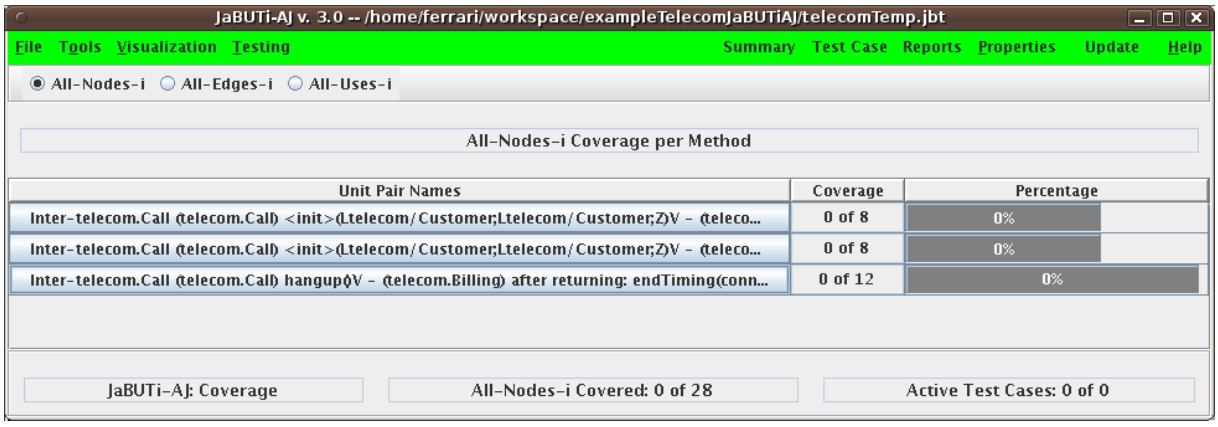


Figure 2.18: A *JaBUTi/AJ* pairwise-based test requirement selection screen.

2.3.5.3 The AjMutator Tool for Mutation Testing

Delamare et al. (2009b) developed a tool named *AjMutator* that automates the mutation testing of AspectJ programs. It focuses on the mutation of PCDs, and supports the generation of mutants, automatic identification of equivalent mutants, and test execution and reporting.

The *AjMutator*'s functionalities and execution flow are depicted in Figure 2.19. The tool initially parses PCDs from aspects individually and performs the mutations over their abstract syntax trees. The modified expressions are reinserted into the code, generating the mutants in the pretty-printing step. The mutants are compiled with the abc compiler (Avgustinov et al., 2005), an alternative compiler for AspectJ programs, and then stored into JAR[9] files.

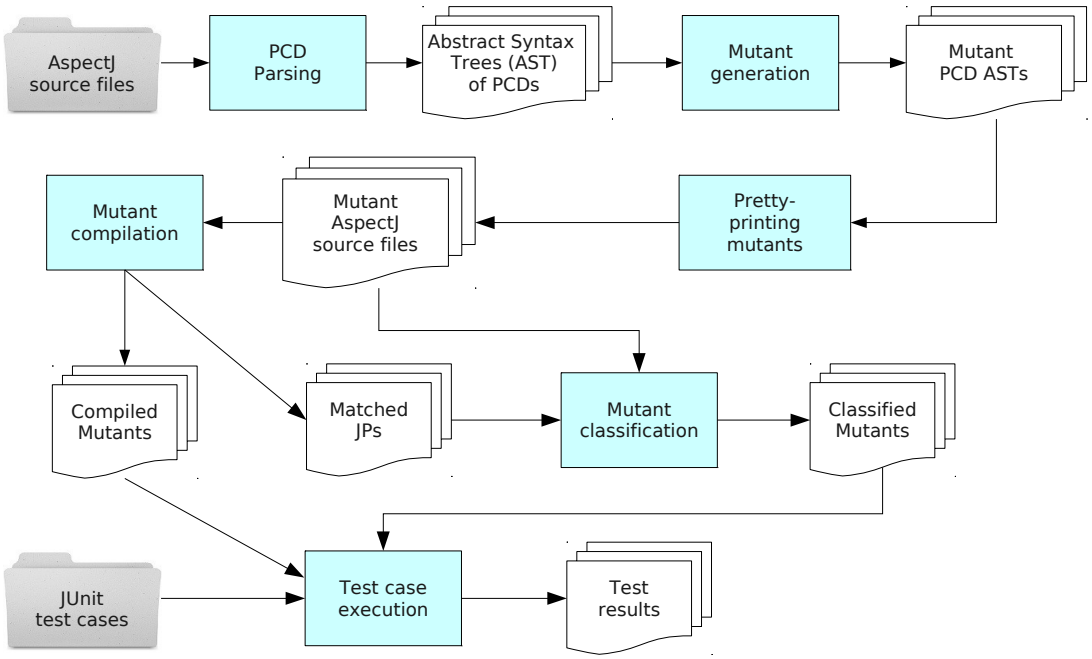


Figure 2.19: The test process supported by *AjMutator*– adapted from Delamare et al. (2009b).

The mutant compilation step also produces a list of join points that are matched by each mutant PCD. This enables the automatic identification of equivalent mutants, which is implemented in *AjMutator* based on a mutant classification schema that relies on a fault model for PCDs defined by Lemos et al. (2006). Mutants that are classified as

[9] *JAR* is an acronym for *Java ARchive*, which is a file the aggregates several files into a single one; it may include regular Java bytecode files, source code files and any other files for which the JAR file is intended to. JAR files are typically used for distribution of APIs and user applications.

non-equivalent are executed on the test set. *AjMutator* allows the tester to run *JUnit* test cases and identifies non-compilable and dead mutants. The tool is executed in command line mode and outputs an XML file that contains information about every mutant handled (e.g. mutant status, PCD ID and aspect ID).

We highlight here that *AjMutator* implements a set of mutation operators that consists in one of the contributions of this thesis (Ferrari et al., 2008). Such mutation operators are described further in Chapter 4 and Appendix C. We also discuss the limitations of *AjMutator* and present a tool that overcomes some of them in Chapter 5 and Appendix D.

2.3.5.4 Summary of AO Testing Tool Features

Table 2.6 summarises the main features of the tools described in this section. The list of features is based on a previous list created by Horgan and Mathur (1992) for characterising tools that support structural and mutation testing of C and Fortran programs.

Table 2.6: Summary of features present in tools for AO testing.

Feature	*Aspectra*	*JaBUTi/AJ*	*AjMutator*
Supported language	AspectJ	AspectJ	AspectJ
Test-case generation	yes	no	no
Interface	command line	GUI	command line
Automatic test execution	yes	yes	yes
Automatic test evaluation	yes	yes	yes
Recording of unfeasible requirements	no	yes (manually)	yes (partially automated)
Supported testing phase	unit and integration[1]	unit and integration	unit and integration[2]

[1] *Aspectra* generates test requirements that involve integrated units (e.g. requirements derived from the *Interaction Coverage* criterion (Xie and Zhao, 2006)).

[2] The PCD-related mutations supported by *AjMutator* affect the number of aspect-base code interactions, that is, they affect the integration of units across the system under test.

As stated in the beginning of this section, we can observe that these tools support the basic steps of software testing. Apart from deriving test requirements according to the automated criteria, they all support automatic test execution and evaluation (i.e. the achieved test coverage). We can also notice that all tools target AspectJ programs and support both unit and integration testing phases. Note that Chapter 5 of this dissertation describes some limitations of *AjMutator* with respect to the mutation testing steps, and presents the *Proteum/AJ* tool (Ferrari et al., 2010c) that overcomes some of them.

2.4 Final Remarks

This chapter described the underlying theory and concepts that are approached in this thesis, which included the fundamentals of AOP and software testing. Furthermore, it characterised the state of the art in AO testing approaches, fault taxonomies and automated tool support. This allowed us to identify some limitations on the current research on AO testing, which are following described:

- Despite the existence of several fault taxonomies for AO software, we noticed they are either described in a high-level, imprecise fashion or generally do not address all main AOP constructs such as PCDs, advices and ITDs. For example, Alexander et al. (2004)'s taxonomy does not include faults related to intertype declarations, while van Deursen et al. (2005) mix more than one fault type into a single item (e.g. "*Wrong advice specification (using before instead of after, using after with the wrong argument, etc.)*"). Finally, the existing taxonomies still lack empirical evaluation, which may represent a hindrance for the definition of testing strategies on top of them as well as their adoption in real software development projects.

- To date, the few fault-based testing approaches for AO software, along with the associated tool support, face similar limitations as faced by fault taxonomies: they focus on a subset of AOP constructs – pointcut descriptors, in particular – and have not been properly evaluated with respect to properties like relevance, required effort and effectiveness.

The contributions we present in the next chapters aim at overcoming some of these limitations. They include an empirical evaluation of the fault-proneness of AO programs based on a coarse-grained fault classification, and a subsequent refinement of this classification in order to define a comprehensive fault taxonomy for AO software (Chapter 3). We also define (Chapter 4), automate (Chapter 5) and evaluate (Chapter 6) a set of mutation operators that model varied instances of the fault types that are described in the taxonomy.

Chapter

3

Evaluating the Fault-Proneness of Aspect-Oriented Programs

As earlier discussed in Chapter 1, AOP has as its main goal enhancing the modularisation of crosscutting concerns that are implemented in the software. Despite the claimed benefits achieved with AOP, previous research has highlighted that its complementary set of concepts and programming mechanisms represents potential sources of faults, thereby requiring specific testing approaches to deal with them (Alexander et al., 2004; Lemos et al., 2007; McEachen and Alexander, 2005; Zhao, 2003).

In Chapter 2 (Section 2.3) we summarised relevant research that concerns testing techniques and fault characterisation for AO software. Our summary was based on the results of a systematic mapping study we have been regularly updating in the last few years (Ferrari et al., 2009; Ferrari and Maldonado, 2006, 2007). It shows that a number of fault taxonomies for AO software have been proposed to date, mostly based on language features and researchers' expertise. However, such taxonomies have not been evaluated with respect to their ability in classifying faults uncovered in real software development scenarios. They seem not to be comprehensive enough to classify the several fault types that may appear in software projects, given that when considered individually, those taxonomies do not fully address the commonly used AOP mechanisms.

In this chapter we present the results of two studies that address the evaluation of the fault-proneness of the main AOP properties and elements as well as the definition of a comprehensive fault taxonomy for AO software. We start with the study presented in Section 3.1, which focuses on collecting and categorising faults from several releases of AO applications according to a coarse-grained fault classification for AO software. Such classification comprises the four main elements we can identify in an AO system: (i) pointcut expressions (PCDs); (ii) intertype declarations (ITDs) and declare-like expressions[1]; (iii) advices; and (iv) the base program. This first study also evaluates the impact of the obliviousness property (Filman and Friedman, 2004) on the fault-proneness of evolving AO programs. The overall results motivates a revision of the claimed benefits provided by obliviousness as well as more intensive research on testing of other AOP-specific mechanisms beyond PCDs.

The second study is presented in Section 3.2. It extends the study presented in Section 3.1 in three different ways: (i) it refines the coarse-grained fault classification in order to compose a comprehensive fault taxonomy for AO software; (ii) it quantifies the fault occurrences according to the proposed fault taxonomy; and (iii) it characterises recurring faulty implementation scenarios based on the analysed set of faults. The results confirm the ability of the fault taxonomy in classifying all faults uncovered from the evaluated systems. Besides that, they provide hints on fault-prone implementation scenarios that should be either avoided or double-checked by developers during the development phase.

3.1 A Study of the Fault-Proneness of AO Programs

The establishment of testing approaches that deal with newly introduced programming technologies should rely on existing knowledge of fault-prone mechanisms and harmful implementation scenarios. Clearly, this also holds for the complementary set of mechanisms and properties that enable AOP in any development paradigm, given that varying AOP approaches are expected to share common characteristics such as quantification, obliviousness and aspect-base code composition (Filman et al., 2004; Kiczales et al., 1997).

Despite the claimed benefits that can be achieved with the adoption of AOP, scepticism about the quality of the derived products still remain. One of the contributing factors for this uncertainty regards the lack of empirical evaluation of how the AOP-specific properties and mechanisms lead to the insertion of faults into the software. This, in turn, hinders the development of adequate testing approaches that help to promote the adoption of AOP

[1]For the sake of simplicity, we hereafter refer to either an intertype declaration or a declare-like expression as an ITD.

by the industry, specially in the context of increasingly incremental software development processes which we can find nowadays.

This section summarises the results of an exploratory study that tackles the aforementioned issues. It develops in terms of two hypotheses that address (i) the impact of an AOP underlying property – the *obliviousness* – on the correctness of evolving AO programs and (ii) the differences amongst the fault-proneness of the main AOP mechanisms in the context of such programs. An overview of the study objectives, employed procedures and achieved results is presented in the sequence. The full contents of this study are presented in Appendix A together with a copyright notice from the Association for Computing Machinery (ACM).

3.1.1 Goals and Method

This study aims at evaluating the fault-proneness of AOP properties and mechanisms when they are applied to evolving AO programs. In particular, we are interested in identifying the underlying factors that lead to the introduction of faults. Our analysis develops in terms of two hypotheses, whose null and alternative variants are as follows:

Hypothesis 1 (H1)

- H1-0: Obliviousness does not exert impact on the fault-proneness of evolving AO programs.
- H1-1: Obliviousness exerts impact on the fault-proneness of evolving AO programs.

Hypothesis 2 (H2)

- H2-0: There is no difference among the fault-proneness of the main AOP mechanisms.
- H2-1: There are differences among the fault-proneness of the main AOP mechanisms.

To achieve our goals, we applied a number of evaluation procedures that include testing, static analysis, debugging, and fault documentation and classification. We analysed faults that were collected from three evolving AO systems, which come from three different domains. The first is iBATIS, a Java-based open source framework for object-relational data mapping (iBATIS Development Team, 2009). The AO versions of iBATIS have some functional and non-functional concerns modularised within aspects (e.g. exception handling, concurrency and type mapping) (Ferrari et al., 2010a). The second is HealthWatcher (HW), which consists in a Web-based application that allows citizens to register

complaints regarding health issues (Greenwood et al., 2007; Soares et al., 2006). Some aspectised concerns in HealthWatcher are distribution, persistence and exception handling. The third is MobileMedia (MM), which consists in a software product line for mobile devices that allows users to manipulate image files in different mobile devices (Figueiredo et al., 2008). In MM, aspects are used to configure the product line instances, enabling the selection of alternative and optional features.

The analysis with respect to the H1 hypothesis developed from two main viewpoints: the *fragile pointcut problem* (Stoerzer and Graf, 2005) and a categorisation of obliviousness listed by Sullivan et al. (2005). The former concerns how software evolution causes PCDs to break, i.e. whether or not PCDs mismatch the intended join points while the programmers perform changes in the base code. The latter addresses different levels of obliviousness that can be present throughout the software life cycle, starting from low-level (language-based) obliviousness and moving to higher levels such as feature obliviousness and perfect (or *pure*) obliviousness (Sullivan et al., 2005).

The analysis regarding the H2 hypothesis started with the overall evaluation of the fault-proneness of the main AOP mechanisms, namely PCDs, advices and ITDs. In the sequence, we analysed the correlation between such mechanisms and the associated fault counts when concern-specific implementations are individually considered. This analysis took into account only crosscutting concerns that have been modularised within aspects in the evaluated systems. It was motivated by the fact that AOP mechanisms may have individual impact on the fault-proneness of a module, a cluster of modules (e.g. modules that implement a given concern) or the full system.

3.1.2 Results

Table 3.1 presents the number of faults identified in the three evaluated systems according the main AOP-related elements: PCDs, ITDs, advices and the base program. Table 3.2, on the other hand, summarises the number of faults that were related to two of the obliviousness levels listed by Sullivan et al. (2005): language-level and feature obliviousness. The former kind of obliviousness is present when there is no local notation in the code about aspect behaviour that may be inserted at possibly selected join points. Differently, feature obliviousness is present when the developer is unaware of the features or the in-depth semantics of an aspect that is advising the base code. While establishing the relationship between each fault with a specific level of obliviousness, we answered the following question: *Could this fault have been avoided if such level of obliviousness was*

not present in this implementation scenario? This allowed us to reason about the impact of such property on the presence of particular faults in the evaluated systems.

Table 3.1: Fault distribution per system – adapted from Ferrari et al. (2010a).

Fault Type	System			Total
	iBATIS	MobileMedia	HealthWatcher	
PCD-related	18	1	0	19
ITD-related	14	4	6	24
Advice-related	15	4	4	23
Base program-related	36	0	2	38
Total	83	9	12	104

Table 3.2: Faults associated with obliviousness – adapted from Ferrari et al. (2010a).

System	Obliviousness Category					
	Language	Feature	Both	Language only	Feature only	Total
iBATIS	31	4	4	27	0	31
HealthWatcher	0	3	0	0	3	3
MobileMedia	8	4	4	4	0	8
Total			8	31	3	42

Results for the H1 hypothesis: in our analysis we noticed that 27 out of 36 base program-related faults identified in the iBATIS system (see Table 3.1) were caused by either perfective or evolutionary changes within the base code, which led PCDs to break. This number corresponds to 33% of the faults revealed for that system, or 26% of the overall number of faults considering all systems. This high occurrence of broken PCDs is known as the *fragile pointcut problem* (Stoerzer and Graf, 2005) and is closely related to the quantification and, more concerning still, the obliviousness models implemented in AspectJ-like languages. We found that this problem is magnified in realistic development scenarios as the one observed in iBATIS, for which several developers worked in parallel, each of them refactoring and evolving different crosscutting concerns into aspects. According to Gybels and Brichau (2003), program changes requires revisions of the crosscut enumerations (i.e. the PCDs), which conflicts with the idea of programs being oblivious to the aspects applied to them.

When we consider the figures presented in Table 3.2, we can observe that a total of 42 faults could be directly associated with at least one of the two levels of obliviousness

considered in this study. This number represents 40% of the total number of faults across the three evaluated systems.

These results provided us with evidence that support the H1 alternative hypothesis (i.e. H1-1), since a large amount of faults could be directly associated with the base code being oblivious to aspects or even aspects being oblivious to other aspects. This corroborates recent trends in research on AOP approaches like aspect-aware interfaces (Kiczales and Mezini, 2005), Crosscut Programming Interfaces (XPIs) (Griswold et al., 2006) and Explicit Join Points (EJPs) (Hoffman and Eugster, 2007). Such approaches tend to reduce the language-level obliviousness in favour of better program comprehension by making aspect-base interactions more explicit. Not surprisingly, language-level obliviousness happened to be the category associated with the largest number of faults in our study.

Results for the H2 hypothesis: the results presented in Table 3.1 go against the general assumption that PCD is the most fault-prone AOP mechanism. The distribution of faults is very similar with respect to the three main AOP mechanisms, varying from 19 to 24 amongst PCDs, ITDs and advices.

Our additional analysis addressed individual concern implementations, as described in the previous section. Table 3.3 lists the results of the Spearman's rank correlation test when considering the fault counts associated with each concern and the maximum and average number of AOP elements used to implement that concern. We can observe that the correlation between PCD and advice usage (both maximum and average) and the average number of faults per concern is significant (lines highlighted in grey in Table 3.3). For ITDs, the correlation coefficient varies between 0.5 and 0.6, what means moderate-to-large correlation on average if we consider a confidence level of ∼85%.

We compared these achieved results with two metrics that have been reported as good fault-proneness indicators in studies that comprise OO programs (Gyimóthy et al., 2005; Subramanyam and Krishnan, 2003): lines of code (LOC) and weighted operations per module (WOM[2]). The values obtained for such metrics – also listed in Table 3.3 – showed non-significant correlation with fault counts in our study. This indicates that when we consider the set of modules involved in AO implementations of crosscutting concerns, the internal number of AOP-specific mechanisms (i.e. PCDs, advices and ITDs) are better fault-proneness indicators than OO-based metrics.

[2]WOM (Ceccato and Tonella, 2004) adapts the original weighted methods per class (WMC) metric (Chidamber and Kemerer, 1994) to count methods inside classes as well as aspect operations (i.e. advices, methods and intertype methods).

Table 3.3: Correlation between maximum and average AOP mechanism usage and the average number of faults per concern/release. – adapted from Ferrari et al. (2010a).

Metric	Coefficient	P-value
MAX-PCDs	0.8809524**	0.0072420
MAX-ADVICES	0.8742672**	0.0045120
MAX-ITDs	0.5509081	0.1570000
MAX-LOC	0.1904762	0.6646000
MAX-WOM	0.1666667	0.7033000
AVG-PCDs	0.8571429*	0.0107100
AVG-ADVICES	0.8571429*	0.0107100
AVG-ITDs	0.5714286	0.1511000
AVG-LOC	0.1904762	0.6646000
AVG-WOM	0.1666667	0.7033000

** correlation is significant at the 0.01 level

* correlation is significant at the 0.05 level

To conclude, our findings support the H2 null hypothesis (i.e. H2.0) since (i) the overall fault distribution per main AOP mechanism is similar; and (ii) the usage rate of each mechanism does not vary independently. Instead, it depends on the set of concerns aspectised within a system and tends to be directly proportional to the number of faults associated with that concern.

3.2 Defining and Evaluating a Fault Taxonomy for AO Programs

The characterisation of software faults is important because it provides empirical evidence on how they occur in practice, as opposed to fault descriptions that are based solely on the characteristics of programming languages or development approaches. This phenomenon has been widely investigated by researchers over the years. It includes analyses of how specific programming features can be sources of faults in software systems (Offutt et al., 2001) and empirical observations of how different types of faults appear in the context of real software development projects (Basili and Perricone, 1984; Endress, 1978; Ostrand and Weyuker, 1984).

When it comes to AOP, however, most of related research on software faults has targeted the classification of faults based on programming features, but not on empirical analysis of real software development data. This section describes the results of a study

that contributes to fill this gap, as an extension of the study presented in Section 3.1. It includes the definition of a comprehensive fault taxonomy that leverages previously described taxonomies which we identified in our systematic mapping study on AO testing (see Chapter 2, Section 2.3 for more details). To evaluate the ability of our taxonomy in classifying varied instances of fault types, we categorised all faults that compose the fault set identified for the study we described in Section 3.1. Besides that, we identified and characterised recurring faulty scenarios in order to provide hints for the establishment of AOP-specific testing approaches. A summary of the study objectives, employed procedures and achieved results is presented in the sequence. The full study contents are presented in Appendix B together with a copyright notice from the Institute of Electrical and Electronics Engineers (IEEE).

3.2.1 Goals and Method

The goals of this study are three-fold: (i) defining a fault taxonomy for AO programs which includes fault types that are distributed across the four categories earlier identified (i.e. PCD-, ITD-, advice- and base program-related faults); (ii) quantifying and categorising faults in AO programs according to the proposed fault taxonomy; and (iii) characterising recurring faulty scenarios of AOP.

To achieve these goals, we initially identified and grouped together several fault types for AO software that have been described by other researchers (the pieces of work that describe fault types for AO software can be identified in Chapter 2, Table 2.4). Additionally, we included new fault types that can occur in programs written in AspectJ, which represents a mainstream AOP supporting technology.

We performed a preliminary evaluation of the taxonomy using the fault set obtained in our previous study about fault-proneness of AO programs (Ferrari et al., 2010a). In doing so, we were able to check whether the taxonomy is complete enough to allow for the classification of all faults revealed from the target systems. Finally, we went through the classified fault set in order to spot recurring problems. This last procedure allowed us to characterise the most fault-prone implementation scenarios, which includes code excerpts extracted from the evaluated systems and the steps that might have led to the insertion of the faults. In this way, developers are provided with hints about harmful scenarios and can take the appropriate actions in order to mitigate the risks.

3.2.2 Results

The proposed fault taxonomy encompasses 26 different fault types distributed over four main categories, which are listed in Tables 3.4–3.7. Category F1 (Table 3.4) includes

eight PCD-related fault types that address, for instance, incorrect join point quantification, misuse of primitive pointcut designators and incorrect PCD composition rules. Category F2 (Table 3.5) 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 (Table 3.6) describes six types of faults related to advice definition and implementation, for example, improper advice type specification, incorrect advice logic and incorrect advice-PCD binding. Finally, category F4 (Table 3.7) includes three faults types whose root causes can be assigned to the base program. For instance, code evolution that causes PCDs to break and duplicated crosscutting code due to improper concern refactoring.

Table 3.4: Faults related to PCDs – adapted from Ferrari et al. (2010b).

ID	Description
F1.1	Selection of a superset of intended JPs.
F1.2	Selection of a subset of intended JPs.
F1.3	Selection of a wrong set of JPs, which includes both intended and unintended items.
F1.4	Selection of a wrong set of JPs, which includes only unintended items.
F1.5	Incorrect use of a primitive PCD[1].
F1.6	Incorrect PCD composition rules.
F1.7	Incorrect JP matching based on exception throwing patterns.
F1.8	Incorrect JP matching based on dynamic circumstances.

[1] Primitive PCDs are predefined PCDs available in AOP languages (e.g. in AspectJ (The Eclipse Foundation, 2010b)).

Table 3.5: Faults related to ITDs and declare-like expressions – adapted from Ferrari et al. (2010b).

ID	Description
F2.1	Improper method introduction, resulting in unanticipated method overriding or not resulting in anticipated method overriding.
F2.2	Introduction of a method into an incorrect class.
F2.3	Incorrect change in class hierarchy through parent declaration clauses V (e.g. `declare parents` statements), resulting in unintended inherited behaviour for a given class.
F2.4	Incorrect method introduction, resulting in unexpected method overriding.
F2.5	Omitted declared interface or introduced interface which breaks object identity.
F2.6	Incorrect changes in exception-dependent control flow, resulting from aspect-class interactions or from clauses that alter exception severity.
F2.7	Incorrect or omitted aspect precedence expression.
F2.8	Incorrect aspect instantiation rules and deployment, resulting in unintended aspect instances.
F2.9	Incorrect policy enforcement rules supported by warning and error declarations.

Table 3.6: Faults related to advices – adapted from Ferrari et al. (2010b).

ID	Description
F3.1	Incorrect advice type specification.
F3.2	Incorrect control or data flow due to incorrect aspect-class interactions.
F3.3	Incorrect advice logic, resulting in invariants violations or failures to establish expected postconditions.
F3.4	Infinite loops resulting from interactions among pieces of advice.
F3.5	Incorrect access to JP static information.
F3.6	Advice bound to incorrect PCD.

Table 3.7: Faults related to the base program – adapted from Ferrari et al. (2010b).

ID	Description
F4.1	The base program does not offer required JPs in which one or more foreign aspects were designed to be applied.
F4.2	The software evolution causes PCDs to break.
F4.3	Other problems related do base programs such as inconsistent refactoring or duplicated crosscutting code.

We classified the 104[3] faults identified from the three evaluated systems according to the fault types defined in the taxonomy. Figure 3.1 presents the resulting fault distribution. Note that we intentionally omitted the "F" from each fault type listed in the chart in order to improve its readability. Therefore, in Figure 3.1, a fault of type F1.1 is represented as 1.1, F1.2 as 1.2 and so on.

Remarks on the fault distribution: Looking at Figure 3.1, we can notice that some fault types present higher occurrence than the remaining types within each individual category. For example, while some fault types within the F3 category (i.e. advice-related faults) show totals of 15 (F3.3) and 5 (F3.1), other types from the same group vary between 0 and 1 (e.g. F3.4, F3.5 and F3.6). Apart from the individual fault types that stood out within each category, we identified three other particularities that yielded worthwhile discussions: (i) the PCD fragility problem; (ii) static crosscutting versus dynamic crosscutting and (iii) advice execution order. The fragile PCD issue has already been discussed in our previous study (see Section 3.1 for more details). The other two issues are discussed in the sequence.

The static crosscutting versus dynamic crosscutting matter regards the significant differences between the fault counts associated with elements that implement the two

[3]The numbers of faults per type slightly differ from the original totals (Ferrari et al., 2010a). The changes resulted from a revision of the fault documentation/classification. Nevertheless, these differences have no significant impact on the achieved results and conclusions of our previous study, which was presented in Section 3.1.

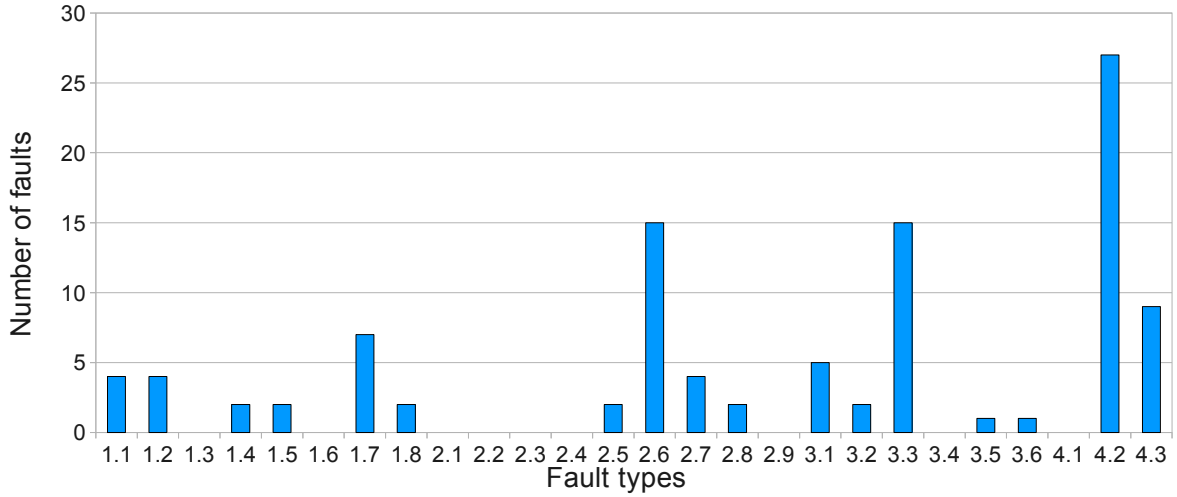


Figure 3.1: Fault distribution according to the fault taxonomy for AO software – extracted from Ferrari et al. (2010b).

models of crosscutting in the analysed systems: static and dynamic (The AspectJ Team, 2003). In general, faults related to static crosscutting resulting from the use of ITDs – more specifically, F2.1, F2.2, F2.3 and F2.4 – can generally be detected at compilation time. On the other hand, faults related to mechanisms that enable dynamic crosscutting – i.e. the advices – can mainly be detected during the software execution, thereby resulting in more occurrences during the testing phase.

In regard to the advice execution order issue, the problem was mainly observed when advices from two independent – or orthogonal (Kienzle et al., 2003) – functionalities share a common join point. This problem can be associated with the level of obliviousness present in code, given that even though two concerns may be aware of each other at source code level (i.e. language-based obliviousness is not present), uncertainty about the in-depth semantics of implemented features (i.e. feature obliviousness) may result in more faults of this nature.

Characterisation of recurring problems: we selected and described representative examples of faults for the four coarse-grained categories defined in our taxonomy, i.e. PCD-, ITD-, advice- and base program-related faults. The selection was based on the most recurring faulty scenarios within each category.

Figure 3.2 depicts an example of a recurring fault of the F2 category. It shows a case in which two different aspects – `SMSAspect` and `PhotoAndMusicAspect` – are advising the

same join point, whose matching is defined by the PCDs shadowed in grey. However, no precedence order is defined for these two aspects. This arbitrary execution order impacts future error recovery actions in case one of the advices presents abnormal behaviour.

```
public privileged aspect SMSAspect {
  ...
  pointcut startApplication(MainUIMidlet middlet):
          execution(public void MainUIMidlet.startApp())
          && this(middlet);

  after(MainUIMidlet middlet): startApplication(middlet) {
    ...
  }
  ...
}

public aspect PhotoAndMusicAspect {
  ...
  pointcut startApp(MainUIMidlet midlet):
          execution( public void MainUIMidlet.startApp() )
          && this(midlet);

  after(MainUIMidlet midlet): startApp(midlet) {
     ...
  }
  ...
}
```

Figure 3.2: Example of fault related to arbitrary advice execution order – extracted from Ferrari et al. (2010b).

Additionally, for each described faulty scenario, we enumerated the main steps that might have led to the introduction of the fault. For instance, considering the example described in Figure 3.2, the steps are: (1) Two PCDs p_1 and p_2 are defined to match a common join point; (2) The advices that are bound to p_1 and p_2 are implemented and the affected join points are possibly verified within each aspect; (3) The affected join points are not verified from the base program side, thus they are advised in arbitrary order. Such steps should be double-checked in order to reduce the risks of new fault introductions.

3.3 Summary of Contributions and Limitations

The studies summarised in this chapter bring the following contributions:

- *Evaluation of the fault-proneness of the main AOP mechanisms.* Differently from previous research on this topic, our evaluation was not solely based on language and programming features but on quantitative data. The data was obtained through extensive evaluation of several releases of AO applications, all having respective OO

counterparts that were used as baselines for implementation assessment. Our results showed that the main AOP mechanisms pose similar risk to the correctness of a program, therefore contradicting the common wisdom of considering that PCDs represent the main source of faults in AOP.

- *Evaluation of the impact of the obliviousness property on the fault-proneness of AO programs.* The benefits and drawbacks of obliviousness have yielded several discussions within the AOP community. However, due to the lack of available data extracted from real software projects, the arguments have been more or less informed speculation. Our study provided evidence suggesting that: (i) aspect-based code interactions should be given more attention during the software evolution in order to reduce the number of faults introduced at those points; and (ii) more investigation of AOP approaches that make interactions between aspects and base code more explicit is required.

- *Definition of a comprehensive fault taxonomy for AO software.* The taxonomy overcomes limitations of previous taxonomies in terms of completeness, detailing of fault type descriptions, and evaluation based on data extracted from varied AO applications. A preliminary evaluation demonstrates the taxonomy's general ability in categorising a set of faults that have been documented from several releases of AO applications.

- *Characterisation of recurring faulty scenarios in AO programs.* We identified and described fault-prone implementation scenarios based on the documented fault set. Within each main fault category, we selected examples of the most recurring faults and highlighted their location using source code excerpts. In addition, we enumerated the steps the might have led to introduction of the fault, therefore provide guidance for code inspection and debugging during the software development and maintenance phases.

The main limitations of the studies described in this chapter concern the generalisation of the achieved results. Firstly, we recognise that the AspectJ-like programming style cannot be assumed as the ultimate or unique AOP approach. In fact, as observed by Filman and Friedman (2004) in the early stages of research on AOP, several other programming techniques (e.g. Intentional Programming, Meta-Programming and Generative Programming) are able to realise the concepts of AOP. Oh the other hand, AspectJ has been far the most investigated AOP language, upon which several facets of AOP have been developed and evaluated. Moreover, the AOP model supported by AspectJ

has been implemented in several other language extensions and frameworks that support AOP. Examples are the JBossAOP (Burke and Brock, 2003) and SpringAOP (Johnson et al., 2007) frameworks, and the CaesarJ (Mezini and Ostermann, 2003), AspectC++ (Gal et al., 2001) and AspectC# (Kim, 2002) languages.

Another factor that can limit the generalisation of our results regards the representativeness of the applications we analysed in our studies. The size of these applications varies between 3,000 to 11,000 lines of code, thus possibly not properly reflecting industry-strength system complexities. Moreover, all systems have been derived from OO implementations in the academic context. However, we should have in mind that AOP is yet a maturing area that still requires considerable effort to make it become the state of the practice in the industry. Its cautious and slow adoption is demonstrated in the results of a survey of the usage of AOP undertaken by Muñoz et al. (2009). Therefore, evaluating the small- and medium-sized AO systems currently available provides insights and evidence that can support larger experimental studies as well as further developments in the area. From a general viewpoint, it can help researchers and practitioners work towards mature and robust AOP and its consequent adoption in the industrial scenario.

Chapter

4

Designing Mutation Operators for Aspect-Oriented Programs

Fault-based testing is a technique that relies on information about recurring mistakes made by programmers during the software development. It can be used, for example, to demonstrate the absence of prespecified faults in the software (Morell, 1990). As previously described in Chapter 2, this goal can be accomplished through mutation testing, which consists in a test selection criterion that systematically simulates faults into the software and evaluates if the current test data is sensitive enough to reveal those faults (DeMillo et al., 1978).

The definition of a mutation-based testing approach should rely on well-founded fault characterisation for the target software development technology. For example, fault taxonomies, fault models and bug pattern catalogues represent suitable means for characterising faults that are likely to be introduced into the software along its development. Once the set of prespecified faults is chosen, mutation operators are designed in order to introduce these faults into the piece of software under evaluation.

This chapter describes a set of mutation operators for AspectJ programs which can be applied in a fault-based testing approach for such programs. We use the following sources of information upon which we design the operators: (i) the fault taxonomy described in Chapter 3 and Appendix B; and (ii) the programming structures and their varied alterna-

tives allowed by the AspectJ language. In using these two sources, we are addressing two basic issues for the design of mutation operators: (1) the need for a well-established fault characterisation, through the use of the fault taxonomy for AO software; and (2) the dependence of mutation operators on the target technology (Delamaro et al., 2001), through the analysis of the AspectJ language syntax. We additionally perform an analysis of to what extent fault types described in the taxonomy can be generalised to AOP approaches and supporting technologies other than AspectJ.

This chapter is a summary of a paper published in the Proceedings of the 1^{st} International Conference on Software Testing, Verification and Validation (ICST). The full paper contents can be found in Appendix C together with a copyright notice from the Institute of Electrical and Electronics Engineers (IEEE).

4.1 Mutation Operators for AspectJ Programs

Tables 4.1, 4.2 and 4.3 summarise the set of mutation operators for AspectJ programs we propose in this chapter. The operators are grouped according to the main AOP mechanisms they target (namely PCDs, declare-like expressions, and advices), thus corresponding to the F1, F2 and F3 fault type categories defined in our fault taxonomy for AO software (see Chapter 3 for further information). Note that this set of mutation operators does not address base program-related faults (i.e. category F4 in the taxonomy) given that these faults can be simulated by existing mutation operators (e.g. for unit (Agrawal et al., 1989), interface (Delamaro et al., 2001) and class (Ma et al., 2002) levels).

Group 1 (Table 4.1) includes 15 operators which model faults related to PCDs. These faults usually result in incorrect join point matchings or undue execution contexts. Group 2 (Table 4.2) contains five operators that model faults related to AspectJ declare-like expressions. Faults that are modelled by them may lead to unintended control flow executions, possibly resulting in erroneous object/aspect state. Note that ITD-specific faults, which also compose the F2 fault category, can be mostly detected at compilation time (Ferrari et al., 2010b, 2008). Therefore, Group 2 does not include operators that alter such elements. Finally, Group 3 includes six operators related to advice definition and implementation.

4.1.1 Mutation Operators versus Fault Types

Table 4.4 shows the relation between AOP-specific fault types and the proposed mutation operators. The table only includes direct effects of operators in relation to the target elements and resulting mutants. Note that we do not focus on indirect effects of each

Table 4.1: Group 1: Mutation operators for PCDs – adapted from Ferrari et al. (2008).

Operator	Description/Consequences
PWSR	PCD weakening by replacing a type with its immediate supertypes.
PWIW	PCD weakening by inserting wildcards into it.
PWAR	PCD weakening by removing annotation tags from type, field, method and constructor patterns.
PSSR	PCD strengthening by replacing a type with its immediate subtype.
PSWR	PCD strengthening by removing wildcards from it.
PSDR	PCD strengthening by removing "`declare @`" statements from the aspect code.
POPL	PCD weakening or strengthening by modifying parameter lists of primitive PCDs.
POAC	PCD weakening or strengthening by modifying "`after [retuning| throwing]`" advice clauses.
POEC	PCD weakening or strengthening modifying exception throwing clauses.
PCTT	PCD changing by replacing "`this`" PCDs with "`target`" ones and vice versa.
PCCE	Context changing by switching "`call/execution/initialization/ preinitialization`" PCDs.
PCGS	PCD changing by replacing "`get`" PCDs with "`set`" ones and vice versa.
PCCR	PCD changing by replacing individual parts of a PCD composition.
PCLO	PCD changing by varying logical operators present in type and PCD compositions.
PCCC	PCD changing by replacing "`cflow`" PCDs with "`cflowbelow`" ones and vice versa.

Table 4.2: Group 2: Mutation operators for ITDs and declare-like expressions – adapted from Ferrari et al. (2008).

Operator	Description/Consequences
DAPC	Aspect precedence changing by alternating the order of aspects involved in `declare precedence` expressions.
DAPO	Arbitrary aspect precedence by removing "`declare precedence`" expressions.
DSSR	Unintended exception handling by removing "`declare soft`" expressions.
DEWC	Unintended control flow execution by changing "`declare error/warning`" expressions.
DAIC	Unintended aspect instantiation by changing "`perthis/pertarget/ percflow/percflowbelow`" deployment clauses.

Table 4.3: Group 3: Mutation operators advices – adapted from Ferrari et al. (2008).

Operator	Description/Consequences
ABAR	Advice kind changing by replacing a `before` clause with an `after [returning|throwing]` one and vice versa.
APSR	Advice logic changing by removing invocations to "`proceed`" statement.
APER	Advice logic changing by removing guard conditions which surround "`proceed`" statements.
AJSC	Static information source changing by replacing a "`thisJoinPoint- StaticPart`" reference with a "`thisEnclosingJoinPointStaticPart`" one and vice versa.
ABHA	Behaviour hindering by removing implemented advices.
ABPR	Changing PCD-advice binding by replacing PCDs which are bound to advices.

operator regarding to faults they model. For example, we defined a relationship between the PCTT operator and the F1.1–F1.5 fault types. This operator replaces a `this` primitive PCD with a `target` one and vice versa, possibly modifying the set of selected join points. If we considered indirect effects, we could also establish a relationship between PCTT and F1.8, since the matching of this primitive PCDs may be based on types which can be redefined at runtime. This observation also holds for the remaining operators.

Table 4.4: Relationship between mutation operators and fault types for AO software – adapted from Ferrari et al. (2008).

	Fault Types																					
	F1								F2									F3				
Operator	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6
PWSR	✓																						
PWIW	✓																						
PWAR	✓																						
PSSR		✓																					
PSWR		✓																					
PSDR		✓																					
POPL	✓	✓																					
POAC	✓	✓	✓	✓																			
POEC	✓	✓	✓	✓			✓																
PCTT	✓	✓	✓	✓	✓																		
PCCE					✓																		
PCGS				✓	✓																		
PCCR	✓	✓	✓	✓		✓															✓		
PCLO	✓	✓	✓	✓		✓																	
PCCC	✓	✓						✓															
DAPC															✓								
DAPO															✓								
DSSR														✓									
DEWC																	✓						
DAIC																✓							
ABAR																		✓					
APSR																			✓	✓			
APER																			✓	✓			
AJSC																						✓	
ABHA																				✓			
ABPR																							✓

4.1.2 Preliminary Cost Analysis

We performed a preliminary cost analysis based on the number of mutants generated for two AspectJ applications from two different domains. The first is the TollSystemDemonstrator (TSD) system, which includes a subset of requirements of a real-world tolling system. It has been developed in the context of the AOSD-Europe Project (AOSD Europe,

2010) and contains functional and non-functional concerns implemented within aspects (e.g. charging variabilities, distribution and logging). The second application is Health-Watcher (HW), which was already described in Chapter 3 and consists in a Web-based system that allows citizens to register complaints regarding health issues (Greenwood et al., 2007; Soares et al., 2006).

Tables 4.5 and 4.6 display the obtained cost analysis results. The figures listed in Table 4.5 regard the total numbers of mutants per group. Mutants have been manually created for each of the systems. Operators from Group 1 yielded the largest number of mutants (more than 80% for both systems), whereas operators from Group 2 produced less than 2% of the total.

Table 4.5: TSD and HW mutants – adapted from Ferrari et al. (2008).

System	Total	Group 1	%	Group 2	%	Group 3	%
TSD	981	847	86	8	1	126	13
HW	388	342	88	8	1.5	121	10.5

The average number of mutants generated for each main AOP element is presented in Table 4.6. For groups which result in the largest numbers of mutants (i.e. PCDs and advices - see Table 4.5), the average per element is similar for both applications. The apparently high number of mutants generated for PCDs and the implications with respect to mutant analysis costs can be mitigated by equivalent mutant detection strategies. An example of such strategy will be presented in Chapter 5, which describes a tool that automates the majority of the mutation operators proposed herein.

Table 4.6: Mutant average per element type – adapted from Ferrari et al. (2008).

System	Aspect	PCD	declare-like	Advice
TSD	39	23	4	3
HW	35	21	1	3

4.2 Generalisation of the Fault Taxonomy

In addition to the defined set of mutation operators, we analysed to what extent the fault taxonomy for AO software can be generalised. For this analysis, we considered four AO technologies: AspectJ, JBossAOP (The JBoss Team, 2010) and SpringAOP (Johnson et al., 2007), which represent technologies that have a significant user base; and Cae-

sarJ (Mezini and Ostermann, 2003), which is another language that provides a number of features to enable AOP.

Table 4.7: Relationship between AOP technologies and fault types – adapted from Ferrari et al. (2008).

	Fault Types																									
	F1								F2									F3						F4		
Technology	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	1	2	3
AspectJ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
SpringAOP	✓	✓	✓	✓	✓	✓	✓	✓			✓		✓		✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
JBossAOP	✓	✓	✓	✓	✓	✓	✓	✓			✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CaesarJ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

The results of our analysis are summarised in Table 4.7. Our observations show that most fault types, initially characterised upon AspectJ features, actually may occur in general AOP implementations and approaches. Considering that the proposed mutation operators for AspectJ programs cover a wide range of faults described in the taxonomy, adapting them for other AspectJ-like languages sounds an interesting approach to support the fault-based testing of other AOP implementations.

4.3 Summary of Contributions and Limitations

The key contributions of this chapter are:

- *Definition of a set of mutation operators for AspectJ-like programs.* The proposed mutation operators address the majority of the fault types described in the fault taxonomy for AO software. Their definition has in fact been guided by the taxonomy as well as by the syntactic alternatives allowed by the AspectJ language, therefore fulfilling the requirements for the establishing of a fault-based testing approach for AO programs.

- *Preliminary cost analysis for the application of the mutation operators.* The operators have been manually applied to two AO applications that encompass a wide range of mechanisms often used to implement different categories of crosscutting concerns. The average number of generated mutants for each group – i.e. PCDs, declare-like expressions and advices – provides a preliminary cost estimate for the application that can be double-checked once the operators are fully automated.

- *Evaluation of the possible generalisations of the fault taxonomy.* To date, research on fault characterisation for AO software has mainly developed in terms of the As-

pectJ language. In spite of that, our analysis showed that most of the fault types can actually occur in other AOP supporting technologies, according to the range of programming constructs they provide. As a consequence, the testing of programs developed with such technologies can leverage existing knowledge on testing AspectJ programs.

The main limitations of the work presented in this chapter concern: (i) the lack of a complexity analysis for each defined mutation operator; (ii) the effort estimation for producing adequate test sets with respect to the generated mutants; and (iii) the lack of automated support for the application of the mutation operators

In regard to limitations (i) and (ii), so far we performed a preliminary cost estimation based on the manual application of the operators in two medium-sized AO systems (Section 4.1.2). It comprises a single measure: the number of derived test requirements which, in particular, is represented by the number of mutants created for the target systems. However, we have not evaluated the complexity of applying individual mutation operators based on the worst case, i.e. based on the maximum number of mutants each operator may produce for a given aspect under test. Such kind of analysis, for instance, was undertaken by Delamaro (1997) for interface mutation operators. Furthermore, other cost-related measures (e.g. the number of equivalent mutants, the number of required test cases to kill the mutants and the effort required for analysing live mutants) should also be taken into consideration. In fact, some of these cost-related attributes are evaluated in Chapter 6. In regard to limitation (iii), it is addressed in the next chapter.

Chapter

5

Automating the Mutation Testing of Aspect-Oriented Programs

As previously discussed in Chapter 2, software testing depends on adequate tool support to be undertaken in a systematic and rigorous fashion, therefore helping to enhance the users' confidence that the software behaves as expected (Harrold, 2000; Weyuker, 1996). When it comes to automating AO testing, we have observed in the systematic mapping study presented in Chapter 2 that initiatives on providing tool support to testing are still limited when contrasted to the large number of proposed approaches. Particularly regarding mutation-based testing, none of the identified tools (Anbalagan and Xie, 2008; Delamare et al., 2009b) provide adequate support for its basic steps, namely original program execution, mutation generation, mutant execution and mutant analysis (DeMillo et al., 1978).

This chapter presents a tool named *Proteum/AJ*, which automates the mutation testing of AO programs written in AspectJ. *Proteum/AJ* implements the set of mutation operators described in Chapter 4, and provides support for the main steps required by the Mutant Analysis criterion. It leverages previous knowledge on developing *Proteum*, a family of tools for mutation testing developed by the Software Engineering group at the University of São Paulo, Brazil (Maldonado et al., 2000). *Proteum/AJ* overcomes some limitations of other existing tools that support mutation testing of AO programs (Anbal-

agan and Xie, 2008; Delamare et al., 2009b). These limitations can be checked against a set of requirements for such kind of tools, as discussed in Section 5.1. *Proteum/AJ*'s functionalities and its implementation details are described in Sections 5.3 and 5.4, respectively.

This chapter is a summary of a paper published in the Proceedings of the 5^{th} ICSE International Workshop on Automation of Software Test (AST). The full paper contents can be found in Appendix D together with a copyright notice from the Association for Computing Machinery (ACM). Note that some additional information about the tool implementation is presented in Section 5.4.

5.1 Requirements for Mutation Tools

We identified a set of requirements for mutation-based testing tools according to a minimal set of requirements listed by Delamaro and Maldonado (1996) and some common features observed by Horgan and Mathur (1992). Besides that, we also identified some requirements that regard experimentation in software testing (Vincenzi et al., 2006b) and testing of modern software systems such as AO software. The final list is presented in Table 5.1.

Table 5.1: Requirements fulfilled by tools for mutation testing of AO programs – adapted from Ferrari et al. (2010c).

Requirement	Proteum/AJ	AjMutator (Delamare et al., 2009b)	Anbalagan and Xie (2008)'s tool
1. Test case handling*	partial	partial	no
2. Mutant handling*	yes	partial	partial
3. Adequacy analysis*	partial	partial	partial
4. Reducing test costs†	no	no	no
5. Unrestricted program size†	yes	n/a	n/a
6. Support for testing strategies⊙	partial	no	no
7. Independent test configuration	yes	yes	no

*From Delamaro and Maldonado (1996) †From Horgan and Mathur (1992) ⊙From Vincenzi et al. (2006b)

Limitations of current tools and improvements implemented in Proteum/AJ: From Table 5.1, we can observe how *Proteum/AJ* and the two other tools for mutation testing of AO programs address the listed requirements. Anbalagan and Xie (2008)'s tool is limited to the creation and classification of mutants based on a very small set of mutation operators. No support for test case and mutant handling is provided. *AjMutator*, on the

other hand, provides better support than Anbalagan and Xie's tool, however it still misses some basic functionalities such as mutation operator selection and proper mutant execution and analysis support (e.g. individual mutant execution and manual classification of mutants).

Contrasting *Proteum/AJ* with the other previous tools, we highlight that it improves test case handling features (e.g. importing and executing test cases into the running test project), enables mutant handling (e.g. individual mutant execution) and supports testing strategies (e.g. incremental selection of mutation operators and target aspects).

Proteum/AJ allows the tester to manage mutants in several ways. For example, mutants can be created, recreated and individually selected for execution. The execution can also be restricted to live mutants only, and these can be manually set as equivalent and vice versa, that is, equivalent mutants can be reset as alive. The tool also enables the tester to import and execute new test cases within an existing test project. The mutation score can be computed at any time after the first tests have been executed.

The size of the application under test is not constrained by *Proteum/AJ*. Decompression and compilation tasks are delegated to third-party tools configured through *Ant* (The Apache Software Foundation, 2009) tasks. The test setup is also fully configurable through Ant tasks that are invoked by the tool. In this way, there are only minor dependencies between the test execution configuration and the tool.

5.2 The Architecture of Proteum/AJ

Figure 5.1 depicts the architecture of *Proteum/AJ*. The architecture is based on a reference architecture for software testing tools called *RefTEST* (Nakagawa et al., 2007). *RefTEST* is based on separation of concerns (SoC) principles, the Model-View-Controller (MVC) and three tier architectural patterns, and the ISO/IEC 12207 standard for Information Technology. *RefTEST* encourages the use of aspects as the mechanism for integrating the core activities of a testing tool with tools that automates supporting and organisational software engineering activities defined in ISO/IEC 12207 (e.g. planning, configuration management and documentation tools). Moreover, aspects are also encouraged for integrating services such as persistence and access control. *Proteum/AJ* benefited mainly from the reuse of domain knowledge contained in *RefTEST*. The instantiation of the architecture provided us with guidance on how we could structure the tool in terms of functionalities and module interactions.

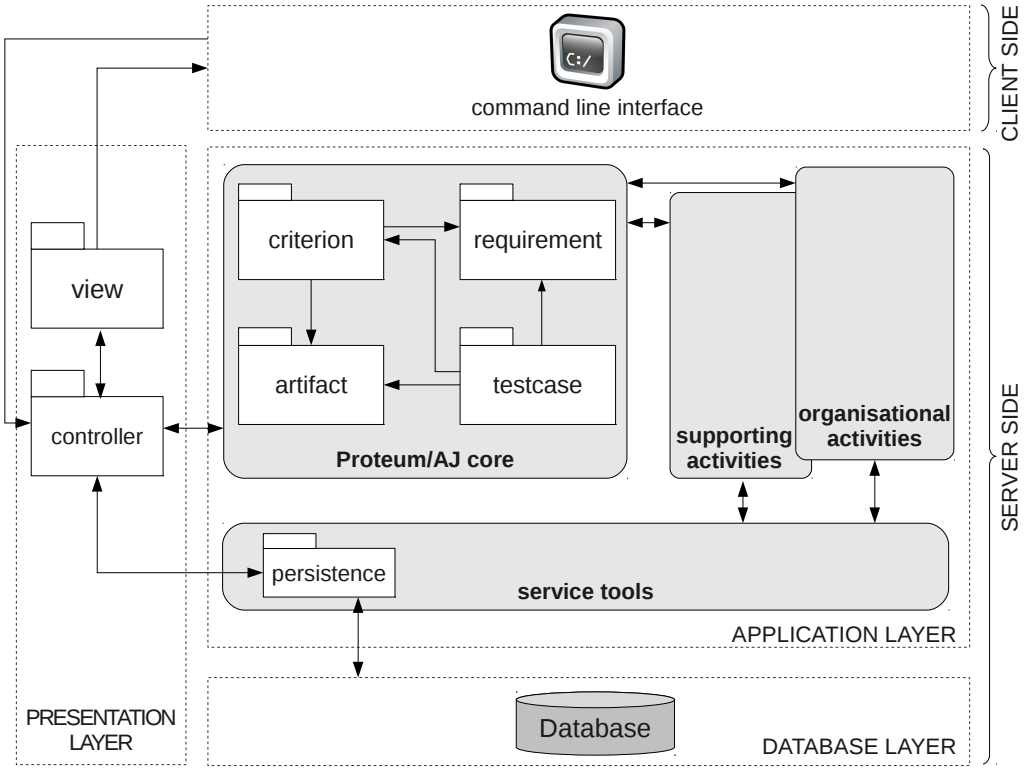


Figure 5.1: Proteum/AJ architecture – adapted from Ferrari et al. (2010c).

The modules currently implemented in *Proteum/AJ* are shown as UML packages in Figure 5.1. The core of the tool comprises the four main concepts that should be handled by testing tools, as proposed in *RefTEST* (Nakagawa et al., 2007): *testing criterion*, *artifact*, *test requirement* and *test case*. Some of them map directly to the requirements presented in Table 5.1. For instance, testcase maps to the *"test case handling"* requirement; and criterion maps to both *"mutant handling"* and *"adequacy analysis"* requirements. The former provides functionalities for running and managing test cases in *Proteum/AJ*, while the latter is responsible for handling tasks related to testing criterion itself (e.g. generating, compiling and analysing mutants).

The artifact and requirement modules comprise, respectively, the artefacts under test (i.e. the AspectJ source code files) and the test requirements (i.e. the generated mutants). The controller module is in charge of receiving requests from the client and properly invoking the modules present in the application layer, which include core functionalities and database-related procedures. The controller is also responsible for updating the view that is presented to the client. In *Proteum/AJ*, the view is basically formed by test execution feedback that is displayed to the users. More details about *Proteum/AJ*'s functionalities and core modules are provided in the next sections.

5.3 The Main Functionalities of Proteum/AJ

This section provides an overview of the main functionalities implemented in *Proteum/AJ*. The description that follows is based on the four basic steps of the mutation-based testing and on the tool's execution flow depicted in Figure 5.2.

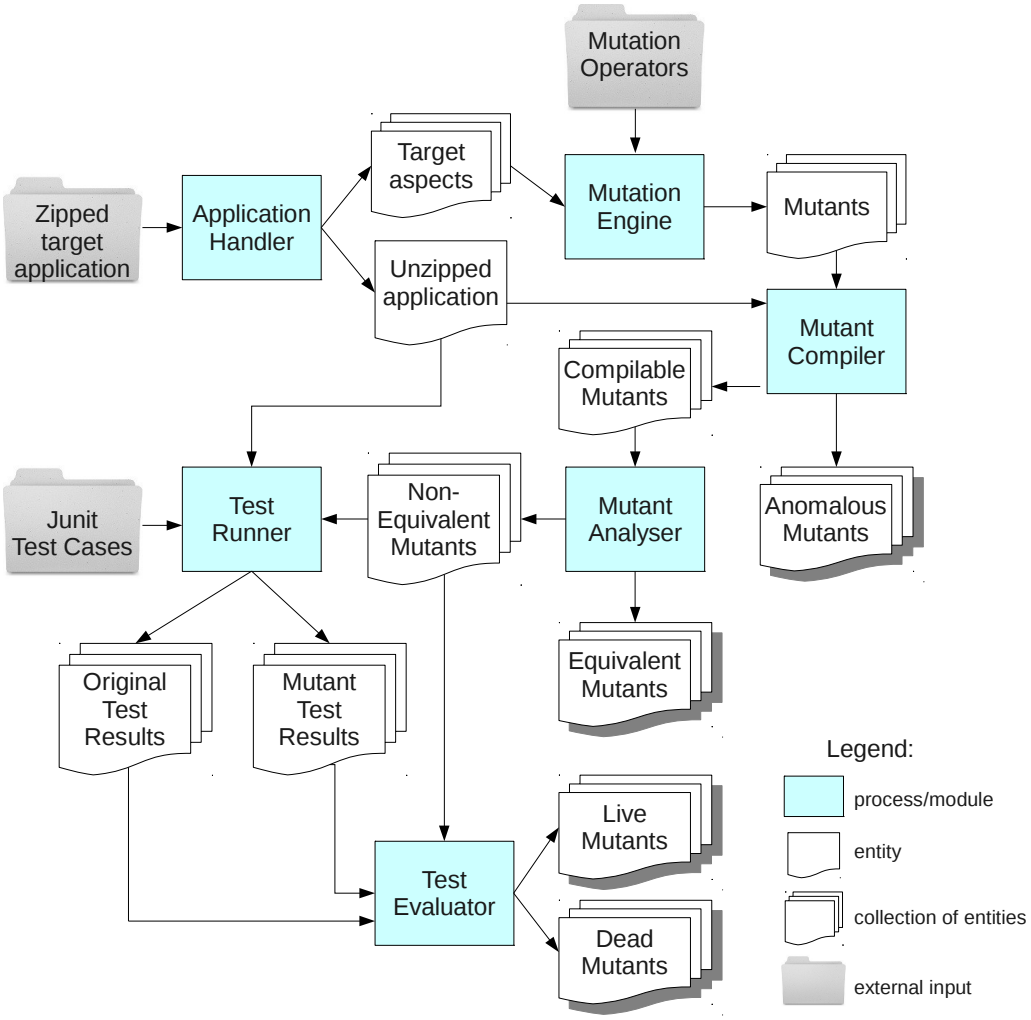


Figure 5.2: Proteum/AJ execution flow – adapted from Ferrari et al. (2010c).

Original program execution: *Proteum/AJ* initially receives a compressed file which includes the application under test and an initial test set. The application is pre-processed by the Application Handler module and forwarded to be executed by the Test Runner module. These combined steps create a test project and store the execution results into the *Proteum/AJ* database.

Mutant generation: The Application Handler module also sends the identified aspects to the Mutation Engine module, which is responsible for generating the mutants. The mutations are performed according to a list of mutation operators previously selected by the tester when the test project is created.

Mutant execution: Initially, the Mutant Compiler module compiles the mutants and detects non-compilable (i.e. *anomalous*) mutants. For compilable mutants, the weaving information produced by the AspectJ weaver is collected and further used by the Mutant Analyser module (see the *Mutant analysis* step next described). The compilable, non-equivalent mutants are executed by the Test Runner module, and the results are stored to be evaluated by the Test Evaluator module.

Mutant analysis: This step is handled in two phases by *Proteum/AJ*: (1) automatic detection of equivalent mutants after the mutant compilation performed by the Mutant Analyser module; and (2) manual analysis carried out by the tester, after the test evaluation performed by the Test Evaluator module. The automatic phase is restricted to PCD-related mutants and relies on the aspect-base code weaving information produced by the AspectJ compiler. The manual phase consists in the typical analysis of live mutants after the execution of tests and evaluation of results.

Note that testers can update test projects during subsequent test sessions. For example, they can add or remove mutation operators and target aspects, as well as import new test cases in order to augment the test coverage. The tool also produces mutant analysis reports that show the current mutation score and the mutated parts of the code for each mutant.

5.4 Implementation Details

Table 5.2 summarises some metrics[1] collected from the current version of *Proteum/AJ*. In total, the tool includes approximately 8,800 LOC, from which ∼700 consists of XML-based scripts implemented to map object-related data to tables in the database. It took around eight months to have an operational version of the tool, which was released in January, 2010, containing the the set of functionalities described in this chapter.

From the main tool packages (i.e. packages that have the "`core`" suffix), `criterion` is the largest one and is responsible for manipulating the source code in order to generate

[1]Metrics for Java code have been collected using the Metrics Eclipse plugin (The Eclipse Foundation, 2010c). LOC of iBATIS mapping scripts have been counted by hand.

the mutants. `controller` and `ui.cmdtool` are responsible for receiving and handling user requests (e.g. creating a new test project or running test cases). The `tests` package contains a set of *JUnit* test cases that exercise several parts of the tool, belonging both from the core and the utility classes.

Table 5.2: Proteum/AJ implementation effort.

Package	LOC	# of classes	# of methods	# of attributes
core	124	2	29	12
core.criterion	620	7	51	14
core.criterion.operators	2,024	25	117	5
core.requirement	91	1	18	6
core.testcase	578	4	38	12
core.artifact	140	1	21	11
controller	1,099	7	31	0
ui.cmdtool	1,093	9	9	0
utils	875	12	40	166
lib.exceptions	248	14	62	28
persistence	79	2	20	7
tests	1,126	29	95	2
iBATIS scripts	∼700	–	–	–
Total	8,797	113	531	263

Proteum/AJ utilises third-party software applications such as the *AspectJ-front* toolkit (Stratego Community, 2009), the iBATIS data mapper framework (iBATIS Development Team, 2009) and the *Ant* tool (The Apache Software Foundation, 2009). More details about the supporting technologies employed in *Proteum/AJ* can be found in Appendix D.

5.4.1 Core Modules

The main modules implemented in *Proteum/AJ* are depicted in Figure 5.3. These modules correspond to the core concepts – *criterion*, *requirement*, *artifact* and *testcase* – identified during the establishment of *RefTEST* (Nakagawa et al., 2007).

In *Proteum/AJ*, the criterion module encompasses functionalities for generating, compiling and analysing mutants. These functionalities are implemented within three main classes, namely `MutationEngine`, `MutantCompiler` and `MutantAnalyser`, whose execution flow was presented in the previous section. The `MutationEngine` class is responsible for generating mutants from AspectJ source code by controlling the application of the

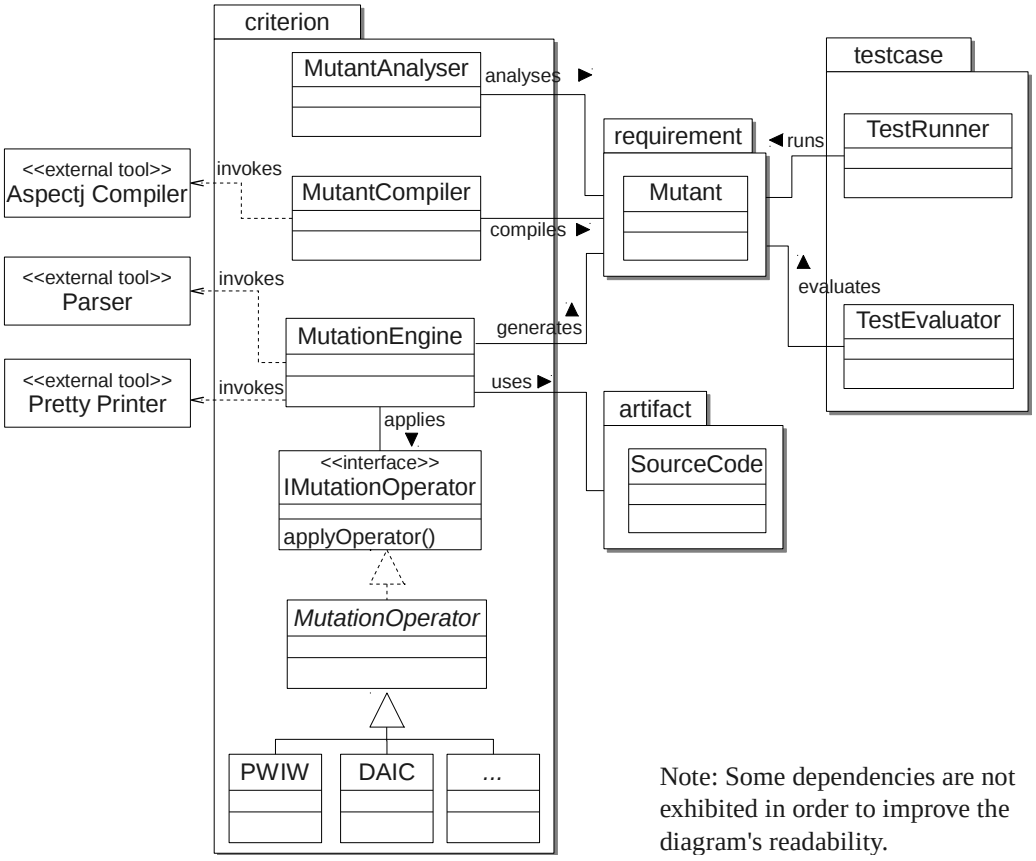


Figure 5.3: Proteum/AJ core modules – adapted from Ferrari et al. (2010c).

mutation operators. It automates 24 out of the 26 mutation operators[2] proposed in Chapter 4. The `MutationEngine` class parses the AspectJ source code into an abstract syntax tree, performs the mutations and invokes the pretty-printer tool to build AspectJ code for every generated mutant. After that, the `MutationCompiler` class invokes the AspectJ compiler, whose output is used by the `MutantAnalyser` class to compare the weaving information that results from the compilation of the original program and the mutants.

The testcase module manages the test execution and evaluation. Its `TestRunner` class implements test runner methods for the original application and the mutant. The `TestEvaluator` class evaluates the test results obtained from the execution of the original application and the mutants. It contrasts test case outputs, identifies the differences and decides whether a given mutant should be killed or not.

[2]The PWSR and PSSR operators (Ferrari et al., 2008) are not implemented in the current version of Proteum/AJ (Ferrari et al., 2010c). The modification rules proposed for these operators requires static, reflexive analysis of the application under test and are planed for the next releases of the tool.

5.5 Summary of Contributions and Limitations

The contributions of this chapter consist of the design and the implementation of a tool named *Proteum/AJ* that supports the mutation testing of AO programs. The tool automates the set of mutation operators defined in our previous research – described in Chapter 4 – and supports the basic steps of mutation testing, thus overcoming some limitations of other similar tools (Anbalagan and Xie, 2008; Delamare et al., 2009b).

Proteum/AJ is able to automatically identify equivalent PCD-related mutants based on the weaving information produced by the standard AspectJ compiler. It also supports incremental testing strategies by allowing the tester to manage – i.e. adding or removing – mutation operators, target aspects and test cases. All produced information is stored into a relational database, therefore the creation of statistical reports is also possible.

The main limitations of *Proteum/AJ* concern the supported mutation approach, the implementation of mutation operators and the mutant compilation step. *Proteum/AJ* relies on *JUnit* test cases (Beck and Gamma, 2010) to evaluate the mutants. Given that *JUnit* tests have the ability of configuring partial program runs (e.g. a single method execution), the evaluation of mutants based such kind of tests characterises the *firm mutation* approach defined by Woodward and Halewood (1988). This, however, might not be seen as a limitation as long as *JUnit* tests can be designed to address complete program executions, thus configuring a *strong mutation* test scenario (Marick, 1991).

Ideally, each mutation should result in a syntactically correct version of the program, since syntactic faults are very likely to be revealed during the compilation process. That is, a mutation should introduce a semantic fault into the programs under test. However, some operators proposed in Chapter 4 and implemented in *Proteum/AJ* are likely to produce non-compilable mutants (e.g. POAC, DSSR, ABAR and ABPR). Similar limitation has been faced by Offutt et al. (1996c) in a mutation system developed for Ada programs.

In regard to the compilation of mutants, the process supported by ajc, the standard AspectJ compiler, requires full weaving of classes and aspects in the occurrence of ITDs (The Eclipse Foundation, 2010b). Since ITDs are typically used in complex AO systems (Ferrari et al., 2010a), the mutant compilation step may become a bottleneck for the use of *Proteum/AJ* with large systems, in spite of possible optimisations allowed by ajc such as the weaving of existing class files. Consequently, reducing the compilation time is one of the enhancements that should be handled during the evolution of *Proteum/AJ*.

Chapter

6

Evaluating the Proposed Mutation Testing Approach

Devising well-founded and practical testing approaches is essential towards high quality software products. Such approaches provide means by which software engineers can reduce the number of faults present in the products, hence increasing the confidence that the software behaves as expected. Equally important, demonstrating their feasibility and effectiveness is fundamental to promote the technology transfer to the industry (Harrold, 2000). This can be achieved through studies that evaluate several properties of a testing approach like cost, effectiveness and strength (Wong, 1993).

Evaluation studies may help, for example, software practitioners to choose between one technique or another, according to budget constraints and the criticality of the ongoing software project. Despite the importance of performing this kind of studies, we can notice that most of the several AO testing approaches have not been properly evaluated to date. Only a few studies concerning evaluation of properties like cost and usefulness can be found in the literature (Lemos et al., 2009; Xie and Zhao, 2006).

This chapter describes two evaluation studies that involve the application of the mutation testing approach we proposed in this dissertation. The general goal is to demonstrate the feasibility of applying the approach to AO systems of varied size and complexity. The evaluation is based on the three following viewpoints: (i) the usefulness of the mutation

operators for simulating non-trivial faults; (ii) the effort required to produce adequate test sets; and (iii) the cost for applying the operators based on estimates.

The first two analyses, described in Section 6.1, comprise the evaluation of a set of small AO applications for which we designed and evolved test suites in order to obtain full coverage of mutants. The second study, described in Section 6.2, addresses the third viewpoint by estimating the cost for applying the approach to larger AO systems. Besides the aforementioned goals, the studies also show the feasibility of using the *Proteum/AJ* tool, earlier described in Chapter 5.

This chapter partially reproduces a paper submitted to the Elsevier's Science of Computer Programming journal. More specifically, it reproduces Sections 7, 8 and 9 of the referred paper, which also recapitulates the mutation testing approach defined in this dissertation. The full paper contents can be found in Appendix E.

6.1 First Study: Evaluating the Usefulness and Required Effort of the Proposed Approach

This section describes a study that evaluates the mutation-based testing approach proposed in Chapter 4 of this dissertation. The main goal of this study is checking the feasibility of the approach in terms or its usefulness and required effort. The evaluation procedures were planned and conducted in order to answer the two following questions: (i) *do the operators have the ability of simulating faults that cannot be detected by pre-existing, systematically derived test suites?* and (ii) *can the proposed mutation operators be applied at a feasible cost* [1] *?*

To answer these questions, we selected and thoroughly tested a set of AO applications based on a well-established functional-based testing approach. We evaluated the coverage obtained with this functional-based test data in regard to the generated mutants for each application. This initial phase addressed the first defined question. We then evolved the test sets to make them adequate to cover all non-equivalent mutants of the selected applications, thus addressing our second question. All procedures are described in the sequence, starting from an overview of the evaluated AO applications.

[1] Note that the cost of applying a testing criterion can include several variables such as the number of test requirements (e.g. mutants), the number of required test cases, and the effort required to compute unfeasible requirements (e.g. equivalent mutants). In this study, we evaluate the cost based on the number of derived test requirements and the number o required test cases do produce adequate test sets.

6.1.1 Target Applications

We selected 12 small AO applications upon which we performed our evaluation. All applications were identified from previous papers that describe AO testing approaches and evaluation. A short description of each of them is following presented. A summary of the size-related metrics of the selected applications is shown in Table 6.1.

Table 6.1: Values of metrics for the selected applications (first study).

Application	LOC[1]	Classes[2]	Aspects	PCDs	Advices	declare
1. BankingSystem	199	9	6	11	7	1
2. Telecom	251	6	3	6	7	1
3. ProdLine	537	8	8	10	10	4
4. FactorialOptimiser	39	1	1	2	3	0
5. MusicOnline	150	7	2	4	3	0
6. VendingMachine	64	1	3	6	6	1
7. PointBoundsChecker	44	1	1	4	4	0
8. StackManager	77	4	3	3	3	0
9. PointShadowManager	66	2	1	3	3	0
10. Math	53	1	1	1	1	0
11. AuthSystem	89	3	2	2	2	0
12. SeqGen	205	8	4	3	3	2
TOTAL	1774	51	35	55	52	9

[1]It considers only real lines of code, excluding comments and blank lines.

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

The BankingSystem application manages transactions for bank accounts (Laddad, 2003b). Aspects in BankingSystem implement logging, minimum balance control and overdraft operations.

Telecom is a telephony system simulator which is originally distributed with AspectJ (The Eclipse Foundation, 2010b). In Telecom, timing and billing of phone calls are handled by aspects. The version evaluated in this section extends the original implementation in order to support a different type of charging for mobile calls (Lemos et al., 2007).

ProdLine consists in a software product line for graph applications that includes a set of common functionalities of the graph domain (Lopez-Herrejon and Batory, 2002). Typical algorithms for graph manipulation are included, e.g. shortest-path between vertices, identification of strongly connected components and cycle checking. Aspects are used

in this application to introduce the features selected for a specific SPL instance. Each feature is implemented through one ore more aspects. AspectJ ITDs are intensively used together with a few PCD-advice pairs.

FactorialOptimiser is a math utility application that implements optimised factorial calculation (Alexander et al., 2004). The calculation is managed by an aspect; if the factorial for a given number has already been calculated, the aspects retrieve it to reduce overhead. Every calculated factorial is cached for reuse purposes.

The MusicOnline application manages an online music store that allows customers to play songs and playlists (Lemos et al., 2009). A customer needs to pay for each played song or playlist. Aspects in this application manage the customer's accounts and the billing system. Once a customer exceeds his credit limit, his account is suspended until he or she proceeds either a total or a partial payment.

VendingMachine consists in an application for a vending machine into which the customer inserts coins in order to get drinks (Liu and Chang, 2008). The aspects are responsible for controlling the sales operations (e.g. number of inserted coins and number of available drinks).

PointBoundsChecker is a two-dimension point constraint checker (Mortensen and Alexander, 2005). An aspect checks if the point coordinates conform to a specific range of values. If not, exceptions are raised.

StackManager implements a simple stack that provides the basic push and pop operations, which are supervised by aspects (Xie et al., 2005). The aspects avoid the insertion of negative values into the stack, perform audit on the stored elements and count the number of push operations.

PointShadowManager is an application for managing two-dimension point coordinates (Zhao, 2003). An aspect creates and manages shadows for point objects. When a point is created, its shadow has exactly the same coordinates. When a point coordinated is updated, the respective shadow's coordinate is added by a fixed offset.

Math is a math utility application that calculates the probability of successes in a sequence of n independent yes/no experiments (Bernoulli trial), each yielding success with probability p (Lemos et al., 2009). The aspect logs exponentiation operations, identifying the type of the exponent (integer or real).

AuthSystem is a simplified version of a banking system that requires user authentication before the execution of certain operations like debit and balance retrieval (Zhou et al., 2004). Furthermore, it monitors amount transfer between accounts by means of atomic transactions. The aspects are responsible for authentications and transaction management.

SeqGen implements a sequence generator of integers and chars values (Bernardi and Lucca, 2007). It includes two aspects that modularise the generation policy (random and Fibonacci sequences) and the logging concerns. Similarly to the ProdLine application, AspectJ ITDs are intensively used in SeqGen.

6.1.2 Building the Initial Test Sets

We applied the Systematic Functional Testing (SFT) criterion (Linkman et al., 2003) to build initial test sets for each selected application. The choice for the SFT criterion was motivated by the significant results reported in previous research (Linkman et al., 2003): test sets that covered all SFT-derived requirements – hereafter called *SFT-adequate* test sets or simply T_{SFT} – yielded high coverage of mutants generated with conventional mutation operators for C programs (Agrawal et al., 1989).

SFT combines two widely used functional-based testing criteria: Equivalence Partitioning and Boundary-Value Analysis (Myers et al., 2004, p. 52, p. 59). Basically, the main difference between SFT and the other two is that SFT requires two test cases for each equivalence class. In this way, one can avoid coincidental correctness possibly observed for a test input which "masks" a fault that could be uncovered by another test input from the same domain partition (Linkman et al., 2003).

For each of the target applications, we developed a test plan that specifies the equivalence classes and boundaries that should be covered at this initial testing phase. We defined the test requirements for every public operation from the classes that compose the application. Moreover, we defined test requirements for aspectual behaviour according to the description of the systems available in the original papers and reports (Alexander et al., 2004; Bernardi and Lucca, 2007; Laddad, 2003b; Lemos et al., 2009, 2007; Liu and Chang, 2008; Lopez-Herrejon and Batory, 2002; Mortensen and Alexander, 2005; Xie et al., 2005; Zhao, 2003; Zhou et al., 2004).

Figure 6.1 and Table 6.2 illustrate the definition of the equivalence classes and boundary values for a method extracted from the BankingSystem application. The `debit` method belongs to the `AccountSimpleImpl` class and is intended to perform debit operations in bank accounts, having as a constraint the current account balance that should never be lower than zero.

Building SFT-adequate test sets requires designing at least two test cases to cover each of the equivalence classes, as well as at least one test case that covers each of the boundary values. As suggested by Myers et al. (2004, p. 55), tests for valid classes can cover one or more of such classes, whereas individual tests should be created for each

```
1 public void debit(float amount) throws InsufficientBalanceException {
2   if (_balance < amount) {
3      throw new InsufficientBalanceException("Total balance not sufficient");
4   }
5   else {
6     _balance = _balance - amount;
7   }
8 }
```

Figure 6.1: Source code of the `debit` method (`AccountSimpleImpl` class).

Table 6.2: Equivalence classes and boundary values for the `debit` method.

Input Condition	Valid class	Invalid Class
`amount` parameter	(C1) amount <= current balance	(C2) amount > current balance
Output Condition	**Valid class**	**Invalid Class**
Resulting balance	(o1) balance = previous balance − debited amount	n/a
	(o2) balance = previous balance	n/a
Boundary values		
(C1) amount = current balance		
(C2) amount = current balance + 0.01		

invalid class. Considering the test plan for the `debit` method presented in Table 6.2, a SFT-adequate test set w.r.t. such method requires at least six test cases: four[2] test cases to cover the equivalence classes and two others to cover the boundary values. The results obtained with the execution of the SFT-adequate test sets in this study are detailed in the sequence.

Table 6.3 summarises the results achieved during this initial phase. Column 2 lists the number of test requirements derived for each application according to the Equivalence Partitioning criterion, and likewise does column 3 for Boundary-Value Analysis. Columns labelled with $|T_{EB}|$ and $|T_{SFT}|$ list the size of test sets which are adequate w.r.t., respectively, the two aforementioned traditional functional-based criteria and the SFT criterion (i.e. *SFT-adequate* test sets). Column 6 lists the increase percentage w.r.t. the size of the test sets when we evolved T_{EB} to T_{SFT}. Finally, Column 7 shows the number of faults revealed in each application. Such faults are related to either ordinary code (e.g. incorrect implemented logic) or AOP constructs (e.g. incorrect PCD definition) and have been all fixed before we started the mutation testing phase that is next described.

[2]Note that a single test case can cover classes C1 and o1, while another test case can cover classes C2 and o2. Therefore, two other test cases are sufficient to fulfil the SFT criterion w.r.t. the equivalence classes.

Table 6.3: Functional-based test requirements and respective adequate test sets.

Application	Equivalence Classes	Boundaries	$\|T_{EB}\|$	$\|T_{SFT}\|$	$T_{EB} \rightarrow T_{SFT}$	Faults
1. BankingSystem	36	10	34	58	71%	1
2. Telecom	49	2	37	63	70%	6
3. ProdLine	23	6	20	36	80%	0
4. FactorialOptimiser	10	6	15	19	27%	0
5. MusicOnline	43	2	25	46	84%	0
6. VendingMachine	13	5	12	18	50%	1
7. PointBoundsChecker	10	6	9	14	56%	0
8. StackManager	12	3	9	15	67%	2
9. PointShadowManager	12	2	6	12	100%	0
10. Math	26	38	41	53	29%	1
11. AuthSystem	16	6	12	17	42%	1
12. SeqGen	46	15	22	39	77%	0
TOTAL	296	101	242	390	61%	12

6.1.3 Applying Mutant Analysis to the Target Applications

We applied the 24 mutation operators implemented in the *Proteum/AJ* tool to the 12 applications of our study. The summary of the mutant generation step is displayed in Table 6.4. Columns 2–4 represent the three groups of mutation operators, namely G1 (related to PCDs), G2 (related to `declare`-like expressions) and G3 (related to advices), together with the associated number of mutants per application. Table 6.4 also includes the number of equivalent mutants that have been automatically identified by *Proteum/AJ* (column 6), the number of anomalous – i.e. non-compilable – mutants (column 7) and the number of mutants that remained alive (column 8).

Note that *Proteum/AJ* allows the tester to enable or disable the option for automatic detection of equivalent mutants. The mutation operators that are eligible for such automatic procedure are all listed in Section 6.2 of this chapter (see Table 6.7). They are related to PCDs and are expected to yield the largest mutant set amongst the three groups of operators (Ferrari et al., 2008), from which a high percentage represent equivalent ones[3]. Indeed, the figures presented in Table 6.4 reveal that nearly 67% of the mutants produced by operators from G1 were automatically detected as equivalent. Besides that, around 9% of mutants are anomalous. We can also observe a very small number of mutants

[3]High percentages of PCD-related equivalent mutants have been observed by Delamare et al. (2009a).

Table 6.4: Mutants generated for the 12 target applications.

Application	Mut. G1	Mut. G2	Mut. G3	Total	Autom. Equiv.	Anom.	Alive
1. BankingSystem	108	2	26	136	68	18	50
2. Telecom	82	2	27	111	46	12	53
3. ProdLine	158	0	41	199	125	16	58
4. FactorialOptimiser	14	0	15	29	8	6	15
5. MusicOnline	47	0	10	57	25	5	27
6. VendingMachine	82	2	29	113	58	8	47
7. PointBoundsChecker	46	0	24	70	32	10	28
8. StackManager	34	0	11	45	24	0	21
9. PointShadowManager	38	0	12	50	25	4	21
10. Math	16	0	4	20	13	0	7
11. AuthSystem	45	0	7	52	28	3	21
12. SeqGen	33	0	7	40	19	3	18
TOTAL	703	6	213	922	471	85	366

produced by operators from G2, mainly due to the rare use of `declare`-like expressions in the selected applications.

In order to evaluate the mutant coverage yielded by the T_{SFT} test sets and augment such coverage, for each application we performed the following sequence of steps: (1) execution of the live mutants on the respective T_{SFT} test set; (2) calculation of the initial mutation score; (3) manual identification of equivalent mutants; (4) calculation of the intermediate mutation score; (5) design of new test cases to kill the remaining live mutants, producing the mutation-adequate (T_M) test sets (i.e. mutation score of value 1.0).

The obtained results after performing the six steps just described are summarised in Table 6.5. In this table, columns 2–5 list, respectively, the initial number of live mutants, the number of mutants killed with the execution of T_{SFT}, the remaining number of live mutants after the execution of T_{SFT}, and the initially achieved mutation score. For those applications whose achieved mutation score was lower than one (i.e. at least one mutant was still alive), columns 6 and 7 show the number of equivalent mutants identified by hand and the updated mutation score. Finally, columns 8–10 present, respectively, the number of test cases added to T_{SFT} in order to obtain the T_M test set, the size of T_M and the final mutation score. Note that mutation scores of value 1.0 are not repeated in subsequent columns of the table. The results are analysed in the sequence.

Table 6.5: Mutation testing results for the 12 target applications.

Application	Initial Alive	Killed by T_{SFT}	Remain Alive	Initial MS	Man. Equiv.	Interm. MS	Added Tests	$\|T_M\|$	Final MS
1.BankingSystem	50	50	0	1.00	–	–	–	58	–
2.Telecom	53	31	22	0.58	10	0.72	4	67	1.00
3.ProdLine	58	58	0	1.00	–	–	–	36	–
4.FactorialOptimiser	15	14	1	0.93	1	1.00	–	19	–
5.MusicOnline	27	22	5	0.81	2	0.88	2	48	1.00
6.VendingMachine	47	23	24	0.49	13	0.68	5	23	1.00
7.PointBoundsChecker	28	28	0	1.00	–	–	–	14	–
8.StackManager	21	21	0	1.00	–	–	–	15	–
9.PointShadowManager	21	13	8	0.62	5	0.81	2	14	1.00
10.Math	7	4	3	0.57	2	0.80	1	54	1.00
11.AuthSystem	21	17	4	0.81	1	0.85	2	19	1.00
12.SeqGen	18	4	14	0.22	8	0.40	3	42	1.00
TOTAL	366	285	81	0.78	42	0.88	19	409	1.00

6.1.4 Analysis of the Results

Comparing SFT with mutation testing: According to the results presented in the previous section, in particular in Table 6.5, the T_{SFT} test sets have yielded high[4] mutant coverage for five applications: $MS = 1.00$ for BankingSystem, ProdLine, PointBoundsChecker and StackManager, and $MS = 0.93$ for FactorialOptimiser.

The mutation scores yielded by the T_{SFT} test sets can be observed in Figure 6.2. The chart depicts the achieved mutant coverage at two different moments: before and after the manual identification of equivalent mutants. The column labelled with *Initial* represents the first case, i.e. the achieved coverage right after the execution of the T_{SFT} test sets. The *All Equiv* column represents the coverage after we performed the analysis of live mutants and classified all the equivalent ones. The chart also includes a column to represent the goal, i.e. the full mutant coverage.

These obtained results provide evidence on the relevance of the mutation operators' ability in simulating faults that cannot be easily revealed by existing, non-mutation-based test sets. In this study, we applied the SFT criterion, which combines widely used

[4]Note that the definition of a threshold value for the goodness of the mutation score (e.g. high or low) depends on the criticality of the program under evaluation. Nevertheless, mutation score values above 0.95 have typically been considered high in previous research (Barbosa et al., 2001; Offutt et al., 1996a). In this study, we use a threshold value of 0.9; therefore, $MS \geq 0.9$ is considered high.

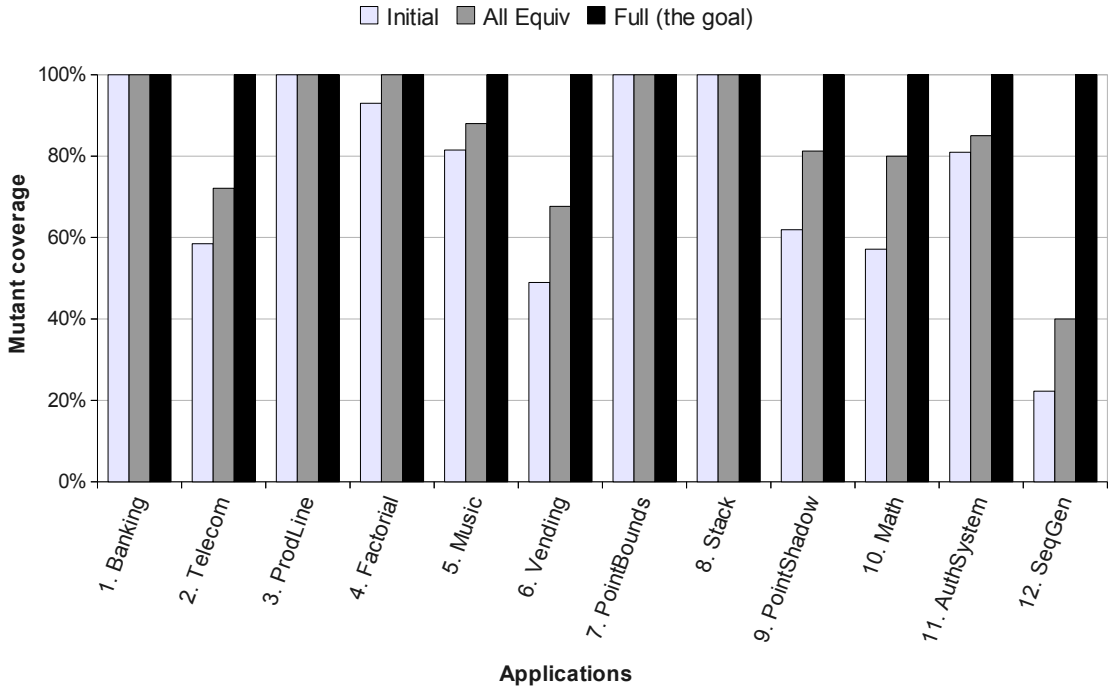


Figure 6.2: Coverage yielded by SFT test sets.

functional-based testing criteria and has shown to be strong w.r.t mutant detection rate (Linkman et al., 2003). However, the *SFT-adequate* test sets were able to achieve high mutation score for only 5 out of 12 evaluated applications.

Effort required to achieve mutation-adequate test sets: In regard to the effort required to obtain the T_M test sets (i.e. test sets that result in mutation scores of 1.0), apart from the FactorialOptimiser application, whose mutation score reached the value of 1.0 after the manual equivalent mutant detection step, the other 7 applications required test set increments. This is depicted in Figure 6.3, which shows the differences in size of T_{SFT} and T_M test sets for all evaluated applications.

Considering the initial and final test set cardinalities (i.e. $|T_{SFT}|$ and $|T_M|$) (see Tables 6.3 and 6.5), in total, 19 test cases were designed to kill the mutants that remained alive after the mutant execution and identification of equivalent mutants steps. It means that, on average, the increase to produce the $|T_M|$ test sets was nearly 5%. From this viewpoint, we can conclude that the application of the proposed mutation operators does not overwhelm the testers while enhancing the existing, systematically derived test sets to achieve high mutant coverage.

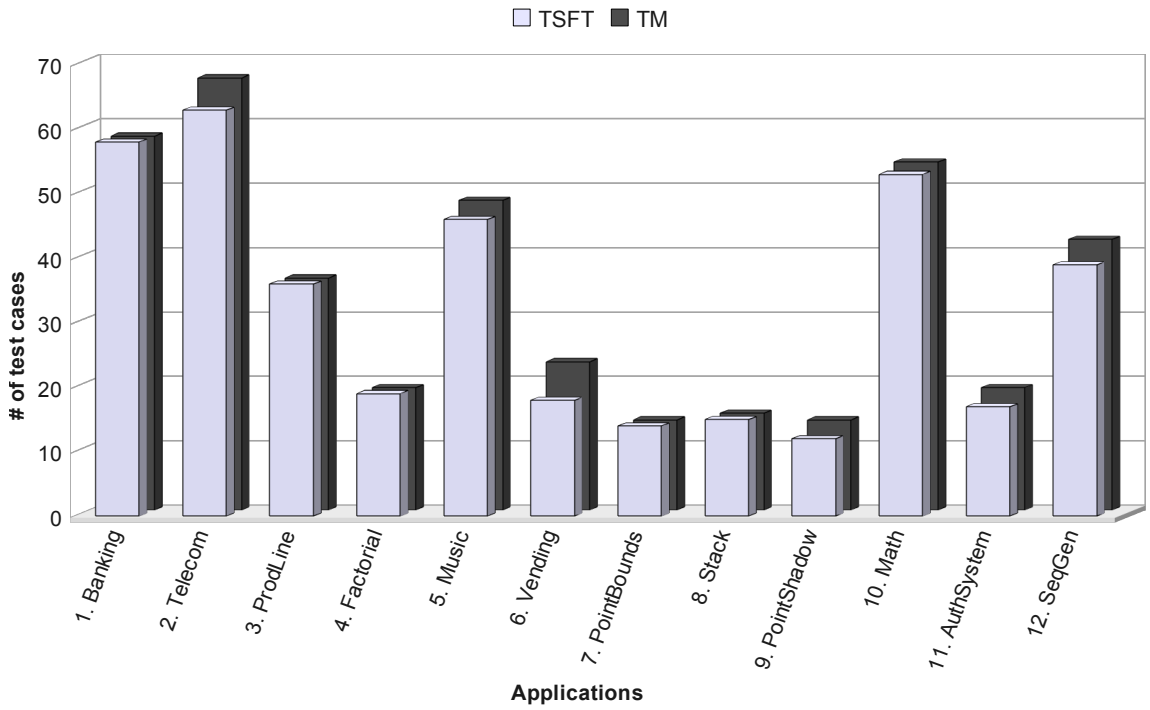


Figure 6.3: Effort required to derive the T_M test sets.

6.1.5 Additional Comments on the Mutant Analysis Step

In an AO system, a simple mutation can have wide impact on the woven application. Consequently, many times analysing a mutant required us to scan several modules of the application in order to realise the real impact of the mutation on the application. However, even though a mutation can impact on the quantification of JPs, the behaviour of the woven application may remain the same.

For example, Figure 6.4 shows a mutant produced by the PWIW for the MusicOnline application. The mutation consisted in replacing a naming part of the PCD (i.e. the `owed` attribute) with the "`*`" wildcard. This modification results in an additional JP selection in the based code, hence this mutant has not been automatically classified as equivalent by *Proteum/AJ*. Nonetheless, during the mutant analysis step, we noticed that the final behaviour of MusicOnline was not altered so that this mutant was manually classified as equivalent.

Note that the analysis of live mutants depends on the current configuration of the application under test (Lemos et al., 2006). That is, while a PCD p may be correct w.r.t. to a given base program P_1, p may "misbehave" (i.e. match a different set of JPs) when it is applied into a different base program P_2. This observation also holds for advices as well as for PCD-advice pairs.

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

Figure 6.4: Example of an equivalent mutant of the MusicOnline application.

6.2 Second Study: Estimating the Cost of the Approach with Larger Systems

This section presents a second study that aims to estimate the cost of applying the AspectJ mutation operators in systems larger than the ones presented in the previous section. This second study comprises four medium-sized AO systems from different application domains, and the achieved results are compared to the results presented in Section 6.1 of this chapter. We start by describing the four analysed systems. The results are presented in the sequence.

6.2.1 Target Systems

The four systems we evaluated in this study are called iBATIS, TollSystemDemonstrator, HealthWatcher and MobileMedia. These systems have already been evaluated within the academic and industrial context (AOSD Europe, 2010; Ferrari et al., 2010a; Figueiredo et al., 2008; Greenwood et al., 2007; Soares et al., 2006) and employ mainstream industrial technologies in their implementations. Table 6.6 lists some size-related metric values for the four systems. Note that these implementations are significantly larger than the applications evaluated in Section 6.1. This allows us to better estimate the cost of applying the mutation operators, in particular w.r.t. the number of generated mutants, including equivalent and anomalous ones. A short description of each system is following presented.

iBATIS (iBATIS Development Team, 2009) is a Java-based open source framework for object-relational data mapping. The iBATIS AO versions (Ferrari et al., 2010a) have some functional and non-functional concerns modularised within aspects (e.g. exception handling, concurrency and type mapping). The TollSystemDemonstrator (TSD) system includes a subset of requirements of a real-world tolling system. It has been developed in the context of the AOSD-Europe Project (AOSD Europe, 2010) and contains functional and non-functional concerns implemented within aspects (e.g. charging variabilities, distribution and logging). HealthWatcher (HW) is a Web-based application that allows citizens to register complaints regarding health issues (Greenwood et al., 2007; Soares et al., 2006).

Table 6.6: Values of metrics for the selected applications (second study).

Application	Aprox. KLOC[1]	Classes	Aspects	PCDs	Advices	declare
iBATIS	11	207	41	97	95	69
TollSystemDemonstrator	4	98	25	37	37	6
HealthWatcher	7	137	26	57	47	14
MobileMedia	3	45	22	65	60	26
TOTAL	24	487	114	256	239	115

[1] It considers only real lines of code, excluding comments and blank lines.

Some aspectised concerns in HealthWatcher are distribution, persistence and exception handling. Finally, MobileMedia (MM) (Figueiredo et al., 2008) is a software product line for mobile devices that allows users to manipulate image files in different mobile devices. In MM, aspects are used to configure the product line instances, enabling the selection of alternative and optional features.

Apart from TollSystemDemonstrator, which has a single release, the other three systems have several releases available for evaluation. In this study, we selected versions 01, 10 and 06 of iBATIS, HealthWatcher and MobileMedia, respectively. For more information about each of them, the reader may refer to the respective placeholder websites or to previous reports of these systems (AOSD Europe, 2010; Ferrari et al., 2010a; Figueiredo et al., 2008; Greenwood et al., 2007).

6.2.2 Generating Mutants for the Target Systems

Applying the 24 mutation operators implemented in *Proteum/AJ* resulted in the numbers of mutants presented in Tables 6.7 and 6.8. Such tables focus on, respectively, the numbers of equivalent and anomalous mutants produced per operator. The two rightmost columns of the tables show the total number of generated mutants for all systems and the respective percentages of equivalent and anomalous mutants. Note that Table 6.7 only includes mutation operators which are eligible in regard to automatic detection of equivalent mutants. Furthermore, a blank cell in any of the tables means that no value could be assigned to that category since no mutant has been generated by the respective operator.

When we consider the overall number of mutants produced per operator, we can see in Table 6.8 that three mutation operators (namely PSDR, DEWC and AJSC) generated no mutants for any of the systems. This indicates that AspectJ constructs targeted by these

Table 6.7: Percentages of equivalent mutants generated for the target systems.

	iBatis AO01		TollSystem		HW AO10		MM AO06		All Apps	
Operator	Total	Equiv.	Total	Equiv.	Total	Equiv.	Total	Equiv.	Total	Equiv.
PWIW	1976	92%	449	92%	646	87%	1075	96%	4146	92%
PWAR	0	–	4	0%	0	–	0	–	4	0%
PSWR	4	0%	5	20%	37	68%	0	–	46	57%
PSDR	0	–	0	–	0	–	0	–	0	–
POPL	193	63%	60	77%	70	74%	117	91%	440	74%
POAC	185	0%	41	0%	63	0%	75	9%	364	2%
POEC	19	58%	0	–	0	–	0	–	19	58%
PCTT	15	100%	21	81%	21	86%	39	85%	96	86%
PCGS	1	0%	1	0%	0	–	0	–	2	0%
PCCR	121	2%	17	0%	43	0%	120	32%	301	13%
PCLO	222	0%	30	37%	24	0%	21	29%	297	6%
PCCC	0	–	8	100%	1	100%	0	–	9	100%
TOTAL	2736	72%	636	78%	905	72%	1447	84%	5724	76%

operators are not present in any of the evaluated versions of those systems. In regard to mutants produced by the PCCC operator, all were automatically classified as equivalent. However, such mutants should be manually revised given that the JP selections by the `cflow` and `cflowbelow` PCDs are evaluated at runtime in order to decide whether every specific JP selection holds of not within the running control flow context.

In the next section, we compare the obtained numbers of equivalent and anomalous mutants with the results obtained in our first study earlier presented in Section 6.1. We also estimate the effort required for developing mutation-adequate test sets based on the previously observed results.

6.2.3 Contrasting the Results with the First Study

From Tables 6.7 and 6.8, we can observe that for all systems the obtained numbers of equivalent and anomalous mutants exceed the averages in our first study. This is graphically shown in Figure 6.5.

Considering the applications evaluated in Section 6.1, the average number of equivalent mutants was 67%, while this value for the four larger systems analysed in this section represents 76% of the total. When it comes to anomalous mutants, the average amount for the larger systems is 19% against 9% for the smaller applications.

Table 6.8: Percentages of anomalous mutants generated for the target systems.

	iBatis AO01		TollSystem		HW AO10		MM AO06		All Apps	
Operator	Total	Anom.	Total	Anom.	Total	Anom.	Total	Anom.	Total	Anom.
PWIW	1976	1%	449	1%	646	4%	1075	2%	4146	2%
PWAR	0	–	4	0%	0	–	0	–	4	0%
PSWR	4	25%	5	0%	37	5%	0	–	46	7%
PSDR	0	–	0	–	0	–	0	–	0	–
POPL	193	22%	60	3%	70	3%	117	6%	440	12%
POAC	185	96%	41	7%	63	5%	75	44%	364	59%
POEC	19	0%	0	–	0	–	0	–	19	0%
PCTT	15	0%	21	0%	21	0%	39	0%	96	0%
PCCE	153	50%	66	0%	52	23%	94	27%	365	31%
PCGS	1	0%	1	0%	0	–	0	–	2	0%
PCCR	121	7%	17	0%	43	5%	120	43%	301	21%
PCLO	222	40%	30	57%	24	17%	21	48%	297	40%
PCCC	0	–	8	0%	1	0%	0	–	9	0%
DAPC	0	–	0	–	1	0%	176	89%	177	88%
DAPO	0	–	0	–	1	0%	6	0%	7	0%
DSSR	69	81%	0	–	14	100%	20	100%	103	87%
DEWC	0	–	0	–	0	–	0	–	0	–
DAIC	8	0%	8	0%	0	–	0	–	16	0%
ABAR	123	48%	23	4%	52	4%	48	23%	246	30%
APSR	16	13%	25	24%	10	0%	41	32%	92	23%
APER	0	–	4	75%	0	–	4	0%	8	38%
AJSC	0	–	0	–	0	–	0	–	0	–
ABHA	95	0%	40	0%	47	0%	60	0%	242	0%
ABPR	405	37%	68	91%	146	52%	250	80%	869	56%
TOTAL	3605	19%	870	11%	1228	12%	2146	25%	7849	19%

Let us consider a test set T_C that is adequate to an AO system as large as the ones evaluated in this study. Moreover, T_C is adequate w.r.t. to a given criterion C other than Mutant Analysis (e.g. T_C is derived from either a functional-based or structural-based test selection criterion). Let us also consider (i) the observed number of equivalent and anomalous mutants listed in Tables 6.7 and 6.8, and; (ii) the effort required to create the T_M test sets in our previous study. We can therefore assume that the effort testers need

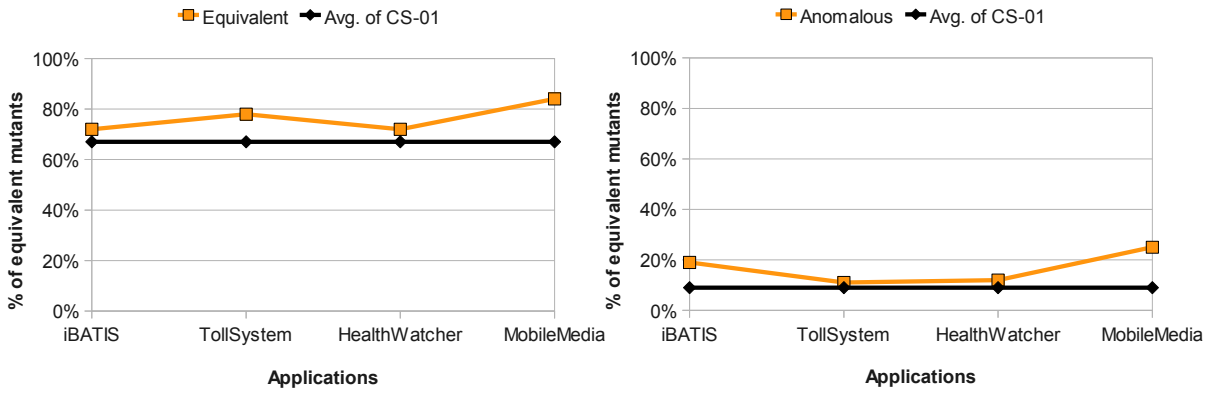


Figure 6.5: Percentages of equivalent and anomalous mutants for the target systems.

to dedicate to evolve T_C to cover all remaining live mutants after the execution of T_C shall be equal or less than 5% of $|T_C|$.

The above estimate relies on the fact that the systems evaluated in this section have a smaller proportion of live mutants to be handled than the systems evaluated in the previous study. Even though this a rough estimate, according to the results observed in our first study we argue that well-designed test suites are able to reveal most of the faults simulated by the mutation operators. Nevertheless, a small but not less important test set increment is still needed in order to yield trustworthy results in regard to the Mutant Analysis criterion.

6.3 Study Limitations

The main threat to the validity of our evaluation studies – and, consequently, to the achieved results – regards the representativeness of the selected applications, specially in our first study (described in Section 6.1). Such study comprised a set of small AO applications upon which we performed the evaluation procedures and drew our conclusions. It required us to undertake a full – thus time consuming – test process, starting from the test plan design and ending up with the creation of adequate test suites that yielded full mutant coverage for all applications. The limited size of the applications allowed us to produce detailed analysis of the systems, comprehensive test plans and subsequent test data. Note that performing such a number of tasks from scratch for larger systems might have been prohibitive specially due to time constraints. Besides that, similar studies in regard to the size and the number of selected applications have been performed in order to evaluate other testing approaches for AO software (Lemos et al., 2009; Xie and Zhao, 2006; Xu and Rountev, 2007).

Another possible threat to the validity of our first study concerns the way we developed the initial test suites. Functional-based testing, as a black-box technique (Myers et al., 2004, p. 9), relies on specification documents to derive the test requirements. However, the lack of detailed specifications for the applications evaluated in that study led us to follow a "reverse" process to derive the *SFT-adequate* test sets. Instead of building our test plans and the respective test sets based exclusively on the specifications, we were forced to analyse the source code in order to comprehend the functionalities of each application. Therefore, the design of some test cases has also been guided by internal (structural) elements of the code rather than solely by functional requirements. As a consequence, the produced test sets may have led to higher mutant coverage given that, traditionally, structural-based (i.e. white-box) test suites tend to outperform black-box ones (Mathur, 2007, p. 482-483).

In regard to our second study (described in Section 6.2), our conclusions are based on estimates derived from the set of generated mutants and on the results achieved in the previous study. In spite of the observed trend regarding the number of generated mutants and on the estimated effort, more definite conclusions are only possible after running similar procedures with a larger set of applications. Exercising such applications with concrete test data will also help us draw more well-founded conclusions on the required effort to fulfil our testing approach requirements.

6.4 Final Remarks

Considering that Aspect-Oriented Programming is still a developing research topic, to date we can only find limited evidence with respect to the evaluation and assessment of related testing approaches. In the systematic mapping study described in Chapter 2, we were able to identify a few pieces of work that regard the evaluation of AO testing, all based on small sets of AO applications (Lemos et al., 2009; Lemos and Masiero, 2010; Xie and Zhao, 2006). This chapter contributed in this context, describing the results of two studies that evaluated the set of mutation operators proposed in this dissertation. The studies also enabled us to demonstrate the feasibility of using the *Proteum/AJ* tool, earlier described in Chapter 5.

In regard to the evaluation of test selection techniques, studies that compare the strength of varied testing criteria can be commonly found in the literature, for both procedural and object-oriented programs. For example, we can find studies that compare mutation testing with other test selection criteria such control flow- and data flow-based (Li et al., 2009; Mathur and Wong, 1994), and functional-based testing (Linkman et al., 2003).

However, to the best of our knowledge, in the context of AO testing, the study presented in Section 6.1 is the first initiative to compare the strength of two testing criteria derived from different testing techniques (i.e. functional and fault-based testing). In spite of that, there is still a need for other similar studies in order to either confirm or eventually contradict the achieved results.

Chapter

7

Conclusions

Innovative software development approaches such as Aspect-Oriented Programming (AOP) claim to bring several advantages in regard to both internal and external software quality attributes. Even so, software practitioners cannot indeed benefit from such advantages if one cannot guarantee the software fulfils the minimum quality standards that lead to its successful execution. Verification and Validation activities, in particular software testing, play a fundamental role in this context, helping to reduce the number of faults present in the software throughout the development and maintenance processes.

In order to deal with testing-related specificities of contemporary programming techniques, at first the software engineers need to realise the impact of such techniques on the correctness of the produced software. In other words, the engineers need to understand how software faults can occur in practice so that they can be either avoided or rapidly localised within the software. In this context, mutation testing is a widely explored test selection criterion that focuses on recurring faults observed in the software. Starting from a well-characterised set of fault types, mutation testing helps to demonstrate the absence of such faults in the evaluated software products.

When it comes to AOP, we can identify a variety of testing approaches tailored for aspect-oriented (AO) software, as well as a number of reports on fault characterisation. To date, however, the few initiatives for customising the mutation testing for AO programs show either limited coverage with respect to the range of simulated faults or a need for

proper evaluation in regard to attributes like application cost and effectiveness. Likewise the fault characterisation reports neither cover the full range of mechanisms that support AOP nor have been evaluated regarding their ability in categorising faults.

This thesis contributed in this sense, proposing a mutation-based testing approach for AO programs. The achievements include the definition and evaluation of a fault taxonomy for AO software, the customisation of the criterion for the context of AO programs, the development of adequate tool support, and the evaluation of the approach. The achieved results support an affirmative answer to the general research question defined in Chapter 1, that is, *fault-based testing can be applied to AO software in a systematically way and at a practicable cost.*

The set of contributions that allowed us to answer our research question are revisited in the next section. In the sequence, Section 7.2 summarises the limitations of the work presented along this dissertation and how we plan to overcome these limitations in our future research.

7.1 Revisiting the Thesis Contributions

This section revisits the achievements of this thesis in terms of three kinds of contributions: (i) theoretical definitions; (ii) implementation of automated support; and (iii) evaluation studies. Each category is following summarised. Note that the contributions next described help to overcome the research limitations highlighted in the final remarks of Chapter 2 with respect to fault taxonomies and fault-based testing of AO software.

7.1.1 Theoretical Definitions

Definition of a fault taxonomy for AO software: we defined a fault taxonomy that encompasses 26 different fault types distributed over four main categories, namely PCD-related, ITD-related, advice-related and base program-related (Chapter 3, Section 3.2). The taxonomy overcomes limitations of previous taxonomies in terms of completeness and detailing of fault type descriptions, and has been further evaluated based on a fault set extracted from several available AO systems (more details in Section 7.1.3).

Definition of mutation operators for AO programs: we defined a set of mutation operators for AspectJ programs to simulate instances of the fault types described in the taxonomy (Chapter 4, Section 4.1). The operators are grouped according to three categories of target elements: PCDs, AspectJ declare-like expressions and

advices. They avoid modelling faults that result in errors at compilation time (e.g. incorrect class member introductions). A preliminary analysis showed that the operators tend to generate a manageable amount of mutants, which has been confirmed in a further evaluation study (more details in Section 7.1.3).

7.1.2 Implementation of Automated Support

Design and implementation of a mutation testing tool for AO programs: we specified and implemented a tool named *Proteum/AJ* that supports the mutation testing of AspectJ programs (Chapter 5). *Proteum/AJ* automates our proposed set of operators and enables the incremental execution of test sessions, hence providing support for varied testing strategies. The output produced by the tool – i.e. the mutant set, test comparison results and analysis reports – is stored into a database, which facilitates the creation of statistical reports. Results achieved with the support of *Proteum/AJ* are described in the next section.

7.1.3 Evaluation Studies

Evaluation of the fault-proneness of AO programs: we performed an evaluation of the fault-proneness of AO programs based on fault-related data gathered from several releases of evolving AO systems (Chapter 3, Section 3.1). This data supported the evaluation of two hypotheses that concerned the obliviousness property and the main AOP mechanisms. The results show that the controversial obliviousness property has negative impact on the correctness of AO programs. Furthermore, the major AOP constructs (namely PCDs, advices and ITDs) pose similar risk in regard to introduction of faults into the programs.

Evaluation of the fault taxonomy for AO software: we categorised the fault set extracted from evolving AO systems according to the fault taxonomy proposed in this thesis (Chapter 3, Section 3.2). The results provide evidence on the taxonomy's completeness and allowed us to spot and characterise recurring faulty implementation scenarios that should be either avoided or verified during the development of AO software.

Analysis of the fault taxonomy generalisation: we analysed to which extent the fault taxonomy proposed in this thesis can be generalised to other mainstream AOP technologies beyond AspectJ (Chapter 4, Section 4.2). The conclusion is that they are similarly prone to occurrences of most of the described fault types. Therefore,

existing testing approaches for AspectJ programs may be adapted to fit specific needs with respect to those technologies.

Evaluation of the proposed mutation operators: supported by the *Proteum/AJ* tool, we applied the mutation operators to two distinct sets of AO systems (Chapter 6, Sections 6.1 and 6.2). The achieved results show that applying the mutation operators to AO programs does not impose high costs, even though it is likely to require increases of 5% in the test suites in order to achieve full mutant coverage. These studies helped to demonstrate the feasibility of fault-based testing for AO programs with adequate tool support.

Systematic Mapping Study of AO testing approaches: we presented an up-to-date systematic map of the research on AO testing (Chapter 2, Section 2.3). The results underlay the definition of the fault taxonomy and, in turn, the design of the mutation operators for AspectJ programs. Moreover, the results helped us to keep aware of the latest developments in the field.

7.2 Limitations and Future Work

This section describes limitations of this thesis' contributions and how we plan to tackle them in the future. Note that every previous chapter of this dissertation has already discussed pertinent limitations. Therefore, we hereby focus on more general limitations that should be overcome as well as possible improvements to be handled during our short and medium term planned research. We conclude pointing out future directions for developments in the field of testing of AO programs.

Evaluation of the fault taxonomy: the evaluation of the fault taxonomy defined in Chapter 3 consisted in the categorisation of a fault set gathered from several releases of three AO systems. This limited number of sources threats the generalisation of the achieved results. Therefore, as long as more fault-related data from other AO systems becomes available, be it gathered by ourselves or by third-party researchers, we plan to re-run our evaluation. This will help us to revisit and re-evaluate the taxonomy with the aim of gaining more confidence in regard to its completeness, in turn promoting its adoption in other software production contexts, both academic and industrial.

The proposed set of mutation operators: the set of mutation operators described in Chapter 4 covers a range of fault types included in the taxonomy. However,

according to the nature of the target systems, new operators may be required in order to reflect possible domain-specific faults. In this context, we plan to go through every fault reported in the studies presented in Chapter 3 with the goal of realising if new mutation operators can be devised from recurring faulty scenarios. Additionally, running other evaluation studies similar to the ones presented in Chapter 6 shall help us pinpoint refinements to the mutation operator set and draw more general conclusions about the mutation operators' effectiveness and efficacy.

Alternative mutation testing approaches: the evaluation studies presented in Chapter 6 involved the application of all mutation operators implemented in *Proteum/AJ*. Even though the number of produced mutants has shown to be manageable, in those studies we have not explored cost reduction techniques for mutation testing. Examples of such techniques are the Constrained Mutation (Mathur and Wong, 1993) and Selective Mutation (Offutt et al., 1993). They all focus on reducing the number of mutants by applying a subset of the mutation operators, although still keeping the efficacy of the criterion in regard to the quality of the derived test suites. We aim to explore these cost reduction techniques in our future research. In doing so, we shall be able to identify sufficient mutation operator sets (Barbosa et al., 2001; Offutt et al., 1996a), therefore making our approach even more applicable in practice.

Tool support: the *Proteum/AJ* tool, described in Chapter 5, does not implement two mutation operators proposed in Chapter 4. Such operators – namely PWSR and PSSR – require reflexive analysis of the target application's class hierarchy in order to produce the intended mutants. We still need to adapt such task in the tool's execution flow, thus achieving support for the whole set of proposed operators. Furthermore, we plan to evolve *Proteum/AJ* to provide support for traditional mutation operators at unit (Agrawal et al., 1989), interface (Delamaro et al., 2001) and class (Ma et al., 2002) levels. Once we have these new features available, we will be able to perform a wide range of experiments, which may address comparative studies regarding OO and AO systems and shall underlie the definition of incremental testing strategies.

7.2.1 Possible Research Directions

As discussed in the final remarks of Chapter 3, AOP is yet a maturing area, whose several facets in regard to the assessment of derived products still require substantial effort to become the state of the practice in the industry. This thesis contributed in this direction,

presenting a fault-based testing approach that can be systematically applied at a feasible cost. In this closing section, we list two possible research topics whose outcomes should ameliorate two major difficulties we faced while developing our research: (i) the lack of well-established benchmarks to support the evaluation of our approach, and (ii) the little availability of shared knowledge and supporting tools for AO testing. They are briefly discussed in the sequence.

Building test benchmarks: building a benchmark of applications for the evaluation of existing and upcoming AO testing approaches would contribute for the development of AOP in general. Although building benchmarks should be handled according to specific usage contexts (e.g. based on the kind of question that should be answered) (Runeson et al., 2008), this would anyway help to fill the current gap related to the lack of evaluations of AO testing approaches. The outcomes of such evaluations would support software practitioners to figure out the most suitable testing approaches to be applied in specific software development scenarios.

Development of a common testing framework: creating a framework that integrates several testing approaches and provides adequate tool support seems to be another interesting research opportunity. Obviously, this would depend on the community making available the individual contributions (e.g. tools, documentation and expertise) in order to synergically compose this framework. For example, the reference architecture upon which the *Proteum/AJ* tool relies – the *RefTEST*: Reference Architecture for Software Testing Tools (Nakagawa et al., 2007) – promotes the integration of software engineering tools by means of standardised interfaces and functionalities. Therefore, *RefTEST* sounds to be a good candidate to provide the needed basis for creating the suggested framework.

References

Agrawal, H. Dominators, super blocks, and program coverage. In: *Proceedings of the 1st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, Portland/Oregon - USA: ACM Press, p. 25–34, 1994.

Agrawal, H.; DeMillo, R. A.; Hathaway, R.; Hsu, W.; Hsu, W.; Krauser, E. W.; Martin, R. J.; Mathur, A. P.; Spafford, E. H. *Design of mutant operators for the C programming language.* Technical Report SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette/IN - USA, 1989.

Alexander, R. T.; Bieman, J. M.; Andrews, A. A. *Towards the systematic testing of aspect-oriented programs.* Tech. Report CS-04-105, Dept. of Computer Science, Colorado State University, Fort Collins/Colorado - USA, 2004.

Anbalagan, P. Automated testing of pointcuts in aspectj programs. In: *Dynamic Languages Symposium (DLS) - held in conjunction with the ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, Portland/OR - USA: ACM Press, p. 758–759, 2006.

Anbalagan, P.; Xie, T. 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*, Raleigh/NC -USA: Kluwer Academic Publishers, p. 51–56, 2006.

Anbalagan, P.; Xie, T. Automated generation of pointcut mutants for testing pointcuts in AspectJ programs. In: *Proceedings of the 19th International Symposium on Software Reliability Engineering (ISSRE)*, Seattle/WA - USA: IEEE Computer Society, p. 239–248, 2008.

ANDREWS, J. H.; BRIAND, L. C.; LABICHE, Y. Is mutation an appropriate tool for testing experiments? In: *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, St. Louis/MO - USA: ACM Press, p. 402–411, 2005.

AOSD EUROPE Project Home Page. `http://www.aosd-europe.net/` - last accessed on 21/09/2010, 2010.

ARACIC, I.; GASIUNAS, V.; MEZINI, M.; OSTERMANN, K. An overview of CaesarJ. In: RASHID, A.; AKŞIT, M., eds. *Transactions on Aspect-Oriented Software Development I*, chapter 5, Springer-Verlag, p. 135–173 (LNCS v.3880), 2006.

ARAÚJO, J.; MOREIRA, A. An aspectual use-case driven approach. In: *VIII Jornadas de Ingeniería de Software y Bases de Datos (JISBD)*, Alicante - Spain: Thompson (Spain), p. 12–14, 2003.

ARAÚJO, J.; MOREIRA, A.; BRITO, I.; RASHID, A. Aspect-oriented requirements with UML. In: *Workshop on Aspect-oriented Modeling with UML (UML)*, Dresden - Germany, 2002.

AVGUSTINOV, P.; CHRISTENSEN, A. S.; HENDREN, L.; KUZINS, S.; LHOTÁK, J.; LHOTÁK, O.; DE MOOR, O.; SERENI, D.; SITTAMPALAM, G.; TIBBLE, J. abc: an extensible AspectJ compiler. In: *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD)*, Chicago/IL - USA: ACM Press, p. 87–98, 2005.

BABU, C.; KRISHNAN, H. R. Fault model and test-case generation for the composition of aspects. *SIGSOFT Software Engineering Notes*, v. 34, n. 1, p. 1–6, 2009.

BADRI, M.; BADRI, L.; BOURQUE-FORTIN, M. Generating unit test sequences for aspect-oriented programs: Towards a formal approach using UML state diagrams. In: *Enabling Technologies for the New Knowledge Society: Proceedings of the ITI 3rd International Conference on Information & Communications Technology (ICICT)*, Cairo - Egypt: IEEE Computer Society, p. 237–253, 2005.

BÆKKEN, J. S. *A fault model for pointcuts and advice in AspectJ programs.* MSc Dissertation, School of Electrical Engineering and Computer Science, Washington State University, Pullman/WA - USA, 2006.

BÆKKEN, J. S.; ALEXANDER, R. T. A candidate fault model for aspectj pointcuts. In: *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE)*, Raleigh/NC - USA: IEEE Computer Society, p. 169–178, 2006a.

Bækken, J. S.; Alexander, R. T. Towards a fault model for AspectJ programs: Step 1 - pointcut faults. In: *Proceedings of the 2nd Workshop on Testing Aspect Oriented Programs (WTAOP) - held in conjunction with ISSTA*, p. 1–6, 2006b.

Baniassad, E.; Clarke, S. Theme: An approach for aspect-oriented analysis and design. In: *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, Edinburgh - UK: IEEE Computer Society, p. 158–167, 2004.

Barbosa, E. F.; Maldonado, J. C.; Vincenzi, A. M. R. Toward the determination of sufficient mutant operators for C. *The Journal of Software Testing, Verification and Reliability*, v. 11, n. 2, p. 113–136, 2001.

Basili, V. R.; Perricone, B. T. Software errors and complexity: An empirical investigation. *Communications of the ACM*, v. 27, n. 1, p. 42–52, 1984.

Beck, K.; Gamma, E. JUnit cookbook. Online, `http://junit.sourceforge.net/doc/cookbook/cookbook.htm` - last accessed on 18/10/2010, 2010.

Beizer, B. *Software testing techniques*. 2nd ed. New York/NY - USA: Van Nostrand Reinhold, 1990.

Bernardi, M. Reverse engineering of aspect oriented systems to support their comprehension, evolution, testing and assessment. In: *Proceedings of the Doctoral Symposium of 12th European Conference on Software Maintenance and Reengineering (CSMR)*, Athens - Greece: IEEE Computer Society, p. 290–293, 2008.

Bernardi, M. L.; Di Lucca, G. A. Testing coverage criteria for aspect-oriented programs. *Software Quality Professional*, v. 10, n. 4, p. 27–38, 2008.

Bernardi, M. L.; Lucca, G. A. D. 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)*, Lisbon - Portugal: IEEE Computer Society, p. 65–76, 2007.

Binder, R. V. *Testing object-oriented systems: Models, patterns and tools*. 1st ed. Reading/MA - USA: Addison Wesley, 1999.

Biolchini, J.; Mian, P. G.; Natali, A. C. C.; Travassos, G. H. *Systematic review in software engineering*. Tech. Report RT-ES 679/05, Systems Engineering and Computer Science Dept., COPPE/UFRJ, Rio de Janeiro/RJ - Brazil, 2005.

Bonér, J.; Vasseur, A. Aspectwerkz - plain Java AOP. Online, eclipseCon 2005: `http://aspectwerkz.codehaus.org/` - last accessed on 21/09/2010, 2005.

Booch, G. *Object-oriented analysis and design with applications.* 2nd. ed. Redwood City/CA - USA: Addison Wesley, 1994.

Budd, T. A. *Mutation analysis of program test data.* PhD Thesis, Graduate School, Yale University, New Haven, CT - USA, 1980.

Burke, B.; Brock, A. Aspect-Oriented Programming and JBoss. Online, `http://onjava.com/pub/a/onjava/2003/05/28/aop_jboss.html` - last accessed on 21/09/2010, 2003.

Burrows, R.; Ferrari, F. C.; Garcia, A.; Taïani, F. An empirical evaluation of coupling metrics on aspect-oriented programs. In: *ICSE Workshop on Emerging Trends in Software Metrics (WETSoM)*, Cape Town - South Africa: ACM Press, p. 53–58, 2010a.

Burrows, R.; Ferrari, F. C.; Lemos, O. A. L.; Garcia, A.; Taïani, F. The impact of coupling on the fault-proneness of aspect-oriented programs: An empirical study. In: *Proceedings of the 21th International Symposium on Software Reliability Engineering (ISSRE)*, San Jose/CA - USA, (to appear), 2010b.

Cafeo, B.; Masiero, P. C. Teste estrutural de integração contextual de programas orientados a objetos e a aspectos. In: *Proceedings of the 4th Latin American Workshop on Aspect-Oriented Software Development (LAWASP)*, Salvador/BA - Brazil: Brazilian Computer Society, (in Portuguese), p. 25–30, 2010.

Ceccato, M.; Tonella, P. Measuring the effects of software aspectization. In: *Proceedings of the 1st Workshop on Aspect Reverse Engineering (WARE)*, Delft - The Netherlands, 2004.

Ceccato, M.; Tonella, P.; Ricca, F. 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, 2005.

Chavez, C. V. F. G. *A model-driven approach for aspect-oriented design.* PhD Thesis, Informatics Department, Pontifical Catholic University (PUC-Rio), 2004.

Chen, H. Y.; Tse, T. H.; Chan, F. T.; Chen, T. Y. In black and white: An integrated approach to class-level testing of object-oriented programs. *ACM Transactions on Software Engineering and Methodololgy (TOSEM)*, v. 7, n. 3, p. 250–295, 1998.

Chidamber, S. R.; Kemerer, C. F. A metrics suite for object oriented design. *IEEE Transactions on Software Engineering*, v. 20, n. 6, p. 476–493, 1994.

Chitchyan, R.; Rashid, A.; Rayson, P.; Waters, R. Semantics-based composition for aspect-oriented requirements engineering. In: *Proceedings of the 6th International Conference on Aspect-Oriented Software Development (AOSD)*, Vancouver - Canada: ACM Press, p. 36–48, 2007.

Coady, Y.; Kiczales, G. Back to the future: A retroactive study of aspect evolution in operating system code. In: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, Boston/MA - USA: ACM Press, p. 50–59, 2003.

Coelho, R.; Lemos, O. A. L.; Ferrari, F. C.; Masiero, P. C.; von Staa, A. On the robustness assessment of aspect-oriented programs. In: *Proceedings of the 3rd Workshop on Assessment of Contemporary Modularization Techniques (ACoM) - held in conjunction with OOPSLA*, Orlando/FL - USA, 2009.

Coelho, R.; Rashid, A.; Garcia, A.; Ferrari, F.; Cacho, N.; Kulesza, U.; von Staa, A.; Lucena, C. Assessing the impact of aspects on exception flows: An exploratory study. In: *Proceedings of the 22nd European Conference on Object-Oriented Programming (ECOOP)*, Paphos - Cyprus: Springer-Verlag, p. 207–234 (LNCS v.5142), 2008a.

Coelho, R.; Rashid, A.; Kulesza, U.; von Staa, A.; Lucena, C.; Noble, J. Exception handling bug patterns in aspect-oriented programs. In: *Proceedings of the 15th Conference on Pattern Languages of Programs (PLoP)*, Chicago/IL - USA, 2008b.

Delamare, R.; Baudry, B.; Ghosh, S.; Le Traon, Y. A test-driven approach to developing pointcut descriptors in aspectj. In: *Proceedings of the 2nd International Conference on Software Testing, Verification and Validation (ICST)*, Denver/CO - USA: IEEE Computer Society, p. 376–385, 2009a.

Delamare, R.; Baudry, B.; Le Traon, Y. AjMutator: A tool for the mutation analysis of aspectj pointcut descriptors. In: *Proceedings of the 4th International Work-*

shop on Mutation Analysis (Mutation), Denver/CO - USA: IEEE Computer Society, p. 200–204, 2009b.

DELAMARO, M. E. *Interface mutation: An interprocedural adequacy criterion for integration testing.* PhD Thesis, Physics Institute of São Carlos (IFSC), University of São Paulo, (in Portuguese), 1997.

DELAMARO, M. E.; MALDONADO, J. C. Proteum: A tool for the assessment of test adequacy for C programs. In: *Conference on Performability in Computing Systems (PCS)*, New Brunswick/NJ - USA, p. 79–95, 1996.

DELAMARO, M. E.; MALDONADO, J. C.; MATHUR, A. P. Interface Mutation: An approach for integration testing. *IEEE Transactions on Software Engineering*, v. 27, n. 3, p. 228–247, 2001.

DEMILLO, R. A.; LIPTON, R. J.; SAYWARD, F. G. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, v. 11, n. 4, p. 34–43, 1978.

DEMILLO, R. A.; MATHUR, A. P. *A grammar based fault classification scheme and its application to the classification of the errors of TEX.* Technical report, Software Engineering Research Center and Department of Computer Sciences - Purdue University, 1995.

DIJKSTRA, E. W. *A discipline of programming.* Englewood Cliffs/NJ - USA: Prentice-Hall, 1976.

DO, H.; ROTHERMEL, G. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering*, v. 32, n. 9, p. 733–752, 2006.

EADDY, M.; AHO, A.; HU, W.; MCDONALD, P.; ; BURGER, J. Debugging aspect-enabled programs. In: *Proceedings of the 6th International Symposium on Software Composition (SC)*, Braga - Portugal, 2007.

ELRAD, T.; FILMAN, R. E.; BADER, A. Aspect-oriented programming: Introduction. *Communications of the ACM*, v. 44, n. 10, p. 29–32, 2001a.

ELRAD, T.; KICZALES, G.; AKŞIT, M.; LIEBERHER, K.; OSSHER, H. Discussing aspects of AOP. *Communications of the ACM*, v. 44, n. 10, p. 33–38, 2001b.

Endress, A. An analysis of errors and their causes in systems programs. *IEEE Transactions on Software Engineering*, v. 2, p. 140–149, 1978.

Fabbri, S. C. P. F.; Maldonado, J. C.; Masiero, P. C.; Delamaro, M. E. Mutation analysis testing for finite state machines. In: *Proceedings of 5th International Symposium on Software Reliability Engineering (ISSRE)*, Monterey/CA - USA: IEEE Computer Society, p. 220–229, 1994.

Fabbri, S. C. P. F.; Maldonado, J. C.; Sugeta, T.; Masiero, P. C. Mutation testing applied to validate specifications based on statecharts. In: *Proceedings of 10th International Symposium on Software Reliability Engineering (ISSRE)*, Boca Raton/FL - USA: IEEE Computer Society, p. 210–219, 1999.

Ferrari, F. C.; Burrows, R.; Lemos, O. A. L.; Garcia, A.; Figueiredo, E.; Cacho, N.; Lopes, F.; Temudo, N.; Silva, L.; Soares, S.; Rashid, A.; Masiero, P.; Batista, T.; Maldonado, J. C. An exploratory study of fault-proneness in evolving aspect-oriented programs. In: *Proceedings of the 32nd International Conference on Software Engineering (ICSE)*, Cape Town - South Africa: ACM Press, p. 65–74, 2010a.

Ferrari, F. C.; Burrows, R.; Lemos, O. A. L.; Garcia, A.; Maldonado, J. C. Characterising faults in aspect-oriented programs: Towards filling the gap between theory and practice. In: *Proceedings of the 24th Brazilian Symposium on Software Engineering (SBES)*, Salvador/BA - Brazil: IEEE Computer Society, p. 50–59, 2010b.

Ferrari, F. C.; Höhn, E. N.; Maldonado, J. C. Testing aspect-oriented software: Evolution and collaboration through the years. In: *Proceedings of the 3rd Latin American Workshop on Aspect-Oriented Software Development (LAWASP)*, Fortaleza/CE - Brazil: Brazilian Computer Society, `http://www.cin.ufpe.br/~lawasp09/papers_aceitos.htm`, p. 24–30, 2009.

Ferrari, F. C.; Maldonado, J. C. Uma revisão sistemática sobre teste de software orientado a aspectos. In: *Proceedings of the 3rd Brazilian Workshop on Aspect-Oriented Software Development (WASP) - in conjunction with SBES*, Florianópolis/SC - Brasil, (in Portuguese), p. 101–110, 2006.

Ferrari, F. C.; Maldonado, J. C. *Teste de software orientado a aspectos: Uma revisão sistemática.* Technical Report 291, ICMC/USP, São Carlos/SP - Brasil, (in Portuguese), 2007.

Ferrari, F. C.; Maldonado, J. C. Experimenting with a multi-iteration systematic review in software engineering. In: *Proceedings of the 5th Experimental Software Engineering Latin American Workshop (ESELAW)*, Salvador/BA - Brazil, 2008.

Ferrari, F. C.; Maldonado, J. C.; Rashid, A. Mutation testing for aspect-oriented programs. In: *Proceedings of the 1st International Conference on Software Testing, Verification and Validation (ICST)*, Lillehammer - Norway: IEEE Computer Society, p. 52–61, 2008.

Ferrari, F. C.; Nakagawa, E. Y.; Rashid, A.; Maldonado, J. C. Automating the mutation testing of aspect-oriented Java programs. In: *Proceedings of the 5th ICSE International Workshop on Automation of Software Test (AST)*, Cape Town - South Africa: ACM Press, p. 51–58, 2010c.

Figueiredo, E.; Cacho, N.; Sant'Anna, C.; Monteiro, M.; Kulesza, U.; Garcia, A.; Soares, S.; Ferrari, F.; Khan, S.; Castor Filho, F.; Dantas, F. Evolving software product lines with aspects: An empirical study on design stability. In: *Proceedings of the 30th International Conference on Software Engineering (ICSE)*, Leipzig - Germany: ACM Press, p. 261–270, 2008.

Filman, R. E.; Elrad, T.; Clarke, S.; Akşit, M. Introduction. In: Filman, R. E.; Elrad, T.; Clarke, S.; Akşit, M., eds. *Aspect-Oriented Software Development*, chapter 1, Boston: Addison-Wesley, p. 21–35, 2004.

Filman, R. E.; Friedman, D. Aspect-oriented programming is quantification and obliviousness. In: *Workshop on Advanced Separation of Concerns - held in conjunction with OOPSLA*, Minneapolis - USA, p. 21–35, 2000.

Filman, R. E.; Friedman, D. Aspect-oriented programming is quantification and obliviousness. In: Filman, R. E.; Elrad, T.; Clarke, S.; Akşit, M., eds. *Aspect-Oriented Software Development*, chapter 2, Boston: Addison-Wesley, p. 21–35, 2004.

Franchin, I. G.; Lemos, O. A. L.; Masiero, P. C. Pairwise structural testing of object and aspect-oriented Java programs. In: *Proceedings of the 21st Brazilian Symposium on Software Engineering (SBES)*, Gramado/RS - Brazil: Brazilian Computer Society, 2007.

Frankl, P. G.; Weiss, S. N.; Hu, C. All-uses vs mutation testing: an experimental comparison of effectiveness. *Journal of Systems and Software*, v. 38, n. 3, p. 235–253, 1997.

Frankl, P. G.; Weyuker, E. J. An applicable family of data flow testing criteria. *IEEE Trans. Softw. Eng.*, v. 14, n. 10, p. 1483–1498, 1988.

Frankl, P. G.; Weyuker, E. J. Testing software to detect and reduce risk. *Journal of Systems and Software*, v. 53, n. 3, p. 275–286, 2000.

Gal, A.; Schröder-Preikschat, W.; Spinczyk, O. AspectC++: Language proposal and prototype implementation. In: *Proceedings of the Workshop on Advanced Separation of Concerns in Object-Oriented Systems*, Tampa Bay/Florida - USA, 2001.

Gong, M.; Muthusamy, V.; Jacobsen, H. AspeCt-Oriented C tutorial. Online, `http://research.msrg.utoronto.ca/ACC/Tutorial` - last accessed on 21/09/2010, 2006.

Greenwood, P.; Bartolomei, T.; Figueiredo, E.; Dosea, M.; Garcia, A.; Cacho, N.; Sant'Anna, C.; Soares, S.; Borba, P.; Kulesza, U.; Rashid, A. On the impact of aspectual decompositions on design stability: An empirical study. In: *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP)*, Berlin - Germany: Springer Berlin, p. 176–200 (LNCS v.4609), 2007.

Griswold, W. G.; Sullivan, K.; Song, Y.; Shonle, M.; Tewari, N.; Cai, Y.; Rajan, H. Modular software design with crosscutting interfaces. *IEEE Software*, v. 23, n. 1, p. 51–60, 2006.

Gybels, K.; Brichau, J. Arranging language features for more robust pattern-based crosscuts. In: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, Boston/MA - USA: ACM Press, p. 60–69, 2003.

Gyimóthy, T.; Ferenc, R.; Siket, I. Empirical validation of object-oriented metrics on open source software for fault prediction. *IEEE Transactions on Software Engineering*, v. 31, n. 10, p. 897–910, 2005.

Harrold, M. J. Testing: A roadmap. In: *Proceedings of the Conference on the Future of Software Engineering - held in conjunction with ICSE*, Limerick - Ireland: ACM Press, p. 61–72, 2000.

HARROLD, M. J.; ROTHERMEL, G. Performing data flow testing on classes. In: *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering (FSE)*, New York, NY: ACM Press, p. 154–163, 1994.

HILSDALE, E.; HUGUNIN, J. Advice weaving in AspectJ. In: *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD)*, Lancaster - UK: ACM Press, p. 26–35, 2004.

HIRSCHFELD, R. AspectS - AOP with Squeak. In: *Workshop on Advanced Separation of Concerns in Object-Oriented Systems (Position Paper) - held in conjunction with OOPSLA*, Tampa Bay/Florida - USA, 2001.

HOFFMAN, K.; EUGSTER, P. Bridging Java and AspectJ through explicit join points. In: *Proceedings of the 5th International Symposium on Principles and Practice of Programming in Java (PPPJ)*, Lisbon - Portugal: ACM Press, p. 63–72, 2007.

HORGAN, J. R.; MATHUR, A. P. Assessing testing tools in research and education. *IEEE Software*, v. 9, n. 3, p. 61–69, 1992.

HOWDEN, W. E. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, v. 8, n. 4, p. 371–379, 1982.

IBATIS DEVELOPMENT TEAM Apache iBATIS home page. Online, `http://attic.apache.org/projects/ibatis.html` - last accessed on 21/09/2010, 2009.

IEEE *IEEE standard glossary of software engineering terminology.* Standard 610.12, Institute of Electric and Electronic Engineers, New York/NY - USA, 1990.

JIA, Y.; HARMAN, M. An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, (in press), 2010.

JOHNSON, R.; HOELLER, J.; ARENDSEN, A.; SAMPALEANU, C.; HARROP, R.; RISBERG, T.; DAVISON, D.; KOPYLENKO, D.; POLLACK, M.; TEMPLIER, T.; VERVAET, E.; TUNG, P.; HALE, B.; COLYER, A.; LEWIS, J.; LEAU, C.; EVANS, R. *Spring - Java/J2EE application framework.* Reference Manual Version 2.0.6, Interface21 Ltd., 2007.

KICZALES, G.; HILSDALE, E.; HUGUNIN, J.; KERSTEN, M.; PALM, J.; GRISWOLD, W. G. Getting started with AspectJ. *Communications of the ACM*, v. 44, n. 10, p. 59–65, 2001a.

Kiczales, G.; Hilsdale, E.; Hugunin, J.; Kersten, M.; Palm, J.; Griswold, W. G. An overview of AspectJ. In: *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP)*, Budapest - Hungary: Springer-Verlag, p. 327–353 (LNCS v.2072), 2001b.

Kiczales, G.; Irwin, J.; Lamping, J.; Loingtier, J.-M.; Lopes, C.; Maeda, C.; Menhdhekar, A. Aspect-oriented programming. In: *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP)*, Jyväskylä - Finland: Springer-Verlag, p. 220–242 (LNCS v.1241), 1997.

Kiczales, G.; Mezini, M. Aspect-oriented programming and modular reasoning. In: *Proceedings of the 27th International Conference on Software Engineering (ICSE)*, ACM Press, p. 49–58, 2005.

Kienzle, J.; Yu, Y.; Xiong, J. On composition and reuse of aspects. In: *Proceedings of the 2nd Foundations of Aspect-Oriented Languages Workshop (FOAL) - held in conjunction with AOSD*, Boston/MA - USA: ACM Press, 2003.

Kim, H. *AspectC#: An AOSD implementation for C#.* MSc Dissertation, Department of Computer Science - Trinity College/The University of Dublin, Dublin - Ireland, 2002.

Kitchenham, B. *Procedures for performing systematic reviews.* Joint Technical Report TR/SE-0401 (Keele) - 0400011T.1 (NICTA), Software Engineering Group - Department of Computer Science - Keele University and Empirical Software Engineering - National ICT Australia Ltd, Keele/Staffs-UK and Eversleigh-Australia, 2004.

Kitchenham, B. A.; Dybå, T.; Jørgensen, M. Evidence-based software engineering. In: *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, Edinburgh - Scotland: IEEE Computer Society, p. 273–281, 2004.

Kumar, N.; Sosale, D.; Konuganti, S. N.; Rathi, A. Enabling the adoption of aspects - testing aspects: A risk model, fault model and patterns. In: *Proceedings of the 8th International Conference on Aspect-Oriented Software Development (AOSD)*, Charlottesville/VA - USA: ACM Press, p. 197–206, 2009.

Laddad, R. Aspect-oriented programming will improve quality. *IEEE Software*, v. 20, n. 6, p. 90–91, 2003a.

Laddad, R. *AspectJ in action.* Greenwich/CT - USA: Manning Publications, 2003b.

LEMOS, O. A. L.; FERRARI, F. C.; MASIERO, P. C.; LOPES, C. V. Testing aspect-oriented programming pointcut descriptors. In: *Proceedings of the 2nd Workshop on Testing Aspect Oriented Programs (WTAOP) - held in conjunction with ISSTA*, Portland/Maine - USA: ACM Press, p. 33–38, 2006.

LEMOS, O. A. L.; FRANCHIN, I. G.; MASIERO, P. C. Integration testing of object-oriented and aspect-oriented programs: A structural pairwise approach for Java. *Science of Computer Programming*, v. 74, n. 10, p. 861–878, 2009.

LEMOS, O. A. L.; MALDONADO, J. C.; MASIERO, P. C. Data flow integration testing criteria for aspect-oriented programs. In: *Proceeding of the 1st Brazilian Workshop on Aspect-Oriented Software Development (WASP)*, Brasília/DF - Brazil, 2004a.

LEMOS, O. A. L.; MALDONADO, J. C.; MASIERO, P. C. Structural unit testing of AspectJ programs. In: *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs (WTAOP) - held in conjunction with AOSD*, Chicago/IL - USA, 2005.

LEMOS, O. A. L.; MASIERO, P. C. Integration testing of aspect-oriented programs: a structural pointcut-based approach. In: *Proceedings of the 22nd Brazilian Symposium on Software Engineering (SBES)*, Campinas/SP - Brazil: Brazilian Computer Society, p. 49–64, 2008a.

LEMOS, O. A. L.; MASIERO, P. C. Using structural testing to identify unintended join points selected by pointcuts in aspect-oriented programs. In: *Proceedings of the 2008 32nd Annual IEEE Software Engineering Workshop (SEW)*, Kassandra - Greece: IEEE Computer Society, p. 84–93, 2008b.

LEMOS, O. A. L.; MASIERO, P. C. A pointcut-based coverage analysis approach for aspect-oriented programs. *Information Sciences*, p. (in press), 2010.

LEMOS, O. A. L.; VINCENZI, A. M. R.; MALDONADO, J. C.; MASIERO, P. C. Unit testing of aspect-oriented programs. In: *Proceedings of the 18th Brazilian Symposium on Software Engineering (SBES)*, Brasília/DF - Brazil, p. 55–70, 2004b.

LEMOS, O. A. L.; VINCENZI, A. M. R.; MALDONADO, J. C.; MASIERO, P. C. Control and data flow structural testing criteria for aspect-oriented programs. *Journal of Systems and Software*, v. 80, n. 6, p. 862–882, 2007.

LESIECKI, N. Unit test your aspects. *IBM developerWorks Homepage*, `http://www.ibm.com/developerworks/java/library/j-aopwork11/` - last accessed on 21/09/2010, 2005.

Li, N.; Praphamontripong, U.; Offutt, J. An experimental comparison of four unit test criteria: Mutation, edge-pair, all-uses and prime path coverage. In: *Proceedings of the 4th International Workshop on Mutation Analysis (Mutation) - held in conjunction with ICST*, Denver/CO - USA: IEEE Computer Society, p. 220–229, 2009.

Linkman, S.; Vincenzi, A. M. R.; Maldonado, J. C. An evaluation of systematic functional testing using mutation testing. In: *Proceedings of the 7th International Conference on Empirical Assessment in Software Engineering (EASE)*, Keele - UK, p. 1–15, 2003.

Liu, C.-H.; Chang, C.-W. A state-based testing approach for aspect-oriented programming. *Journal of Information Science and Engineering*, v. 24, n. 1, p. 11–31, 2008.

Lopez-Herrejon, R. E.; Batory, D. *Using AspectJ to implement product-lines: A case study.* Technical report, Department of Computer Sciences, The University of Texas, Austin, Texas- USA, 2002.

Ma, Y. S.; Kwon, Y. R.; Offutt, J. Inter-class mutation operators for Java. In: *Proceedings of the 13th International Symposium on Software Reliability Engineering (ISSRE)*, Annapolis/MD - USA: IEEE Computer Society Press, p. 352–366, 2002.

Maldonado, J. C. *Critérios potenciais usos: Uma contribuição ao teste estrutural de software.* PhD Thesis, DCA/FEE, State University of Campinas (UNICAMP), Campinas, SP - Brazil, (in Portuguese), 1991.

Maldonado, J. C.; Delamaro, M. E.; Fabbri, S. C. P. F.; Simão, A. S.; Sugeta, T.; Masiero, P. C. Proteum: A family of tools to support specification and program testing based on mutation. In: *Mutation 2000 Symposium - Tool Session*, San Jose/CA - USA: Kluwer Academic Publishers, p. 113–116, 2000.

Marick, B. The weak mutation hypothesis. In: *Proceedings of the Symposium on Testing, Analysis, and Verification*, Victoria/British Columbia - Canada: ACM Press, p. 190–199, 1991.

Masiero, P. C.; Lemos, O. A. L.; Ferrari, F. C.; Maldonado, J. C. Teste de software orientado a objetos e a aspectos: Teoria e prática. In: Breitman, K.; Anido, R., eds. *Atualizações em Informática*, chapter 1, Rio de Janeiro/RJ - Brasil: Editora PUC-Rio, p. 13–71, 2006a.

Masiero, P. C.; Lemos, O. A. L.; Ferrari, F. C.; Maldonado, J. C. Teste de software orientado a objetos e a aspectos: Teoria e prática. In: Breitman, K.; Anido, R., eds. *Atualizações em Informática*, chapter 1, Rio de Janeiro/RJ - Brazil: PUC-Rio, p. 13–71, (in Portuguese), 2006b.

Massicotte, P.; Badri, L.; Badri, M. Aspects-classes integration testing strategy: An incremental approach. In: *Proceedings of the 2nd International Workshop on Rapid Integration of Software Engineering Techniques (RISE) - Revised Selected Paper*, Heraklion/Crete - Greece: Springer-Verlag, p. 158–173 (LNCS v.3943), 2006.

Massicotte, P.; Badri, L.; Badri, M. Towards a tool supporting integration testing of aspect-oriented programs. *Journal of Object Technology (JOT)*, v. 6, n. 1, p. 67–89, 2007.

Massicotte, P.; Badri, M.; Badri, L. Generating aspects-classes integration testing sequences: A collaboration diagram based strategy. In: *Proceedings of the 3rd International Conference on Software Engineering Research, Management and Applications (ACIS)*, Mt. Pleasant/MI - USA: IEEE Computer Society, p. 30–37, 2005.

Mathur, A. P. *Foundations of software testing.* Canada: Addison-Wesley Professional, 2007.

Mathur, A. P.; Wong, W. E. Evaluation of the cost of alternative mutation strategies. In: *Proceedings of the 7th Brazilian Symposium on Software Engineering (SBES)*, João Pessoa/PB - Brazil, p. 320–335, 1993.

Mathur, A. P.; Wong, W. E. An empirical comparison of data flow and mutation based test adequacy criteria. *The Journal of Software Testing, Verification, and Reliability*, v. 4, n. 1, p. 9–31, 1994.

McCabe, T. J. A complexity measure. *IEEE Transactions on Software Engineering*, v. SE-2, n. 4, p. 308–320, 1976.

McEachen, N.; Alexander, R. T. Distributing classes with woven concerns: An exploration of potential fault scenarios. In: *Proceedings of the 4th International Conference on Aspect-oriented Software Development (AOSD)*, Chicago/IL - USA: ACM Press, p. 192–200, 2005.

Mezini, M.; Ostermann, K. Conquering aspects with Caesar. In: *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD)*, Boston/MA - USA: ACM Press, p. 90–99, 2003.

MILLS, H. D. *On the statistical validation of computer programs.* Technical Report FSC-72-6015, IBM Federal Systems Division, Gaithersburg, MD - USA, 1972.

MOHAGHEGHI, P.; CONRADI, R.; KILLI, O. M.; SCHWARZ, H. An empirical study of software reuse vs. defect-density and stability. In: *Proceedings of the 26th International Conference on Software Engineering (ICSE)*, Edinburgh - UK: IEEE Computer Society, p. 282–292, 2004.

MORELL, L. J. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, v. 16, n. 8, p. 844–857, 1990.

MORTENSEN, M.; ALEXANDER, R. T. *Adequate testing of aspect-oriented programs.* Technical Report CS 01-110, Department of Computer Science, Colorado State University, Fort Collins/Colorado - USA, 2004.

MORTENSEN, M.; ALEXANDER, R. T. 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, 2005.

MORTENSEN, M.; GHOSH, S.; BIEMAN, J. M. Testing during refactoring: Adding aspects to legacy systems. In: *Proceedings of the 17th International Symposium on Software Reliability Engineering (ISSRE)*, Raleigh/NC - USA: IEEE Computer Society, p. 221–230, 2006.

MORTENSEN, M.; GHOSH, S.; BIEMAN, J. M. Aspect-oriented refactoring of legacy applications: An evaluation. *IEEE Transactions on Software Engineering*, (in press), 2010.

MUÑOZ, F.; BAUDRY, B.; DELAMARE, R.; TRAON, Y. L. Inquiring the usage of aspect-oriented programming: An empirical study. In: *Proceedings of the 25th International Conference on Software Maintenance (ICSM)*, Edmonton/AB - Canada: IEEE Computer Society, p. 137–146, 2009.

MYERS, G. J.; SANDLER, C.; BADGETT, T.; THOMAS, T. M. *The art of software testing.* 2nd ed. Hoboken/NJ - USA: John Wiley & Sons, 2004.

NAKAGAWA, E. Y.; SIMÃO, A. S.; FERRARI, F. C.; MALDONADO, J. C. Towards a reference architecture for software testing tools. In: *Proceedings of the 19th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Boston/MA - USA, p. 157–162, 2007.

Neves, V.; Lemos, O. A. L.; Masiero, P. C. 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 (LAWASP)*, Fortaleza/CE - Brazil: Brazilian Computer Society, (in Portuguese), p. 31–38, 2009.

Offutt, A. J.; Lee, A.; Rothermel, G.; Untch, R. H.; Zapf, C. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 5, n. 2, p. 99–118, 1996a.

Offutt, A. J.; Pan, J. Automatically detecting equivalent mutants and infeasible paths. *Journal of Software Testing, Verification, and Reliability*, v. 7, n. 3, p. 165–192, 1997.

Offutt, A. J.; Pan, J.; Tewary, K.; Zhang, T. An experimental evaluation of data flow and mutation testing. *Software Practice and Experience*, v. 26, n. 2, p. 165–176, 1996b.

Offutt, A. J.; Rothermel, G.; Zapf, C. An experimental evaluation of selective mutation. In: *Proceedings of the 15th International Conference on Software Engineering (ICSE)*, Baltimore/MD - USA: IEEE Computer Society, p. 100–107, 1993.

Offutt, A. J.; Voas, J.; Payne, J. *Mutation operators for Ada.* Technical Report ISSE-TR-96-06, Department of Information and Software Systems Engineering, George Mason Universit, Fairfax/VA - USA, 1996c.

Offutt, J.; Alexander, R.; Wu, Y.; Xiao, Q.; Hutchinson, C. A fault model for subtype inheritance and polymorphism. In: *Proceedings of the 12th International Symposium on Software Reliability Engineering (ISSRE)*, Hong Kong - China: IEEE Computer Society Press, p. 84–93, 2001.

Offutt, J.; Liu, S.; Abdurazik, A.; Ammann, P. Generating test data from state-based specifications. *The Journal of Software Testing, Verification and Reliability*, v. 13, n. 1, p. 25–53, 2003.

Ostrand, T. J.; Weyuker, E. J. Collecting and categorizing software error data in an industrial environment. *Journal of Systems and Software*, v. 4, n. 4, p. 289–300, 1984.

Parasoft Jtest: Java testing, static analysis, code review. Online, `http://www.parasoft.com/jtest` - last accessed on 21/09/2010, 2010.

Petersen, K.; Feldt, R.; Mujtaba, S.; Mattsson, M. Systematic mapping studies in software engineering. In: *Proceedings of the 12th International Conference on Evaluation and Assessment in Software Engineering (EASE)*, Bari - Italy: The British Computer Society, p. 1–10, 2008.

Popovici, A.; Gross, T.; Alonso, G. Dynamic weaving for aspectoriented programming. In: *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD)*, Enschede - The Netherlands: ACM Press, p. 141–147, 2002.

Pressman, R. S. *Software engineering - a practitioner's approach.* 6th. ed. New York/NY - USA: McGraw-Hill, 2005.

Rapps, S.; Weyuker, E. J. Data flow analysis techniques for program test data selection. In: *Proceedings of the 6th International Conference on Software Engineering (ICSE)*, Tokio - Japan: IEEE Computer Society, p. 272–278, 1982.

Rashid, A.; Sawyer, P.; Moreira, A.; Araújo, J. Early aspects: A model for aspect-oriented requirements engineering. In: *Proceedings of the 10th Anniversary IEEE Joint International Conference on Requirements Engineering (RE)*, Essen - Germany: IEEE Computer Society, p. 199–202, 2002.

Robillard, M. P.; Murphy, G. C. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 16, n. 1, 2007.

Runeson, P.; Skoglund, M.; Engström, E. Test benchmarks - what is the question? In: *Proceedings of the 1st Testing Benchmark Workshop (TESTBENCH) - held in conjunction with ICST*, Lillehammer - Norway, 2008.

Simão, A. S.; Souza, S. R. S.; Maldonado, J. C. A family of coverage testing criteria for coloured Petri nets. In: *Proceedings of the 17th Brazilian Symposium of Software Engineering (SBES)*, Manaus/AM - Brasil: Brazilian Computer Society, p. 209–224, 2003.

Soares, S.; Borba, P.; Laureano, E. Distribution and persistence as aspects. *Software - Practice & Experience*, v. 36, n. 7, p. 711–759, 2006.

Sommerville, I. *Software engineering.* 8th ed. Harlow - England: Addison-Wesley, 2007.

Souza, S. R. S. *Evaluation of cost and efficacy of the mutant analysis criterion in the program testing activity.* MSc Dissertation, ICMC/USP, São Carlos/SP - Brazil, 1996.

Souza, S. R. S.; Maldonado, J. C.; Fabbri, S. C. P. F.; Souza, W. L. Mutation testing applied to estelle specifications. *Software Quality Control*, v. 8, n. 4, p. 285–301, 1999.

Stamey, J.; Saunders, B.; Cameron, M. Aspect-Oriented PHP. Online, `http://www.aophp.net/` - last accessed on 17/03/2010, 2005.

Stoerzer, M.; Graf, J. Using pointcut delta analysis to support evolution of aspect-oriented software. In: *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM)*, Budapest - Hungary: IEEE Computer Society, p. 653–656, 2005.

Stratego Community AspectJ-front project home page. Online, `http://strategoxt.org/Stratego/AspectJFront` - last accessed on 09/12/2009, 2009.

Subramanyam, R.; Krishnan, M. Empirical analysis of CK metrics for object-oriented design complexity: Implications for software defects. *IEEE Transactions on Software Engineering*, v. 29, n. 4, p. 297–310, 2003.

Sugeta, T.; Maldonado, J. C.; Wong, W. E. Mutation testing applied to validate sdl specifications. In: *Proceedings of the 16th IFIP International Conference on Testing of Communicating Systems*, Oxford - UK: Springer-Verlag, p. 193–208 (LNCS v.2978), 2004.

Sullivan, K.; Griswold, W. G.; Song, Y.; Cai, Y.; Shonle, M.; Tewari, N.; Rajan, H. Information hiding interfaces for aspect-oriented design. In: *Proceedings of the 10th European Software Engineering Conference - held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Sngineering (ESEC/FSE)*, Lisbon - Portugal: ACM, p. 166–175, 2005.

The Apache Software Foundation Ant project home page. Online, `http://ant.apache.org/` - last accessed on 18/10/2010, 2009.

The AspectJ Team The AspectJ programming guide. Online, `http://www.eclipse.org/aspectj/doc/released/progguide/index.html` - last accessed on 18/10/2010, 2003.

The AspectJ Team The AspectJ 5 development kit developer's notebook. Online, `http://www.eclipse.org/aspectj/doc/released/adk15notebook/index.html` - last accessed on 21/09/2010, 2005a.

The AspectJ Team The AspectJ development environment guide. Online, `http://www.eclipse.org/aspectj/doc/next/devguide/index.html` - last accessed on 21/09/2010, 2005b.

The Eclipse Foundation AspectJ - crosscutting objects for better modularity. Online, `http://www.eclipse.org/aspectj/` - last accessed on 21/09/2010, 2010a.

The Eclipse Foundation AspectJ documentation. Online, `http://www.eclipse.org/aspectj/docs.php` - last accessed on 21/09/2010, 2010b.

The Eclipse Foundation Metrics Eclipse plugin. Online, `http://metrics.sourceforge.net/` - last accessed on 08/09/2010, 2010c.

The JBoss Team JBoss AOP Reference Documentation V2.0. Online, `http://docs.jboss.org/jbossaop/docs/index.html` - last accessed 21/09/2010, 2010.

Turner, C. D.; Robson, D. J. The state-based testing of object-oriented programs. In: *Proceedings of the Conference on Software Maintenance (ICSM)*, Montreal/Que - Canada: IEEE Computer Society, p. 302–310, 1993.

van Deursen, A.; Marin, M.; Moonen, L. *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, 2005.

Vincenzi, A. M. R. *Object-oriented: Definition, implementation and analysis of validation and testing resources.* PhD Thesis, ICMC/USP, São Carlos, SP - Brazil, (in Portuguese), 2004.

Vincenzi, A. M. R.; Delamaro, M. E.; Maldonado, J. C.; Wong, W. E. Establishing structural testing criteria for java bytecode. *Software: Practice and Experience*, v. 36, n. 14, p. 1513–1541, 2006a.

Vincenzi, A. M. R.; Maldonado, J. C.; Wong, W. E.; Delamaro, M. E. Coverage testing of java programs and components. *Science of Computer Programming*, v. 56, n. 1-2, p. 211–230, 2005.

Vincenzi, A. M. R.; Simão, A. S.; Delamaro, M. E.; Maldonado, J. C. Muta-Pro: Towards the definition of a mutation testing process. *Journal of the Brazilian Computer Society*, v. 12, n. 2, p. 49–61, 2006b.

VINCENZI, A. M. R.; WONG, W. E.; DELAMARO, M. E.; MALDONADO, J. C. Jabuti: A coverage analysis tool for java programs. In: *Proceedings of the 17th Brazilian Symposium on Software Engineering (SBES)*, Manaus/AM - Brazil: Brazilian Computer Society, p. 79–84, 2003.

VOAS, J.; MORREL, L.; MILLER, K. Predicting where faults can hide from testing. *IEEE Software*, v. 8, n. 2, p. 41–48, 1991.

WEYUKER, E. J. Using failure cost information for testing and reliability assessment. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, v. 5, n. 2, p. 87–98, 1996.

WONG, W. E. *On mutation and data flow.* PhD Thesis, Department of Computer Science, Purdue University, West Lafayette/IN - USA, 1993.

WONG, W. E.; MATHUR, A. P. Fault detection effectiveness of mutation and data flow testing. *Software Quality Journal*, v. 4, n. 1, p. 69–83, 1995.

WOODWARD, M.; HALEWOOD, K. From weak to strong, dead or alive? An analysis of some mutation testing issues. In: *Proceedings of the 2nd Workshop on Software Testing, Verification, and Analysis*, Banff/AB - Canada: IEEE Computer Society, p. 152–158, 1988.

XIE, T.; MARINOV, D.; NOTKIN, D. Rostra: A framework for detecting redundant object-oriented unit tests. In: *Proceedings of the 19th International Conference on Automated Software Engineering (ASE)*, Linz - Austria: IEEE Computer Society, p. 196–205, 2004.

XIE, T.; ZHAO, J. 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)*, Bonn - Germany: ACM Press, p. 190–201, 2006.

XIE, T.; ZHAO, J.; MARINOV, D.; NOTKIN, D. Automated test generation for AspectJ programs. In: *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs (WTAOP) - held in conjunction with AOSD*, Chicago/IL - USA, 2005.

XU, D.; DING, J. Prioritizing state-based aspect tests. In: *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST)*, Paris - France: IEEE Computer Society, p. 265–274, 2010.

Xu, D.; He, X. Generation of test requirements from aspectual use cases. In: *Proceedings of the 3rd Workshop on Testing Aspect Oriented Programs (WTAOP) - held in conjunction with AOSD*, Vancouver/BC - Canada: ACM Press, p. 17–22, 2007.

Xu, D.; Xu, W. State-based incremental testing of aspect-oriented programs. In: *Proceedings of the 5th International Conference on Aspect-Oriented Software Development (AOSD)*, Bonn - Germany: ACM Press, p. 180–189, 2006a.

Xu, D.; Xu, W.; Nygard, K. *A state-based approach to testing aspect-oriented programs.* Technical Report NDSU-CS-TR04-XU03, Department of Computer Science, North Dakota State University, Fargo/ND - USA, 2004.

Xu, D.; Xu, W.; Nygard, K. A state-based approach to testing aspect-oriented programs. In: *Proceedings of the 17th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, Taiwan, 2005.

Xu, D.; Xu, W.; WONG, W. E. Testing aspect-oriented programs with UML models. *International Journal of Software Engineering and Knowledge Engineering (IJSEKE)*, v. 18, n. 3, p. 413–437, 2008.
Available at `http://www.worldscinet.com/ijseke/18/1803/S0218194008003672.html`

Xu, G.; Rountev, A. Regression test selection for AspectJ software. In: *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, Minneapolis/MN - USA: IEEE Computer Society, p. 65–74, 2007.

Xu, W.; Xu, D. A model-based approach to test generation for aspect-oriented programs. In: *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs (WTAOP) - held in conjunction with AOSD*, Chicago/IL - USA, 2005.

Xu, W.; Xu, D. State-based testing of integration aspects. In: *Proceedings of the 2nd Workshop on Testing Aspect Oriented Programs (WTAOP) - held in conjunction with ISSTA*, Portland/Maine - USA: ACM Press, p. 7–14, 2006b.

Yamazaki, Y.; Sakurai, K.; Matsuura, S.; Masuhara, H.; Hashiura, H.; Komiya, S. A unit testing framework for aspects without weaving. In: *Proceedings of the 1st Workshop on Testing Aspect Oriented Programs (WTAOP) - held in conjunction with AOSD*, Chicago/IL - USA, 2005.

ZHANG, S.; ZHAO, J. On identifying bug patterns in aspect-oriented programs. In: *Proceedings of the 31st Annual International Computer Software and Applications Conference (COMPSAC)*, Beijing - China, p. 431–438, 2007.

ZHAO, J. 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, 2002.

ZHAO, J. Data-flow-based unit testing of aspect-oriented programs. In: *Proceedings of the 27th Annual IEEE International Computer Software and Applications Conference (COMPSAC)*, Dallas/Texas - USA: IEEE Computer Society, p. 188–197, 2003.

ZHOU, Y.; RICHARDSON, D. J.; ZIV, H. Towards a practical approach to test aspect-oriented software. In: *Proceedings of the Net.ObjectiveDays 2004 Workshop on Testing Component-based Systems (TECOS)*, Germany, p. 1–16, 2004.

ZHU, H.; HALL, P.; MAY, J. Software unit test coverage and adequacy. *ACM Computing Surveys*, v. 29, n. 4, p. 366–427, 1997.

Appendix

A

Paper: An Exploratory Study of Fault-Proneness in Evolving Aspect-Oriented Programs

This appendix presents the full contents of a paper published in the Proceedings of the 32^{nd} International Conference on Software Engineering (ICSE'10). An overview of this work was provided in Chapter 3 of this dissertation. A copyright notice in regard to it is next shown.

[1] `http://www.acm.org/publications/policies/copyright_policy#Retained` - last accessed on 06/07/2010.

An Exploratory Study of Fault-Proneness in Evolving Aspect-Oriented Programs

Fabiano Ferrari[1], Rachel Burrows[2,3], Otávio Lemos[4], Alessandro Garcia[2], Eduardo Figueiredo[3], Nelio Cacho[5], Frederico Lopes[6], Nathalia Temudo[7], Liana Silva[7], Sergio Soares[8], Awais Rashid[3], Paulo Masiero[1], Thais Batista[6], José Maldonado[1]

[1] Computer Systems Department, University of São Paulo – USP, São Carlos, Brazil
[2] Informatics Department, Pontifical Catholic University of Rio de Janeiro – PUC-Rio, Rio de Janeiro, Brazil
[3] Computing Department, Lancaster University, Lancaster, United Kingdom
[4] Department of Science and Technology, Federal University of São Paulo – UNIFESP, S.J. Campos, Brazil
[5] School of Science and Technology, Federal University of Rio Grande do Norte – UFRN, Natal, Brazil
[6] Computer Science Department, Federal University of Rio Grande do Norte – UFRN, Natal, Brazil
[7] Department of Computing and Systems, University of Pernambuco – UPE, Recife, Brazil
[8] Informatics Centre, Federal University of Pernambuco – UFPE, Recife, Brazil

{ferrari,masiero,jcmaldon}@icmc.usp.br, {rachel.burrows,e.figueiredo,marash}@comp.lancs.ac.uk, otavio.lemos@unifesp.br afgarcia@inf.puc-rio.br, neliocacho@ect.ufrn.br, {nmt,lsos}@dsc.upe.br, scbs@cin.ufpe.br, {fred.lopes,thais}@ufrnet.br

ABSTRACT

This paper presents the results of an exploratory study on the fault-proneness of aspect-oriented programs. We analysed the faults collected from three evolving aspect-oriented systems, all from different application domains. The analysis develops from two different angles. Firstly, we measured the impact of the obliviousness property on the fault-proneness of the evaluated systems. The results show that 40% of reported faults were due to the lack of awareness among base code and aspects. The second analysis regarded the fault-proneness of the main aspect-oriented programming (AOP) mechanisms, namely pointcuts, advices and intertype declarations. The results indicate that these mechanisms present similar fault-proneness when we consider both the overall system and concern-specific implementations. Our findings are reinforced by means of statistical tests. In general, this result contradicts the common intuition stating that the use of pointcut languages is the main source of faults in AOP.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging – Diagnostics, Code Inspections; D.3.3 [**Programming Languages**]: Language Constructs and Features.

General Terms

Measurement, Experimentation, Languages, Verification.

Keywords

Aspect-oriented programming, fault-proneness, software testing.

1. INTRODUCTION

With software development becoming increasingly incremental, programmers should be aware of contemporary implementation mechanisms that are fault-prone in the presence of changes. In particular, the recent adoption of aspect-oriented programming (AOP) languages and frameworks, such as AspectJ [32] and Spring [23], requires a better understanding of the fault causes in AOP. AOP [25] is a programming technique that aims to improve the modularisation and robust implementation of concerns that cut across multiple software modules. Classical examples of crosscutting concerns usually implemented as aspects are logging, exception handling, concurrency, and certain design patterns.

The modularisation of crosscutting concerns in AOP is achieved through a complementary set of programming mechanisms, such as pointcut, advice, and intertype declaration (ITD) [32]. In addition, a basic property associated with AOP is *obliviousness*. This property implies that the developers of core functionality need not be aware of, anticipate or design code to be advised by aspects [15]. Obliviousness is also influenced by quantification, i.e. the ability to declaratively select sets of points via pointcuts in the execution of a program. The more expressive a pointcut language is, the more support for obliviousness it provides [31].

New AOP models, frameworks and language extensions are constantly emerging, in some cases leveraging different degrees of obliviousness [17, 20, 26, 35]. While some researchers have been optimistic about the benefits of AOP [26], others have shown scepticism [1, 2, 28]. For example, previous research has indicated that the use of certain AOP mechanisms can violate module encapsulation [1] and even introduce new types of faults [2]. In particular, some researchers claim these faults are likely to be amplified in the presence of evolutionary changes [24]. However, the empirical knowledge about the actual impact of AOP on fault-proneness remains very limited even for core programming mechanisms, such as pointcuts and intertype declarations.

Concerned with these issues, we present a first exploratory analysis on the fault-proneness of AOP properties and mechanisms. Our analysis develops in terms of the three following questions: (1) How does obliviousness influence the presence of faults in aspect-oriented (AO) programs? (2) What is the impact of specific AOP mechanisms on fault-proneness? and (3) Whether and how do certain characteristics of concern implementations correlate with the introduction of faults? To the best of our knowledge, this is the first initiative that systematically

ICSE '10, May 2-8, 2010, Cape Town, South Africa

tackles these questions based on the exploratory analysis of real-world AO systems. Previous research has mainly focused on defining fault taxonomies and testing approaches based on AOP main concepts and current technologies [2, 3, 4, 10, 12, 34, 36]. However, we can only find limited empirical evidence on how such faults occur in practice.

To achieve our goals, we selected three AO systems from different application domains. We analysed fault-prone AOP mechanisms by means of various testing and static analysis. Throughout the study, faults were reported by two means: (1) by developers during system development and evolution; then afterwards (2) by independent testers who reported post-releases faults. The main findings of our study are:

- Obliviousness has been a controversial software property since the early research in AOP [28, 31]. Our analysis confirms that the lack of awareness between base and aspectual modules tends to lead to incorrect implementations. Despite the modern support provided by AOP-specific IDEs, uncertainty about module interactions still remains in the presence of aspects within the system.
- Interestingly, our findings contradict the common intuition stating that the use of pointcut languages is the main source of faults in AOP [10, 12, 29]. The main AOP mechanisms currently available in AspectJ-like languages – namely, pointcuts, advices and intertype declarations (ITDs) – present similar fault-proneness when the overall system is considered.
- The numbers of internal AOP mechanisms showed to be good fault indicators when considering the sets of modules within each concern[1] implementation separately. In this case the number of faults associated with a concern was directly proportional to the number of AOP mechanisms used to implement that concern.

In addition, the gathered results in our study support these findings with statistical significance. The remainder of this paper is organised as follows: Section 2 summarises the related research. Section 3 describes the study configuration. It includes our research hypotheses and a description of the target systems and evaluation procedures. Following, Section 4 brings the collected data. This data is analysed and discussed in Section 5. Section 6 discusses the limitations of this work. Finally, Section 7 presents our conclusion and future research.

2. RELATED WORK

The difficulty of performing fault-proneness evaluation in AOP is mainly twofold: (i) there are not yet several releases and documented faults for AO software projects available for analyses, and (ii) AOP has introduced new properties, such as obliviousness and quantification, and a wide range of mechanisms often used to implement different categories of crosscutting concerns. Previous research was limited to evaluate the benefits and drawbacks of aspects from different angles, such as design stability [13, 16] and system robustness [5, 9, 14]. However, the fault-proneness of AO programs is not yet a well-understood phenomenon.

To date, there is limited empirical knowledge of the impact of AOP on software correctness. In spite of that, we present pieces of work that we believe are mostly related to our own. They are basically distributed in two categories discussed in the following:

Fault taxonomies and fault-proneness of AO programs: A number of fault taxonomies and bug patterns for AO programs can be found in the recent literature [2, 4, 9, 12, 36]. Alexander et al. [2] proposed the first study on AOP-specific faults. Their main contribution is a characterisation of possible sources of faults that are specific to AO programs. Based on these sources, Alexander et al. also defined a high-level fault taxonomy for AO software, mainly focusing on features of AspectJ-like languages. Further fault taxonomies were built on top of it [4, 36]. They either include new fault types or refine the already existing ones. However, none of them have been empirically evaluated to date. Ferrari et al. [12] summarised these and other taxonomies in a broader fault categorisation for AO software. The study presented in this paper uses such categorisation for classifying the AOP-related faults found in the target systems.

So far, we have identified a single attempt to evaluate the fault-proneness of AOP programs. Coelho et al. [9] present an exploratory study of the robustness of AspectJ-based systems. The study analyses the flow of exceptions in five medium-sized OO systems from different domains and their AO counterparts, whose exception-handling code have been aspectised. The results show that in all AO systems there was an increase in the number of uncaught exceptions (i.e. exceptions that cross the system boundaries without being handled) and also in the number of unexpected handler actions. Differently from our work, Coelho et al. only focus their analysis on exception handling mechanisms, while we focus on AOP mechanisms in general.

Fault-proneness of crosscutting concerns: Eaddy et al. [11] performed an empirical study which provides evidence suggesting that crosscutting concerns cause defects. The authors examined the concerns of three medium-sized open-source Java projects and found that the more scattered the implementation of a concern is, the more likely it is to have defects. Their findings recommend the use of AOP techniques to modularise crosscutting concerns in order to reduce the number of faults potentially caused by them. However, Eaddy et al. have not investigated the fault-proneness of AOP mechanisms as we do in this study. In our study, we also analyse crosscutting concerns, although from a different angle. We aim to evaluate the impact of the specific characteristics of AO concern implementations on the system fault-proneness. We also highlight that we do not contrast AO implementations with OO counterparts in order to find out which approach results in more or less faults related to crosscutting concerns.

3. STUDY SETTING

This section describes our study configuration. Section 3.1 presents our goals and hypotheses. Section 3.2 provides an overview of the target systems. Section 3.3 explains the evaluation procedures we applied to each selected system and the associated tooling support.

3.1 Goal Statement and Research Hypotheses

Our objective is to evaluate the fault-proneness of AOP properties and mechanisms when they are applied to evolving programs. We are particularly interested in observing the underlying reasons of the introduction of faults. First, we analyse whether a key property of AOP, obliviousness, facilitates the emergence of faults under software evolution conditions. Moreover, we aim at analysing how specific characteristics of concerns being *aspectised* impact on the fault-proneness of AO programs. For example, we

[1] From hereafter, we use the term "concerns" to refer to crosscutting concerns in general, as defined by Kiczales et al. [25].

investigate how the internal implementation details of a concern, such as lines of code and use of specific AOP mechanisms, are correlated with the presence of faults. Based on these goals, we defined two research hypotheses. For each of them, the null and the alternative hypotheses are as follows:

Hypothesis 1 (H1)

- H1-0: Obliviousness does not exert impact on the fault-proneness of evolving AO programs;
- H1-1: Obliviousness exerts impact on the fault-proneness of evolving AO programs;

Hypothesis 2 (H2)

- H2-0: There is no difference among the fault-proneness of the main AOP mechanisms;
- H2-1: There are differences among the fault-proneness of the main AOP mechanisms;

To achieve our goals, we needed to apply a number of evaluation procedures. Our analysis embraced 12 releases of three AO systems from different application domains. Such systems include a wide range of heterogeneous concerns implemented as aspects. The systems and evaluation procedures are following described.

3.2 The Target Systems

The three medium-sized applications used in this study are from significantly different application domains. The first one, called iBATIS [22], is a Java-based open source framework for object-relational data mapping. It was originally developed in 2002 and over 60 releases are available at SourceForge.net[2] and Apache.org[3] repositories. The second application is HealthWatcher (HW) [16, 27], a typical Java web-based information system. HW was first released in 2001 and allows citizens to register complaints about health issues. The third evaluated system is a software product line for mobile devices, called MobileMedia (MM) [13]. MM was originally developed in 2005 to allow users to manipulate image files in different mobile devices. It has then evolved to support the manipulation of additional media files, such as videos and MP3 files.

Every AO release of a given system has an OO counterpart. In particular, iBATIS had its AO releases derived from OO builds available at SourceForge.net, which were used as baselines for implementation alignment. HW and MM, on the other hand, have evolved based on a sequence of planned changes, and both OO and AO versions of given release were developed concurrently [13, 16]. All releases have experienced exhaustive assessment procedures – code revision and testing (Section 3.3) – by independent developers in order to achieve functionalities well aligned with the original Java system.

From hereafter, we refer to the AO versions[4] of the target systems by their simple names or abbreviations, i.e. iBATIS, HW and MM. Four releases of iBATIS were considered in our evaluation, namely iBATIS[5] 01, 01.3, 01.5 and 02. We also analysed four HW releases – HW 01, 04, 07 and 10 – and four MM releases – MM 01, 02, 03 and 06. These releases were chosen because they encompass a wide range of different fine-grained and coarse-grained changes, such as refactorings and functionality increments or removals. Table 1 shows some general characteristics of the three target systems. For more information about each of them, the reader may refer to the respective placeholder websites or to previous reports of these systems [13, 16, 22].

Table 1. Key characteristics of the target applications.

	iBATIS	Health Watcher	MobileMedia
Application Type	data mapper framework	health vigillance application	product line for mobile data
Code Availability	Java/AspectJ	Java/AspectJ	Java/AspectJ
# of Releases	60 / 4	10 / 10	10 / 10
Selected Releases	4	4	4
Avg. KLOC	11	6	3
Avg. # of Aspects (only AspectJ)	46	23	10
Evaluation Procedure	testing	testing	interference analysis

We selected iBATIS as the main subject of illustrative examples in this paper in order to promote coherence in the discussions. This is the most complex target system from which we derived the largest data set (e.g. number of faults) and on which we mostly draw our analyses. However, we also refer to examples of the other systems in order to highlight recurring observations across the three systems. The highest number of faults in iBATIS was expected. The already-stable implementation of the other two systems yielded less fault-related data than iBATIS. Their implementations are more mature as they have been originally released for four years or more and underwent more corrective and perfective changes. HW and MM have also been targeted by a number of previous studies focusing in other equally-important quality attributes [9, 13, 14, 16, 27]. Therefore, as the AO implementations have different degrees of stability, the results originated from these systems will provide support for drawing more general findings.

3.3 The Evaluation Procedures

We followed different approaches to evaluate each target system. The evaluation procedures were defined according to the system characteristics and information available at the moment this study started. In short, we aimed at identifying as many faults as possible given time and resource constraints, while systematically avoiding bias while evaluating the three systems.

3.3.1 iBATIS Evaluation

Evaluation strategy: The iBATIS system has experienced two testing phases: pre-release and post-release testing. Pre-release testing aimed at producing defect-free code to be committed to a CVS repository. The test sets provided with the original OO implementations were used as baselines in this phase. Any abnormal behaviour when regressively testing the AO version of a given release was investigated. When a fault was uncovered, it should be documented in an appropriate detailed report (Section 3.3.4). Post-release testing, on the other hand, aimed at assessing the implementation through enhancement of the original test sets. The enhanced tests were executed against both OO and AO versions of a given release. A fault should *only* be reported if it was noticed in the AO version but *not* in the OO counterpart. This procedure ensured that only faults introduced during the aspectisation process would be reported for further analysis.

Tooling support: For test case design and execution, we used JUnit[6] and GroboUtils[7], a JUnit extension that enables multi-

[2] http:// sourceforge.net/projects/ibatisdb/files/ (03/02/2010)

[3] http://archive.apache.org/dist/ibatis/binaries/ibatis.java/ (03/02/2010)

[4] References to OO counterparts will be made explicit throughout the text.

[5] Such releases correspond to the original builds #150, #174, #203 and #243 found in SourceForge.net, respectively.

[6] http://www.junit.org/ (03/02/2010)

[7] http://groboutils.sourceforge.net/ (03/02/2010)

threaded tests. To measure test coverage, we used Cobertura[8], a tool that allows for fast code instrumentation and test execution.

3.3.2 HW Evaluation

Evaluation strategy: HW was tested in a single phase in our study as initial testing was already performed during its development time. Such initial tests involved people who were unaware of evaluations that would be further performed. Hence, the testing of the HW system was extended in this study to cover the assessment phase, thereby improving the degree of test coverage. Differently from iBATIS, however, no original test set was made available. Thus a full test set was built from scratch based on the system specification and code documentation. In order to reduce test effort and avoid systematic bias during test creation, test cases were automatically generated with adequate tool support. As well as for iBATIS, a fault should *only* be reported if it was noticed in the AO version but *not* in the OO counterpart, and all uncovered faults were similarly reported.

Tooling support: We used CodePro[9], an Eclipse plugin for automatic JUnit test case generation for Java programs. We also used Mockrunner[10], a lightweight framework for testing web-based applications. It extends JUnit and provides the necessary facilities to test the servlets implemented within the HW system. Finally, we used Cobertura to measure the test coverage.

3.3.3 MM Evaluation

Evaluation strategy: As well as HW, MM has not been developed with awareness of further fault-based evaluation. Moreover, post-release tests during system evolution and maintenance only revealed faults related to robustness (e.g. data input validation), however not necessarily being related to AOP mechanisms. Despite this, we have evaluated MM using the Composition Integrity Framework (CIF) [6]. CIF helped us identify problems in aspect interactions established either between aspects and base code or among multiple aspects. Since MM is a software product line and includes mandatory, optional and alternative features, CIF was applied in varied configurations of MM. In doing so, we were able to derive a set of faults that resulted from a broad range of aspect-oriented compositions.

Tooling support: We used the CIF framework, which performs static analysis of join point shadows. CIF is able to: (i) report the join point shadows that are involved in a specific composition, and (ii) report aspect interactions which are not governed by an explicit dependency, e.g. via the use of the declare precedence statement.

3.3.4 Fault Reporting

Every fault identified either during development (iBATIS only), during the assessment phase (iBATIS and HW), or during static analysis (MM) was documented in a customised report form. During the assessment or static analysis phases, the testers provided information as much as possible, with special attention to the test case(s) that revealed the fault (if applicable) and the fault symptom. In addition, the tester provided some hints about the fault location. Then, the reports were forwarded to the original developers, who were in charge of concluding the fault documentation.

3.3.5 Fault Classification

After the fault documentation step, each fault was classified according to a fault taxonomy for AO software proposed in our previous research [12]. This taxonomy includes four high-level categories of faults that comprise the core mechanisms of AOP: (1) pointcuts; (2) introductions (or intertype declarations – ITDs) and other `declare`-like structures; (3) advices; and (4) the base program. In order to systematically classify every fault, it was took into account the fault origin and not only its side-effects. The classification based on the first three categories above was straightforward. For instance, it was relatively trivial to identify mismatching pointcuts, misuse of `declare`-like structures or wrong advice types. However, base program-related faults required extended analysis and reasoning about them. For example, base code changes that result in broken pointcuts should be classified as base program-related although their side-effects might be unmatched join points in the code. Section 5 analyses the impact of each fault category on the overall fault distribution considering all targeted systems.

4. DATA COLLECTION

This section presents the results obtained for each target system. Section 4.1 describes the results of test execution in iBATIS and HW releases, and the number of aspect interaction problems identified in the MM configurations. Section 4.2 presents the fault distribution per fault category and the fault distribution per concern.

4.1 Test Execution Results

Figure 1 shows the test execution results for iBATIS (on the left-hand side) and HW (on the right-hand side). The iBATIS original test sets comprise 100, 103, 108 and 130 test cases for each release, respectively. The final, improved test sets include 246 test cases for iBATIS 01, 01.3 and 01.5, and 256 tests for release 02. For a given release, new test cases were either mined from the successive releases in the SourceForge repository or designed from scratch. According to Figure 1, the number of successful test cases increased across the releases, what might mean code enhancement. However, as discussed in the next sections, this not necessarily means reduction in number of faults.

HW test sets were entirely designed for this study, with support from CodePro. Additionally, a few tests were manually implemented based on functional requirements. In total, 925, 981, 998 and 998 tests were generated for releases 01, 04, 07 and 10, respectively. We can observe in Figure 1 that the number of tests that failed plus the tests that raised exceptions (labelled as *error* in the legend) increased across the releases. This suggests that the number of faults increased during the system evolution. However, as we will see in Section 4.2, this not necessarily means larger number of faults in successive releases.

Figure 2 presents the coverage achieved for each release of iBATIS and HW. For iBATIS in particular, the figure shows the coverage obtained with the original and enhanced test sets. In spite of HW test sets being significantly larger than the iBATIS ones, the yielded coverage in HW is lower than in iBATIS. This is due to automatically-generated redundant tests that exercise common parts of the code. Nevertheless, for both systems, we focused on the creation of tests that exercise parts of the code affected by the aspectual behaviour. Following this strategy, we were able to uncover faults not yet revealed in previous system evaluations.

[8] http://cobertura.sourceforge.net/ (03/02/2010)

[9] http://www.instantiations.com/ (03/02/2010)

[10] http://mockrunner.sourceforge.net/ (03/02/2010)

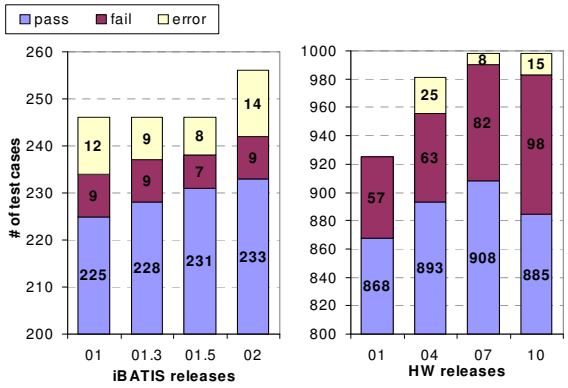


Figure 1. Test case execution in iBATIS (left) and HW (right) releases.

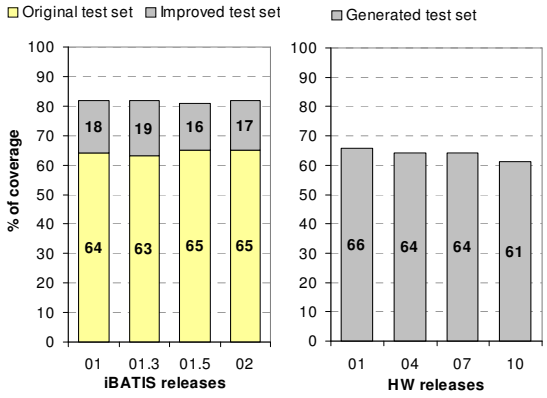


Figure 2. Test coverage in iBATIS (left) and HW (right) releases.

As described in Section 3.3, the MM system was evaluated through static analysis using the CIF framework [6]. This activity yielded a list of potential faults related to aspect interactions regarding advices that share common join points. In MM, such list included 16 potential faults for the varied configurations of the four evaluated releases. Further analysis enabled us to identify and classify the real faults. The results are presented in the next section.

4.2 Fault Distribution

This section summarises all faults we identified along our study. Faults are grouped per category (Section 4.2.1) and per concern (Section 4.2.2). The fault categorisation is in accordance to the fault taxonomy for AO programs [12].

4.2.1 Fault Distribution per Fault Category

Tables 2 and 3 respectively present the number of reported faults for iBATIS and for all the systems. The reported faults are grouped per category. The lower number of faults uncovered for HW and MM can be explained by their size – they are smaller than iBATIS – and by these systems having experienced only post-release evaluation. A total of 83 faults in iBATIS (20.8 faults on average per release) and 104 considering all systems have been documented. In-depth analyses of the reported faults drive the discussion presented in Section 5.

.

Table 2. Fault distribution per category in iBATIS.

Fault Category	iBATIS releases				Total	Average
	01	01.3	01.5	02		
Pointcut-related	10	2	1	5	**18**	4.5
ITD-related	5	2	1	6	**14**	3.5
Advice-related	6	4	1	4	**15**	3.8
Base-program related	2	1	10	23	**36**	9.0
Total	**23**	**9**	**13**	**38**	**83**	**20.8**

Table 3. Fault distribution per category in all systems.

Fault Category	iBATIS	MM	HW	Total
Pointcut-related	18	1	0	19
ITD-related	14	4	6	24
Advice-related	15	4	4	23
Base-program related	36	0	2	38
Total	**83**	**9**	**12**	104

4.2.2 Fault Distribution per Concern in iBATIS

Table 5 presents the fault distribution per concern in iBATIS. To analyse this distribution, we considered only the iBATIS data set because it contains the largest amount of faults. Moreover, iBATIS is the only system where faults appear distributed over all the aspectised concerns. Data sets collected for the other two applications, on the other hand, were limited to post-release evaluations only and are not considered in this section. The concerns listed in Table 5 are briefly described in Table 4. Notice that some concerns are aspectised only in releases 01.5 and 02. Section 5.2 discusses how certain implementation characteristics of a concern (e.g. required AOP mechanisms) may impact on the fault-proneness of that specific concern.

Table 4. Concerns implemented with aspects in iBATIS.

Concern	Description	Release
Concurrency	Ensures multiple activities and requests could be executed within the framework in a consistent manner.	all
Type Mapping	Deals with the mapping of data into different formats. E.g. when data is retrieved and stored in the database, the application checks to see if the data content is not null before proceeding with the transaction.	all
Design Patterns	Subset of the Gang-of-Four design patterns such as Singleton, Observer, Adapter, and Strategy	all
Error Context	ErrorContext object stores data about executing activities. This data is used and sometimes printed as an event trace in the event of an exception.	all
Exception Handling	Mechanisms that deal with to an erroneous execution flows (In Java this includes try/catch/throws and finally clauses).	all
Connection, Session & Transaction	Detected as three separate concerns, regards mechanisms that allow for database access and control. E.g. transaction managers and SQL query runners.	01.5 and 02

Table 5. Fault distribution per concern in iBATIS.

Concern	iBATIS releases				Total	Average
	01	01.3	01.5	02		
CC - Concurrency	0	0	0	1	**1**	0.25
TM - Type Mapping	2	0	0	1	**3**	0.75
DP - Design Patterns	2	0	0	0	**2**	0.5
EC - Error Context	13	5	1	2	**21**	5.25
EH - Exception Handling	6	4	2	16	**28**	7
CN - Connection	--	--	7	10	**17**	8.5
SS - Session	--	--	3	3	**6**	3
TR - Transaction	--	--	0	3	**3**	1.5
OT - Others	0	0	0	2	**2**	0.5
Total	**23**	**9**	**13**	**38**	**83**	**20.8**

5. DATA ANALYSIS AND DISCUSSION

This section performs exploratory and statistical analyses of the measures presented in Section 4. We aim at identifying AOP properties and mechanisms that tend to yield faulty implementations. Section 5.1 evaluates the H1 hypothesis, i.e. how the obliviousness property impacts the correctness of AO programs in the presence of code evolution. Section 5.2 evaluates the H2 hypothesis in order to identify the fault-proneness of specific AOP mechanisms. We evaluate H2 from two points of view: considering the overall system implementation, and the implementations of specific concerns. Both analyses for H2 are supported by statistical tests.

For the statistical tests performed along section 5.2, we used the R language and environment[11]. We use the *Pearson's chi-square* test to check whether or not there is a statistically significant difference between the fault counts. We assume the commonly used confidence level of 95% (that is, p-value threshold = 0.05). The *Spearman's rank correlation* test is used to check how the fault counts correlate with AOP mechanism counts. This test is used because in our analysis the used metrics (i.e. number of faults) are nonparametric. For evaluating the results of the correlation tests, we adopted the Hopkins criteria to judge the goodness of a correlation coefficient [21]: < 0.1 means trivial, 0.1-0.3 means minor, 0.3-0.5 means moderate, 0.5-0.7 means large, 0.7-0.9 means very large, and 0.9-1 means almost perfect.

5.1 H1: The Impact of Obliviousness

Obliviousness plays a central role in AOP, but there is little empirical knowledge on how this property actually affects the fault-proneness under usual development settings. We then analysed its impact on the fault-proneness of AO systems from two viewpoints: (i) obliviousness and software evolution; and (ii) a categorisation of obliviousness listed by Sullivan et al. [31]. The results are following presented.

5.1.1 Obliviousness and Software Evolution

Considering the fault distribution per fault category (Tables 2 and 3) for iBATIS, the total number of faults related to the base code was 36, what is at least twice as large as any other number of faults within the other three categories. From these, 27 faults were caused by either perfective or evolutionary changes within the base code, what led pointcuts to break. They represent the largest number amongst all fault types reported for the iBATIS system, which in turn corresponds to 33% of the total number of faults.

This problem, usually referred to as the *fragile pointcuts* problem [29], is closely related to the quantification and obliviousness models implemented in AspectJ-like languages. Changing a program requires a review of the crosscut enumerations (i.e. the pointcuts) which conflicts with the idea of programs being oblivious to the aspects applied to them [18]. We found that this problem is magnified in realistic development scenarios as the one observed in iBATIS. Several developers worked in parallel in the iBATIS project, each of them refactoring and evolving different crosscutting concerns into aspects. This means that obliviousness was present not only between base code and aspects. The aspect implementations were oblivious to each other as well. For example, an aspect that advises a set of join points might not be aware of other aspects inserting behaviour into the same join points, hence rising the risk of either misbehaviour or pointcut mismatching in the event of any code change. Partial aid currently provided by AOP IDEs increases the developers' awareness of the aspect effects in the base code. However, uncertainty about how aspects indirectly interfere in the base code still remains.

We noticed this problem occurred mainly in the iBATIS system as faults were reported during both development and assessment phases (Section 3.3.4). Evolving some functionality necessarily required fixing faults identified in the existing base or aspect code. On the other hand, HW and MM implementations were more stable and were extensively evaluated in previous research [9, 13, 14, 16, 27]. For example, since HW had been first released, a number of incremental and perfective changes took place [16], what resulted in both base and crosscutting code being more stable than in iBATIS code. Due to these refinements, HW and MM had proportionally fewer faults in this category, most likely due to the robustness of the code. Nonetheless, none of the three systems have been evaluated in terms of fault-proneness, as presented in this paper.

We further analysed the MM system this using the CIF framework [6], and the results reinforce our findings. The majority of faults here were sourced from areas of code where aspect interactions occurred at runtime. For example, in 45% of cases (4 out of 9), faults were caused due to missing `declare precedence` statements. These faults resulted in arbitrary execution order of advices that share the same join point. Obliviousness was clearly the main reason for the introduction of faults in these cases, where aspects were successively introduced along the development cycles. Considering the same scenario in Java, the developers were naturally enforced to make an explicit design decision on the order of respective pieces of behaviour within a method.

5.1.2 Obliviousness Categories

We classified all documented faults according to the four categories of obliviousness listed by Sullivan et al. [31]. The summary of this categorisation is presented in Table 6. The goal was to gather further evidence about the impact of obliviousness on the correctness of the evaluated systems. The obliviousness categories represent different types of information hiding. We focused on two types of obliviousness that are relevant for the purposes of this analysis, briefly described as follows: (i) *Language-level obliviousness* is present when there is no local notation in the code about aspect behaviour that may be inserted at this point; and (ii) *Feature obliviousness*, which is present when the base code developer is unaware of the features or the in-depth semantics of an aspect that is advising the base code.

Even though it is impossible to be entirely sure of the true causes of faults, we followed a set of guidelines to decide if the collected faults were likely to have been caused by language-level and/or feature obliviousness. In short, when behaviour is inserted at join points via advice, we can claim that language-level obliviousness

[11] http://www.r-project.org/ (03/02/2010)

is present. This is because there is no explicit call or notation of this extra behaviour within the base code. We categorised a fault as caused by language-level obliviousness if the fault was likely to be avoided in case such an explicit local notation was present. Now, let us consider a base code developer who has followed specific design rules to expose certain join points or create hooks for an aspect developer without full knowledge of the implementation details of this aspect. In this case, language-level obliviousness is not present, but feature obliviousness is. We classify a fault as being related to feature obliviousness if, in order for the fault to have been avoided, further attention would need to be given to the aspect semantics.

Table 6. Faults associated with obliviousness.

System	Obliviousness Category					
	Language	Feature	Both	Language only	Feature only	Total
iBATIS	31	4	4	27	0	31
HW	0	3	0	0	3	3
MM	8	4	4	4	0	8
Total			8	31	3	42

The results of this categorisation show that most of the faults related to obliviousness were categorised as language-level. They represent around 70% of all base program-related faults (i.e. 26 out of 38 – see Total column in Table 3). In regard to faults related to feature obliviousness, they were mostly found in cases where aspects either directly interact within the same module or share common join points. This indicates that evolving code that is oblivious to aspects has varying impacts on the fault-proneness of the system. This problem was mainly observed in iBATIS, in which faults have been reported during the evolution of the releases. MM and HW, on the other hand, experienced only post-release tests. Nevertheless, 11 out of 21 faults revealed for HW and MM were assigned to obliviousness at either language-level, feature or both (see Total column in Table 6).

To conclude, analysing the impact of obliviousness on the fault-proneness of the evaluated systems provided us with evidences that support the H1 alternative hypothesis (H1-1). That is, "*Obliviousness exerts impact on the fault-proneness of evolving AO programs*". In our study, a large amount of faults (40%) could be directly associated with the base code being oblivious to aspects. Their majority was observed in the iBATIS evolution. Many faults observed were also related to aspects being oblivious to other aspects. Missing `declare precedence` statements were responsible for 45% of the faults found in MM. Considering all faults (Table 3), 11% were categorised as feature obliviousness-related, although a much larger proportion were related to language-level (i.e. 38%). This result is interesting because it might further reinforce the motivation for AOP models based on explicit class-aspect interfaces, such as XPIs [17] and EJPs [20].

5.2 H2: Fault-proneness of AOP Mechanisms

There is often an assumption that the use of pointcut languages is the main source of faults in aspect-oriented programs [10, 12, 29]. However, there is limited understanding of the real magnitude of pointcut faults with respect to other AOP mechanisms. The analysis of our second hypothesis is drawn in terms of the fault categorisation presented in Section 4. The null hypothesis (H2-0) states that there is no significant difference amongst the fault-proneness of core AOP mechanisms.

5.2.1 Analysing the overall fault distribution

Initially, we analyse the values presented in Table 2. Examination of this data indicates that there is a similarity among the total number of faults found in iBATIS, considering the first three categories (pointcut-, advice- and ITD-related). These results suggest these mechanisms present similar fault-proneness; that is, none of them stands out with respect to the number of faults. To further analyse this observation statistically, we first applied a Pearson's chi-square test. This test checks the probability of sample data coming from a population with a specific distribution. If we reach a probability (p-value) higher than, say, 0.05, we can assert with 95% confidence level that there is no reason to reject the hypothesis that the observed data fits the given distribution. In our case, at a confidence level of 95%, the result confirms the uniformity of the fault frequencies among each fault category: the p-value is evaluated to 0.7584, significantly higher than 0.05. This is easy to see since fault counts were 18, 14, and 15; very close to the fitted model where each category is expected to have approximately the same number of faults (15.67 in this case). We then applied the chi-square test to the overall fault set, considering all target systems (i.e. the total of 104 faults presented in Table 3). Again, assuming the confidence level of 95%, the result corroborates the previous finding, i.e. the faults are uniformly distributed over the three main AOP mechanisms, with p-value being evaluated to 0.7275, again significantly higher than 0.05.

Contradicting the conventional wisdom, we have first evidence that supports the H2 null hypothesis (H2-0) that "*there is no difference among the fault-proneness of the main AOP mechanisms*". This is also an interesting result as many researchers have strictly focused on improving the design of pointcut languages (e.g. providing support for more expressive pointcuts [5, 18, 20]). Less attention has been given to support more robust programming with other classical AOP mechanisms. The next section presents a more refined analysis that brings additional evidence on the fault-proneness of such mechanisms from a different point of view.

5.2.2 Analysing the fault distribution per concern

The analysis of fault distribution per fault category only took into account the overall number of faults per category. This section provides a more refined analysis about fault-proneness of AOP mechanisms. For this, we considered certain internal details in the implementation of each concern. We performed a correlation analysis to gather empirical evidence of a cause-effect relationship between the number of AOP mechanisms and defects. This is motivated by the fact that AOP mechanisms may have individual impact on the fault-proneness of a module, a cluster of modules (e.g. modules that implement a given concern) or the full system. For this analysis, we considered only the set of faults identified in iBATIS, since it includes representatives distributed over all the aspectised concerns (Table 4). Moreover, we also considered base program-related faults in order to measure the impact of AOP mechanisms in the system as a whole.

Initially, we applied the Spearman's rank correlation to check how the overall number of AOP mechanisms (pointcuts, advices and ITDs) per release in iBATIS (Table 7) correlates with the number of faults in the same release. The results are presented in Table 8. The correlation is generally low, considering all AOP mechanisms, thus contradicting the results that regard fault distribution per fault category (Section 5.2.1).

Table 7. Number of AOP mechanisms and faults in iBATIS.

iBATIS release	Pointcuts	Advices	ITDs	Faults
01	97	94	79	23
01.3	121	118	86	9
01.5	244	240	238	13
02	244	238	237	38

However, while performing such an analysis based in internal properties of the systems, we should take into account concern-specific characteristics. This is due to the nature of concern implementations, which usually require subsets of AOP mechanisms to be used together. For example, aspectising an exception handler usually requires three coding structures in AspectJ: a pointcut expression, a `declare soft` statement and an advice. In other words, we can investigate whether the number of pointcuts, advices and ITDs (including `declare`-like mechanisms) implemented for the purposes of a concern impact on the number of faults it presents.

Table 8. Correlation between number of faults and number of AOP mechanisms in iBATIS releases.

Metric	Coefficient	P-value
POINTCUTS	0.2108185	0.7892000
ADVICES	0.0000000	1.0000000
DECLARATIONS	0.0000000	1.0000000

Hence, we checked how the number of AOP mechanisms used to implement a concern correlates with the fault distribution per concern. We measured the maximum and the average number of each AOP mechanism per concern across all iBATIS releases. Note that, for a given concern, we considered all modules (aspects and classes) that were involved in its implementation. We applied again the Spearman's rank correlation to the total and average number of faults per concern across the releases. We compared these numbers against the maximum and average number of advices, pointcuts, and ITDs per concern across releases. With such an analysis we can observe whether and how the usage frequency of each AOP mechanism seems to impact on the fault distribution per concern. We used the maximum and average number of mechanisms across releases because they might repeat from release to release. That is, the same pointcut implemented in an exception handling aspect in one release may be present in the same aspect in another release

We also chose two metrics typically applied to OO and AO programs in order to compare the results obtained in this analysis. These metrics are lines of code (LOC) and weighted operations per module (WOM) [7]. WOM adapts the original weighted methods per class (WMC) metric [8] to count methods inside classes as well as aspect operations (i.e. advices, methods and intertype methods). These metrics have been reported as good fault-proneness indicators in studies that comprised OO programs [19, 30]. Again, we considered their maximum and average values across releases for the clusters of modules required for the implementation of each concern.

Table 9 shows the statistics for the AOP mechanisms, LOC and WOM metrics in iBATIS. We again adopted the confidence level of 95%. Tables 10 and 11 present the results of the Spearman's correlation rank run against: (i) the maximum and average number of mechanisms across releases, and (ii) the total and average number of faults per concern across releases. The values and results comprising LOC and WOM metrics are also presented in these tables.

Note that the correlation coefficient is very large for all correlations that take into account the maximum and the average number of pointcuts and advices. In fact, we can observe that the correlation between the maximum number of pointcuts and advice and the average number of faults per concern is significant. We observed a 99% level of confidence (Table 10). For ITDs, the correlation coefficient varies between 0.5 and 0.6, what means moderate-to-large correlation on average if we consider a confidence level to 85%.

Table 10. Correlation with average number of faults per concern/release.

Metric	Coefficient	P-value
MAX-POINTCUTS	0.8809524**	0.0072420
MAX-ADVICES	0.8742672**	0.0045120
MAX-ITDs	0.5509081	0.1570000
MAX-LOC	0.1904762	0.6646000
MAX-WOM	0.1666667	0.7033000
AVG-POINTCUTS	0.8571429*	0.0107100
AVG-ADVICES	0.8571429*	0.0107100
AVG-ITDs	0.5714286	0.1511000
AVG-LOC	0.1904762	0.6646000
AVG-WOM	0.1666667	0.7033000

** correlation is significant at the 0.01 level

* correlation is significant at the 0.05 level

Differently from results of previous studies comprising OO programs [19, 30], LOC and WOM metrics show non-significant correlation with both average and maximum number of faults in our study. These results indicate that when we consider the set of modules involved in AO implementations of crosscutting concerns, the internal number of AOP-specific mechanisms (i.e. pointcuts, advices and ITDs) are good fault-proneness indicators. Moreover, they seem to be better indicators than the traditional LOC and WOM metrics.

While performing the analysis presented in this section, we noticed that: (i) concern-specific characteristics define the set of AOP mechanisms that must be used in conjunction to implement such a concern, and (ii) given a specific concern implementation, the usage rate of each mechanism tends to be directly proportional

Table 9. Number of AOP mechanisms, LOC and faults per concern in iBATIS.

	Pointcuts		Advices		ITDs		LOC		WOM		Faults	
	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Max	Avg	Total	Avg
Concurrency	7	6.75	5	4.5	3	2.75	1,605	1,435	395	373	1	0.25
Type Mapping	2	2	2	2	3	3	496	496	298	298	3	0.75
Design Patterns	9	6	5	3.5	14	9.5	1,725	1,566	448	391	2	0.5
Error Context	42	26.75	45	29.75	1	0.5	2,109	1,926	450	404	21	5.25
Exception Handling	77	70.25	75	69	82	74.75	4,761	4,490	1,159	994	28	7
Connection	64	64	64	32	61	60.5	901	890	359	358	17	8.5
Session	46	45.5	46	22.75	25	25	331	325	208	203	6	3
Transaction	20	20	20	10	54	53.5	686	684	223	221	3	1.5

to the number of faults associated with that concern, what is supported by the correlation test results. These findings support the H2 null hypothesis (H2-0) because the overall fault distribution per main AOP mechanism showed to be uniform. In addition, the usage rate of each mechanism does not vary independently. That is, it depends on the set of concerns aspectised within a system.

Table 11. Correlation with total number of faults per concern.

Metric	Coefficient	P-value
MAX-POINTCUTS	0.8263621*	0.0114400
MAX-ADVICES	0.8192771*	0.0128300
MAX-ITDs	0.3915663	0.3374000
MAX-LOC	0.3473116	0.3993000
MAX-WOM	0.3473116	0.3993000
AVG-POINTCUTS	0.8024096*	0.0165400
AVG-ADVICES	0.8024096*	0.0165400
AVG-ITDs	0.4191692	0.3013000
AVG-LOC	0.3473116	0.3993000
AVG-WOM	0.3473116	0.3993000

* correlation is significant at the 0.05 level

6. STUDY LIMITATIONS

This section discusses the study limitations based on the four categories of validity threats described by Wohlin et al. [33]. Each category includes a set of possible threats for an empirical study. We identified the items within each category that might threat our study, which are discussed in the following. For each category, we list possible threats and the measures we took to reduce each risk.

Conclusion validity. We identified two categories in this case: (i) *reliability of measures*: subjective decisions were made during the fault classification steps, specially regarding obliviousness levels (Section 5.1); besides, one of the target systems was evaluated with auto-generated test cases (Section 3.3.2), what might have risked the reliability of test results; and (ii) *random heterogeneity of subjects*: evaluated systems came from different application domains. To reduce risk (i), we designed detailed fault report forms and defined a set of guidelines that were followed in order to systematically classify each fault (Section 3.3.5). In regard to the evaluation based on auto-generated tests, this technique has previously yielded relevant results [37, 38], so it should not be seen as a major issue. Regarding risk (ii), although the applications' heterogeneity is considered a threat to the conclusion validity, it helps to promote the external validity of the study.

Internal validity. We detected two possible risks: (i) *ambiguity about direction of causal influence*: the complexity of the aspectised concerns might have made a system release more faulty than the others; and (ii) *history and maturation*: HW and MM systems have been extensively evaluated and continuously improved through the last years, what reduced the number of uncovered faults in such systems. Risk (i) cannot be completely avoided as each functionality differs from the others w.r.t. complexity. However, it was reduced because all systems were developed and revised by experienced programmers. They used implementation guides, design patterns or specific AOP idioms, where applicable. Moreover, systematic regression testing helped developers preserve the semantics of the OO counterparts. In order to reduce risk (ii), we focused our analyses on iBATIS, which consists in the most recent from all target systems and yielded the largest data set to be analysed.

Construct validity. We identified the following construct validity threats: (i) *inadequate operational explanation of constructs*: unclear procedures that should be followed in the event of a fault being uncovered might have biased the results (e.g. should the fault be fixed? How should this fault be classified?); (ii) *confounding constructs and levels of constructs*: different maturity levels of the investigated systems impacted the number of uncovered faults; and (iii) *interaction of testing and treatment*: iBATIS developers were aware of further system evaluation. To reduce risks (i) and (iii), we defined clear procedures and roles that were applied throughout all study steps. In particular, iBATIS development was strongly based on regression testing in order to make only error-free code available in the CVS repository. Although this approach made developers aware of the system evaluation procedures, it was important since it enabled developers to collect data since the early development phases. Risk (ii), on the other hand, could not be avoided due to the few options of medium-sized AO systems currently available for evaluation. Such systems present different levels of maturity, what includes varied fault rates.

External validity. The major risk here is related to the *interaction of setting and treatment*: the evaluated systems might not be representative of the industrial practice. To reduce this risk, we evaluated systems that come from heterogeneous application domains and are implemented with AspectJ, which is one of the representatives in the state of AOP practice. The iBATIS system is a widely-used open source project for object-relational mapping. Even though HW and MM are smaller applications, they are also heavily based on industry-strength technologies. In addition, both systems have been extensively used and evaluated in previous research [9, 13, 14, 16, 27]. To conclude, the characteristics of the selected systems, when contrasted with the state of practice in AO software development, represent a first step towards the generalisation of the achieved results.

7. CONCLUSIONS

This paper presented an exploratory study of the fault-proneness of AOP mechanisms used in the implementation of evolving AO programs. We analysed three systems from different application domains, from which we collected fault-related data upon which we performed our analyses. The results revealed the negative impact of obliviousness on the fault-proneness of programs implemented with AspectJ (H1 hypothesis). More recent methods and languages for AOP can help to ameliorate this problem. Examples of such approaches are aspect-aware interfaces [26], Crosscut Programming Interfaces (XPIs) [17] and Explicit Join Points (EJPs) [20]. Although they reduce the obliviousness among system modules, these approaches help to improve program comprehension by making aspect-base interaction more explicit. In particular, they tend to reduce the language-level obliviousness, which happened to be the category with the largest number of faults in our study.

As far as the H2 hypothesis is concerned, we did not confirm the common intuition that defining pointcuts is the most fault-prone scenario in AOP. There was no AOP mechanism that stood out as the main responsible for the detected faults. We also argue that this correlational study provides a good lead for more probing controlled experiments to investigate this issue further. Nevertheless, recent research on fault taxonomies and testing approaches for AO software has mainly focused on pointcuts as the key bottleneck in AOP [3, 4, 10, 12]. However, the results obtained for the H2 hypothesis motivate more intensive research on the testing support for other AOP mechanisms beyond pointcut expressions, such as intertype declarations and advice.

We believe that these study outcomes are helpful in several ways, such as: (i) providing information about harmful AOP mechanisms; (ii) supporting testing processes by pinpointing recurring faulty scenarios; and (iii) enhancing the general understanding towards robust AOP, so that other controlled experiments can be derived in our future research.

8. ACKNOWLEDGMENTS

We would like to thank the iBATIS AO developers Elliackin Figueiredo, Diego Araujo, Marcelo Moura and Mário Monteiro. We also thank Andrew Camilleri for his valuable help while analysing the MobileMedia system with the CIF framework.

The authors received full of partial funding from the following agencies and projects: *Fabiano Ferrari*: FAPESP (grant 05/55403-6), CAPES (grant 0653/07-1) and EC Grant AOSD-Europe (IST-2-004349); *Alessandro Garcia*: FAPERJ (distinguished scientist grant E-26/102.211/2009), CNPq (productivity grant 305526/2009-0 and Universal Project grant 483882/2009-7) and PUC-Rio (productivity grant); *Rachel Burrows*: UK EPSRC grant; *Otávio Lemos*: FAPESP (grant 2008/10300-3); *Sergio Soares*: CNPq (grant 309234/2007-7) and FACEPE (grant APQ-0093-1.03/08); *José Maldonado*: EC Grant QualiPSo (IST-FP6-IP-034763) and CNPq; *Other authors*: CAPES and CNPq, Brazil.

9. REFERENCES

[1] Aldrich, J. 2004. Open Modules: Reconciling Extensibility and Information Hiding. In: SPLAT AOSD'04 Workshop.

[2] Alexander, R. T., et al. 2004. Towards the Systematic Testing of Aspect-Oriented Programs. Report CS-04-105, Colorado State University, Fort Collins-USA.

[3] Anbalagan, P., and Xie, T. 2008. Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs. In: ISSRE'08. 239-248.

[4] Bækken, J. S., and Alexander, R. T. 2006. A Candidate Fault Model for AspectJ Pointcuts. In: ISSRE'06. 169-178.

[5] Cacho, N., Filho, F. C., Garcia, A., and Figueiredo, E. 2008. EJFlow: Taming Exceptional Control Flows in Aspect-Oriented Programming. In: AOSD'08. 72-83.

[6] Camilleri, A., Coulson, G., Blair, L. 2009. CIF: A Framework for Managing Integrity in Aspect-Oriented Composition. In: TOOLS'09. 16-26.

[7] Ceccato, M., and Tonella, P. 2004 Measuring the Effects of Software Aspectization. In: 1st Workshop on Aspect Reverse Engineering (ARE).

[8] Chidamber, S.R., and Kemerer, C.F. 1994. A Metrics Suite for Object-Oriented Design. IEEE Transactions on Software Engineering 20 (6). 476-493.

[9] Coelho, R., et al. 2008. Assessing the Impact of Aspects on Exception Flows: An Exploratory Study. In: ECOOP'08. (LNCS, vol. 5142). 207-234.

[10] Delamare, R., Baudry, B., Ghosh, S., Le Traon, Y. 2009. A Test-Driven Approach to Developing Pointcut Descriptors in AspectJ. In: ICST'09. 376-385.

[11] Eaddy, M., et al. 2008. Do Crosscutting Concerns Cause Defects? IEEE Transactions on Software Engineering 34 (4). 497-515.

[12] Ferrari, F., Maldonado, J., and Rashid, A. 2008. Mutation Testing for Aspect-Oriented Programs. In: ICST'08. 52-61.

[13] Figueiredo, E., et al. 2008. Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability. In: ICSE'08. 261-270.

[14] Filho, F. C., et al. 2006. Exceptions and Aspects: The Devil is in the Details. In: FSE'06. 152-162.

[15] Filman, R. E., and Friedman, D. 2004. Aspect-Oriented Programming is Quantification and Obliviousness. In: Aspect-Oriented Software Development. Addison-Wesley.

[16] Greenwood, P., et al. 2007. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: ECOOP'07 (LNCS, vol.4609). 176-200.

[17] Griswold, W. G., et al. 2006. Modular Software Design with Crosscutting Interfaces. In: IEEE Software 23(1). 51-60.

[18] Gybels, K., and Brichau, J. 2003. Arranging Language Features for More Robust Pattern-Based Crosscuts. In: AOSD'03. 60-69.

[19] Gyimóthy, T., Ferenc, R., and Siket, I. 2005. Empirical Validation of Object-Oriented Metrics on Open Source Software for Fault Prediction. IEEE Transactions on Software Engineering 31 (10). 897-910.

[20] Hoffman, K., and Eugster, P. 2007. Bridging Java and AspectJ through Explicit Join Points. In: PPPJ'07. 63-72.

[21] Hopkins, W. G. 2003. A New View of Statistics. Sport Science, http://www.sportsci.org/resource/stats (01/09/2009)

[22] iBATIS Data Mapper - http://ibatis.apache.org/ (01/09/2009).

[23] Johnson, R., et al. 2007. Spring - Java/J2EE application framework. Ref, Manual Version 2.0.6, Interface21 Ltd.

[24] Kastner, C., Apel, S., and Batory, D. 2007. A Case Study Implementing Features Using AspectJ. In: SPLC'07. 223-232.

[25] Kiczales, G., et al. 1997. Aspect-Oriented Programming. In: ECOOP'97 (LNCS, vol. 1241). 220-242.

[26] Kiczales, G., and Mezini, M. 2005. Aspect-Oriented Programming and Modular Reasoning. In: ICSE'05. 49-58.

[27] Soares, S., Laureano, E., and Borba, P. 2002. Implementing Distribution and Persistence Aspects with AspectJ. In: OOPSLA'02. 174-190.

[28] Steimann, F. 2006. The Paradoxical Success of Aspect-Oriented Programming. In: OOPSLA'06. 481-497.

[29] Stoerzer, M., and Graf, J. 2005. Using Pointcut Delta Analysis to Support Evolution of Aspect-Oriented Software. In: ICSM'05. 653-656.

[30] Subramanyam, R., and Krishnan, M. S. 2003. Empirical Analysis of CK Metrics for Object-Oriented Design Complexity: Implications for Software Defects. IEEE Transactions on Software Engineering. 29 (4). 297-310.

[31] Sullivan, K., et al. 2005. Information Hiding Interfaces for Aspect-Oriented Design. In: ESEC/FSE'05. 166-175.

[32] The AspectJ Project. http://www.eclipse.org/aspectj/

[33] Wohlin, C., et al. 2000. Experimentation in Software Engineering - An Introduction. Kluwer Academic Publishers.

[34] Xie, T., and Zhao, J. 2006. A Framework and Tool Supports for Generating Test Inputs of AspectJ Programs. In: AOSD'06. 190-201

[35] Huang, S. S., Smaragdakis, Y. 2006. Easy Language Extension with Meta-AspectJ. In: ICSE'06. 865-868

[36] Zhang, S., and Zhao, J. 2007. On Identifying Bug Patterns in Aspect-Oriented Programs. In: COMPSAC'07. 431-438.

[37] Csallner, C., and Smaragdakis, Y. 2005. Check 'n' crash: combining static checking and testing. In: ICSE'05, 422-431.

[38] Harman, M., et. al. 2009. Automated test data generation for aspect-oriented programs. In: AOSD'09. 185-196.

Appendix

B

Paper: Characterising Faults in Aspect-Oriented Programs: Towards Filling the Gap between Theory and Practice

This appendix presents the full contents of a paper published in the Proceedings of the 24^{th} Brazilian Symposium on Software Engineering (SBES'10). An overview of this work was provided in Chapter 3 of this dissertation. A copyright notice in regard to it is next shown.

IEEE COPYRIGHT NOTICE[1]**:** "*Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.*"

[1] `http://www.ieee.org/portal/cms_docs/pubs/transactions/auinfo03.pdf` - last accessed on 17/08/2010.

Characterising Faults in Aspect-Oriented Programs: Towards Filling the Gap between Theory and Practice

Fabiano C. Ferrari*, Rachel Burrows†◇, Otávio A. L. Lemos‡, Alessandro Garcia† and José C. Maldonado*
*Computer Systems Department – University of São Paulo (ICMC-USP) – São Carlos – Brazil
Email: {ferrari,jcmaldon}@icmc.usp.br
†Informatics Department – Pontifical Catholic University of Rio de Janeiro (PUC-Rio) – Rio de Janeiro – Brazil
Email: r.burrows@comp.lancs.ac.uk, afgarcia@inf.puc-rio.br
‡Department of Science and Technology – Federal University of São Paulo (UNIFESP) – S. J. Campos – Brazil
Email: otavio.lemos@unifesp.br

***Abstract*—Since the proposal of Aspect-Oriented Programming, several candidate fault taxonomies for aspect-oriented (AO) software have been proposed. Such taxonomies, however, generally rely on language features, hence still requiring practical evaluation based on realistic implementation scenarios. The current lack of available AO systems for evaluation as well as historical data are the two major obstacles for this kind of study. This paper quantifies, documents and classifies faults uncovered in several releases of three AO systems, all from different application domains. Our empirical analysis naturally led us to revisit and refine a previously defined fault taxonomy. We identified particular fault types that stood out amongst the categories defined in the taxonomy. Besides this, we illustrate recurring faulty scenarios extracted from the analysed systems. We believe such scenarios should be considered for the establishment of testing strategies along the software development process.**

I. Introduction

Software faults are artefacts that have been widely studied by researchers over the years. Among other types of study, some authors have analysed how specific programming features can be sources of faults in software systems [1]. Others have empirically studied how different types of faults appear in the context of real software development projects [2, 3, 4]. This type of fault characterisation is important because it provides empirical evidence on how software faults actually occur, as opposed to theoretical fault taxonomies that are based solely on the characteristics of programming languages or development approaches.

Specifically for Aspect-Oriented Programming (AOP) [5], which is a contemporary software development approach, most of research related to software faults has targeted the classification of faults based on programming features [6], but not on empirical analysis of real software development data. Consequently, there is an open research question related to the characterisation of real faults in AOP. So far, the several candidate fault taxonomies, [7, 8, 9, 10, 11, 6, 12] and bug pattern catalogues [13, 14] are either only based on language features or limited to specific programming mechanisms. Thus, there is still a lack of studies that quantify and, equally importantly, characterise faulty implementation scenarios in AO programs. Results of such kind of studies could motivate the definition of fault models for AO software as well as the adoption of AOP in real-world software projects.

Although a number of industry-strength frameworks that support or employ AOP are emerging, such as JBoss AOP and Demoiselle [15], we can still notice some scepticism of practitioners; this results in the cautious adoption of AOP by the industry [16]. Therefore, it is still hard to find AOP-based software projects with systematic documentation of faults. Nevertheless, building a body of knowledge about recurring mistakes programmers make while developing AO software is an important matter that would enable, mainly: (i) a better understanding of harmful AO implementation scenarios; and (ii) the definition of testing strategies with focus on recurring faulty scenarios.

Concerned with the aforementioned issues, this paper presents the results of a study of faults in AO programs which has the following main goals: (i) quantifying and categorising faults in AO programs according to a fine-grained fault taxonomy; and (ii) characterising recurring faulty scenarios of AOP. Our empirical analysis naturally resulted in a refinement a previously defined taxonomy [6] based on recurring observed fault scenarios. To achieve these goals, we analysed several releases of three available AO systems and, by means of testing and code analysis, we quantified and reported faults during varied evaluation phases. Each analysed system comes from a different application domain and has been extensively analysed in previous research [17, 18, 14, 19, 20, 21]. We classified the faults uncovered from these systems according to the proposed fault taxonomy. The fault types included in the taxonomy are split into four categories, which encompass faults that are likely to result from the inherent complexity and consequent misuse of the main AOP mechanisms.

The results revealed that a subset of fault types stood out when compared to faults within a given category. We also identified and described implementation scenarios that are representatives for them. We believe these results can be of help for the definition of testing strategies that focus on particular implementation scenarios. In a more general perspective, these results represent a step towards enhancing the general understanding about robust AOP, from which controlled experiments can be derived.

◇Also affiliated with Lancaster University, UK.

The remainder of this paper is organised as follows: Section II presents the background for the research. Section III brings details of the fault taxonomy for AO software. The study setup is described in Section IV. The results of the fault quantification and characterisation are presented in Sections V and VI, respectively. The limitations of this study and a summary of related research come in Section VII. Finally, Section VIII brings our conclusions and future work.

II. Background

A. AOP and AspectJ

Aspect-Oriented Programming (AOP) [5] arose in the late 90's as a possible solution to enhance software modularisation. It was mainly motivated by the fact that traditional approaches (e.g. procedural and object-oriented programming) could not satisfactorily cope with crosscutting concerns (hereafter called *cc-concerns*). Non-cc-concerns, on the other hand, compose the *base code* of an application and comprise the set of functionalities that can be modularised within conventional implementation units (e.g. classes and data structures).

AOP is strongly based on the idea of separation of concerns (SoC), which claims that computer systems are better developed if their several concerns are specified and implemented separately [22]. Such separation is achieved by means of new conceptual[1] modular units called *aspects* [5], which encapsulate code that usually appears either scattered over several modules in a system or tangled with code that realise other concerns – i.e. aspects encapsulate the cc-concerns.

In an AOP system, the behaviour implemented within an aspect runs when specific events – the *joint points* (JPs) – occur during the program execution. Typical examples of JPs are a method call, a method execution or a field access. In AspectJ [23], which is the most popular Java-based AOP language, a *pointcut* expression (or *pointcut designator* – PCD) selects a set of JPs by means of declarative expressions. A PCD is generally formed by patterns (e.g. method signatures) and predicates. The JP model together with the PCD implement a quantification mechanism that enables crosscutting behaviour to run at several JPs during the software execution. The PCDs are bound to *advices*, which consist of method-like portions of code that implement crosscutting behaviour. AspectJ also allows static modifications of class structure and hierarchy through inter-type declarations (ITDs) and other `declare`-like expressions. For a complete list of AspectJ features, the reader may refer to the AspectJ project website[2].

B. Fault Taxonomies for AO Software

The concepts and elements introduced by AOP represent new potential sources of faults [7], hence posing new challenges for software quality assurance activities such as testing and debugging. For example, faults can arise from crosscutting behaviour or from aspects–base program interactions. Concerned with these issues, several authors have proposed AOP-specific fault taxonomies during the last years.

[1]An aspect can be a conceptual unit since symmetric AOP approaches ideally do not require new software entities to implement crosscutting behaviour.

[2]http://www.eclipse.org/aspectj/ - accessed on 28/07/2010

The first initiative was presented by Alexander et al. [7], who identified possible sources of faults in AO programs. For instance, a fault may reside in a portion of the base program not affected by an aspect, or it may be related to an emergent property created by aspect-base program interactions. Besides, the authors proposed a high-level fault taxonomy for AspectJ-like programs. For instance, incorrect strength of PCD patterns may lead to faulty PCDs that select unintended sets of JPs. Other types of faults include the arbitrary execution order of advices that share common JPs and failures to establish expected post-conditions and invariants, when an aspect break contracts established in the software specification. Other authors [11, 13, 24] also defined AOP-specific fault types which partially overlap Alexander et al.'s taxonomy [7], although also characterising additional fault types.

In the next section, we present a comprehensive fault taxonomy for AO software that is based on an extensive survey of AOP-specific fault types. We briefly introduced this taxonomy in our previous research [6] and discussed possible generalisations for varied AOP supporting technologies. In the next section we provide more details of each fault type and generic examples of how they can appear in AO programs. This taxonomy is used in the remaining sections of this paper to support quantitative and descriptive analysis or faults extracted from real-world AO systems.

III. A Fault Taxonomy for AO Software

In our previous research [6], we introduced a fault taxonomy for AO software that was built upon the results of a systematic literature review of AO testing [25, 26]. The taxonomy includes fault types that have been characterised by several authors along with some types identified by ourselves. They are distributed across four main categories which were defined based on the main elements involved in an AO software, namely: (1) pointcut expressions; (2) ITDs and other declarations; (3) advice definitions and implementations; and (4) the base program.

In this section we refine the description of each fault type along with simple examples of how the faults can occur in AO programs. Real examples that we extracted from AspectJ systems and classified according to the taxonomy are presented further in Section VI. We call the reader's attention to the fact that, to date, research on fault taxonomies for AO software has mostly been based on researchers' expertise and relies on mechanisms and characteristics of AspectJ-like languages. Nevertheless, we could observe that AO software implemented with support of other AOP technologies (e.g. Spring AOP and JBoss AOP) is also prone to the majority of the characterised fault types. More information about the generalisation of the taxonomy can be found elsewhere [6].

Group F1 – Faults related to PCDs: faults of this type are mostly related to sentences that define PCDs. For instance, a fault may occur due to the misuse of a wildcard or due to an incorrect pattern definition, resulting in incorrect JP matchings. The following fault types belong to this group:

F1.1: Selection of a superset of intended JPs. *Example:* a PCD is defined to pick JPs from a type `A`, however the intended type is a specific subtype of `A`. In this

case, other subtypes of A that also offer similar JPs are affected as well.

F1.2: Selection of a subset of intended JPs. *Example:* a PCD is expected to match all executions of methods of a type A, however the PCD is incorrectly defined thus matching only some of A's method executions (e.g. due to a missing "$\star$" wildcard).

F1.3: Selection of a wrong set of JPs, which includes both intended and unintended items. *Example:* a PCD is defined to pick JPs from two types A and B, however only JPs of A should be selected.

F1.4: Selection of a wrong set of JPs, which includes only unintended items. *Example:* a PCD is defined to pick JPs from a type B, however only JPs from a different type (let us say A) should be selected.

F1.5: Incorrect use of a primitive PCD. *Example:* the `execution` primitive PCD should have been used in place of a `call`, thus resulting in undesired execution context.

F1.6: Incorrect PCD composition rules. *Example:* A compound PCD P_1 is formed by two other PCDs P_2 and P_3, however P_1 should be composed by P_2 and another PCD P_4.

F1.7: Incorrect JP matching based on exception throwing patterns. *Example:* The signature of an advice includes exceptions of types Ex_1 and Ex_2 that might be thrown by that advice. However, the intended advised JPs only throw exceptions of type Ex_3. The side-effect of this fault is a selection of a subset of JPs (i.e. F1.2).

F1.8: Incorrect JP matching based on dynamic circumstances. *Example:* a PCD contains an `if` primitive PCD whose predicate is incorrectly defined, thus resulting in incorrect JP matching during the system execution.

Group F2 – Faults related to ITDs or other declare-like expressions: in general, faults of this group may occur due to incorrect static modifications of the base code made by aspects. They might result from the lack of knowledge about the structure of the base program (e.g. its class hierarchy) and might be detected through static analysis of the code. The following fault types belong to this group:

F2.1: Improper method introduction, resulting in unanticipated method overriding or not resulting in anticipated method overriding. *Example:* a method with incorrect name `m` is introduced into a class `C`, thus overriding an original method within `C`.

F2.2: Introduction of a method into an incorrect class. *Example:* a method `m` is unexpectedly introduced into a class `C`, thus resulting in undesired functionality available from `C` and possibly for its children.

F2.3: Incorrect change in class hierarchy through parent declaration clauses V (e.g. "`declare parents`" statements), resulting in unintended inherited behaviour for a given class. *Example:* a class C_1 is incorrectly declared as child of a class C_2.

F2.4: Incorrect method introduction, resulting in unexpected method overriding. *Example:* differently from fault F2.1, a method `m` has the correct name, however `m` is introduced into an unexpected class `C`, thus incorrectly overriding the `C`'s original method.

F2.5: Omitted declared interface or introduced interface which breaks object identity. *Example:* an aspect `A` misses a `declare parents` statement that would specify an interface that should be implemented by a given class.

F2.6: Incorrect changes in exception-dependent control flow, resulting from aspect-class interactions or from clauses that alter exception severity. *Example:* An exception `e`, which may be thrown at a specific JP, is softened (e.g. by a `declare soft` statement) and no specific handler is defined for it, thus `e` becomes uncaught.

F2.7: Incorrect or omitted aspect precedence declaration. *Example:* a JP is advised by two advices a_1 and a_2, however a_1 and a_2 run in arbitrary order.

F2.8: Incorrect aspect instantiation rules and deployment, resulting in unintended aspect instances. *Example:* a new instance of an aspect `A` is incorrectly created for every JP selected by a PCD (e.g. by a `perthis` clause).

F2.9: Incorrect policy enforcement rules supported by warning and error declarations. *Example:* a compilation warning message is shown when specific JPs are matched in the base program (e.g. by a `declare warning` clause).

Group F3 – Faults related to advice definition and implementation: faults of this type may occur due to misunderstandings of system requirements or due to incorrect definition of advices. The following fault types belong to this group:

F3.1: Incorrect advice type specification *Example:* an advice which should run before a JP is actually running after the JP.

F3.2: Incorrect control or data flow due to incorrect aspect-class interactions. *Example:* incorrect execution of the base program behaviour through the invocation of the `proceed` statement.

F3.3: Incorrect advice logic, resulting in invariants violations or failures to establish expected postconditions. *Example:* portions of the base program code are refactored out to an advice `a`, however `a` fails to behave as expected.

F3.4: Infinite loops resulting from interactions among advices. *Example:* a circular dependency between two advices a_1 and a_2 is created, thus resulting in infinite loops.

F3.5: Incorrect access to JP static information. *Example:* unintended static information is extracted from a JP (e.g. by accessing the `thisJoinPointStaticPart` special variable) and used within the advice.

F3.6: Advice bound to incorrect PCD. *Example:* an advice `a` is bound to a PCD p_1 instead of the intended PCD p_2, thus `a` runs at unintended JPs.

Group F4 – Faults related to the base program: faults of this type may occur due to the lack of knowledge about aspects which will be woven into the base program or due to

the software evolution process.

F4.1: The base program does not offer required JPs in which one or more foreign aspects were designed to be applied. *Example:* an aspect `A` is expected to be applied to a class `C` that extends the base program, however `C` does not offer the intended JPs.

F4.2: The software evolution causes PCDs to break. *Example:* an AO system evolves and a PCD defined in an aspect `A` becomes obsolete, no longer picking out JPs that `A` should advise.

F4.3: Other problems related do base programs such as inconsistent refactoring or duplicated crosscutting code. *Example:* an exception handler is refactored out to an aspect, however it is also left (i.e. duplicated) in the base program, thus the original handler becomes obsolete (i.e. unreachable).

Additional Notes: Recently, some authors [14, 27] have proposed additional fault taxonomies and bug pattern catalogues that were not available by the time we defined the above fault categories and types. Nevertheless, the fault types characterised by them either can be directly mapped to our taxonomy or consist in complex faulty implementation scenarios that can be decomposed into simpler faults that belong to one of the aforementioned groups. For example, Coelho et al. [14] defined a set of bug patterns related to exception handling in AO systems. One of these patterns, the *Inactive Aspect Handler*, has as its main cause an incorrect pointcut definition that can be mapped to the faults F1.3 or F1.4. Another bug pattern, the *Obsolete (or Outdated) Handler in the Base Code*, consists in a typical inconsistent refactoring, thus mapped to the fault F4.3.

IV. Collecting Faults from AO Programs

A. Target Systems

The three medium-sized applications used in this study are from significantly different application domains. The first application used in this study, iBATIS [28], is a Java-based open source framework for object-relational data mapping. The second application is HealthWatcher (HW) [21], a typical Java web-based information system. The third is MobileMedia (MM) [20], a software product line for mobile devices that allows users to manipulate image files in different mobile devices. The choice for these applications was motivated by the wide range of different fine-grained and coarse-grained changes (e.g. refactorings and functionality increments or removals) found across the selected releases.

Along the evolution of these applications, a subset of cc-concerns, both functional and non-functional, was aspectised [19, 20, 21]. Examples of such concerns include certain design patterns, persistence, exception handling (all applications); concurrency (HW and iBATIS) and other application-specific concerns such as error context (iBATIS) or copy media (MM). Besides, the three systems are rich in kinds of non-cc-concerns and cc-concerns.

From hereafter, we refer to the AO versions of the target systems by their simple names or abbreviations, i.e. iBATIS, HW and MM. Four releases of iBATIS were considered in our evaluation, namely iBATIS 01, 01.3, 01.5 and 02. We also analysed four HW releases – HW 01, 04, 07 and 10 – and four MM releases – MM 01, 02, 03 and 06. Table I shows some general characteristics of the three target applications. For more information about each of them, the reader may refer to the respective placeholder websites or to previous reports of these systems [19, 20, 21].

TABLE I
Target systems

	iBATIS	HealthWatcher	MobileMedia
Application type	data mapper framework	health vigilance application	product line for mobile data
Code availability	Java/AspectJ	Java/AspectJ	Java/AspectJ
# of releases	60 / 4	10 / 10	10 / 10
Selected releases	4	4	4
Avg. KLOC	11	6	3
Avg. # of modules*	264	132	39
Avg. # of aspects	46	23	10
Evaluation procedure	testing	testing	interference analysis

*Interfaces, classes and aspects.

The evaluation procedures were defined according to the system characteristics and available information as follows:

1) iBATIS Evaluation: The iBATIS system has experienced two testing phases: pre-release and post-release testing. Developers aimed at producing defect-free code to be committed to a CVS repository. Pre-release testing was performed by executing original OO JUnit tests available with the application. Any abnormal behaviour when regressively testing the AO version of a given release was investigated. When a fault was discovered, it was documented in an appropriate detailed report. Post-release testing, on the other hand, aimed at assessing the implementation through the enhancement of the original test sets. The enhanced tests were executed against both OO and AO versions of a given release. A fault should only be reported if it was noticed in the AO version but not in the OO counterpart. This procedure ensured that only faults introduced during the aspectisation process would be reported for further analysis.

2) HW Evaluation: HW was tested in a single phase in our study given that initial tests have already been performed during its development. However, people who were involved in the original tests were unaware of further analyses as the one performed in this paper, thus faults have not been properly documented. Hence, the testing of HW was extended to cover the assessment (post-release) phase, thereby improving the degree of test coverage. Differently from iBATIS, however, no original test set was made available. Thus a full test set was built from scratch based on the system specification and code documentation. In order to reduce test effort and avoid systematic bias during test creation, test cases were automatically generated with adequate tooling support. As well as for iBATIS, a fault should only be reported if it was noticed in the AO version but not in the OO counterpart, and all uncovered faults were similarly reported.

3) MM Evaluation: As well as HW, MM has not been developed with awareness of further fault-based evaluation. Moreover, post-release tests during the system evolution and

maintenance only revealed faults related to robustness (e.g. data input validation), however not necessarily being related to AOP mechanisms. Despite this, we have evaluated MM using the Composition Integrity Framework (CIF) [29]. CIF helped us identify problems in aspect interactions established either between aspects and base code or among multiple aspects. Since MM is a software product line and includes mandatory, optional and alternative features, CIF was applied in varied configurations of MM. In doing so, we were able to derive a set of faults that resulted from a broad range of aspect-oriented compositions.

Every fault identified during pre-release or post-release testing was documented in a customised report form. During the assessment phase, the testers provided as much information as possible, with special attention to the test case(s) that revealed the fault (if applicable) and the fault symptom. In addition, the tester provided some hints about the fault location. Then, this report was forwarded to the original developers, who were responsible for concluding the fault documentation.

B. Fault Classification Procedures

After the fault documentation step, each fault was classified according to the fault taxonomy for AO software proposed in Section III. The customised fault report form included appropriate fields for this step.

In order to systematically classify every fault, we took into account the fault origin (or *root cause*) and not only consequent side-effects. The classification based on the first three categories (i.e. PCD-, ITD- and advice-related) was generally straightforward. For instance, it was relatively easy to identify mismatching PCDs (e.g. fault types F1.1 and F1.2), misuse of `declare`-like expressions (e.g. fault types F2.6 and F2.7) and incorrect advice types (e.g. fault type F3.1). Base program-related faults, on the other hand, required more in-depth analysis and reasoning about them. For example, base code changes that result in broken PCDs should be classified as base program-related (i.e. Group 4) although their side-effects consists in unmatched JPs, what might lead to such faults being classified within Group 1, i.e. PCD-related faults.

V. Quantifying Faults in AO Programs

Performing the data collection procedures described in the Section IV allowed us to identify a total of 104 faults from the three target systems. Note that the highest number of faults in iBATIS was expected because the other two systems have already-stable, more mature implementations. They have been available for more than four years and underwent several corrective and perfective changes.

HW and MM have also been targeted by previous assessments of other equally-important quality attributes [14, 20, 30, 21]. Thus, as the AO versions have different degrees of stability, the results obtained from them provided support for drawing more general findings.

The general fault distribution is displayed in Table II[3] and depicted in Figure 1. We can see that when we consider a coarse-grained fault categorisation – in this case based on the main AOP mechanisms (highlighted in Table II) – we can conclude that the mechanisms similarly impact the correctness of evolving AO programs. This was demonstrated with statistical significance for the analysed systems [19]. We next present the results of a refinement of this fault classification, which consists in one of the contributions of this paper. For that, we apply the full fine-grained taxonomy described in Section III.

TABLE II
Fault distribution in all systems

	System			Total
	iBATIS	HW	MM	
PCD-related	18	2	1	21
ITD-related	12	7	4	23
Advice-related	15	5	4	23
Base program-related	35	1	0	36
Total	80	15	9	104

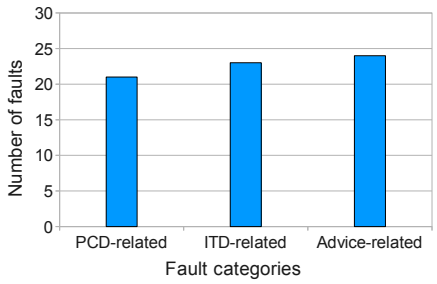


Fig. 1. Fault distribution per main AOP mechanism.

A. Fine-Grained Fault Classification

We applied our fine-grained fault taxonomy to the faults earlier classified according to the main AOP mechanisms. The obtained fault distribution is presented in Table III and is depicted in Figure 2[4]. Despite the similar fault distribution across the main AOP constructs observed in Figure 1, we can observe in Figure 2 that the fault counts are not evenly distributed across the several types included in the taxonomy. However, we can observe that the main elements are used in similar amounts in the systems. For example, the average number of PCDs, ITDs and advices in iBATIS is 177, 173 and 160, respectively.

Even when we look at a particular group we can notice large differences in the amounts. For example, while some fault types within Group 3 (i.e. advice-related faults) show totals of 15 (F3.3) and 5 (F3.1), other types from the same group vary between 0 and 1 (e.g. F3.4, F3.5 and F3.6).

This suggests that, even though the coarse-grained AOP mechanisms present similar fault-proneness [19], they may negatively impact the correctness of an AO system in particular usage scenarios. In the remaining of this section, we point out and analyse some key differences we observed in the fault distribution. The analysis develops from four specific viewpoints: (i) the fault counts that stood out; (ii) the fragile pointcut problem; (iii) static versus dynamic crosscutting behaviour; and (iv) advice execution order.

[3]The total numbers of faults displayed for iBATIS and HW slightly differ from the original totals [19]. The changes resulted from a revision of the fault documentation/classification. Nevertheless, these differences has no significant impact on the achieved results and conclusions.

[4]Note that in Figure 2 we omitted the "F" from each fault type label in order to improve the chart's readability.

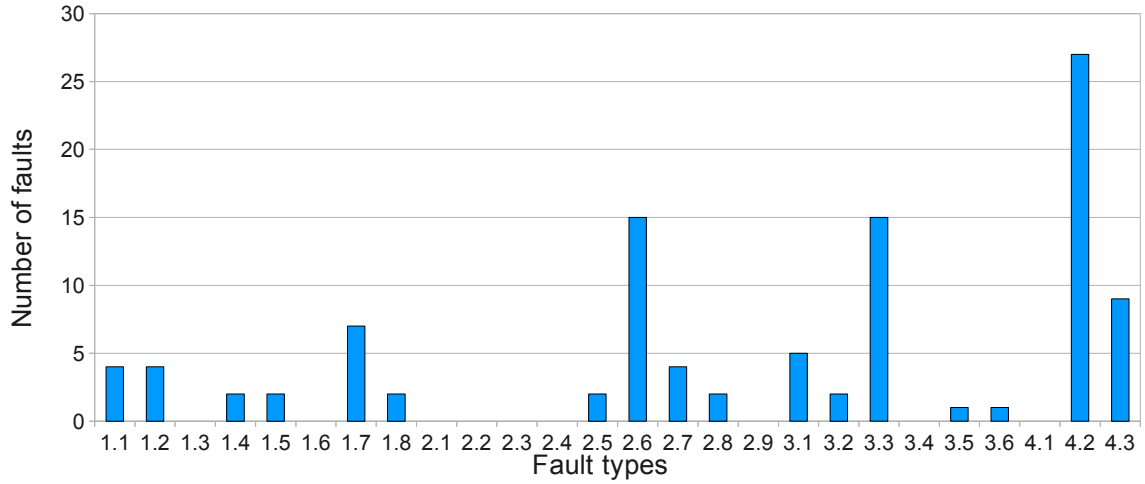


Fig. 2. Fault distribution per type.

TABLE III
FAULTS UNCOVERED IN THE 3 TARGET SYSTEMS.

Fault type		System			Total
		iBATIS	HW	MM	
PCD-related	F1.1	4	–	–	4
	F1.2	4	–	–	4
	F1.3	–	–	–	0
	F1.4	1	–	1	2
	F1.5	2	–	–	2
	F1.6	–	–	–	0
	F1.7	7	–	–	7
	F1.8	–	2	–	2
Subtotal		18	2	1	21
ITD-related	F2.1	–	–	–	0
	F2.2	–	–	–	0
	F2.3	–	–	–	0
	F2.4	–	–	–	0
	F2.5	2	–	–	2
	F2.6	8	7	–	15
	F2.7	–	–	4	4
	F2.8	2	–	–	2
	F2.9	–	–	–	0
Subtotal		12	7	4	23
Advice-related	F3.1	1	–	4	5
	F3.2	–	2	–	2
	F3.3	12	3	–	15
	F3.4	–	–	–	0
	F3.5	1	–	–	1
	F3.6	1	–	–	1
Subtotal		15	5	4	24
Base program-related	F4.1	–	–	–	0
	F4.2	27	–	–	27
	F4.3	8	1	–	9
Subtotal		35	1	0	36
Overall total		80	15	9	104

Fault types that stood out: Within each category, some individual fault counts stood out. In particular, faults types F1.7, F2.6, F3.3 and F4.2 showed the highest number of occurrences when we consider the three analysed systems. Together, they account for 61% of all faults reported in our study. The three first types are analysed in the sequence, while F4.2 is discussed further in this subsection.

Aspectising concerns such as exception handling (iBATIS, HW and MM) and error context (iBATIS) required several portions of code being moved from classes to aspects as well as the definition of PCDs and advices to properly deal with exceptional scenarios. Identifying JPs that include exception flow information resulted in several faults in the AO versions of iBATIS. For example, faults of type F1.7, which represent 7% of the total number of identified faults, regard JP selection based on exception throwing patterns. Mistakes in the pattern definition resulted in inappropriate JP matchings and consequent malfunction of the system under abnormal execution circumstances (e.g. activation of advices that should help the system recover in the event of erroneous execution).

Another recurring problem identified in iBATIS and HW regards to use of the exception softening constructs available in AspectJ. The misuse of these constructs is the root cause of faults of type F2.6, what represents 14% of all faults.

We highlight that, according to previous reports of these systems [19, 20, 21], the aspectisation of exception handlers has as far as possible followed good practices described in a cookbook for error handling with aspects [31]. However, in spite of the difficulty of testing exception handling-related code, it is typically overlooked by testing approaches, therefore developers can fail in assuring that such code is fault-free even with the availability of cookbooks. As a consequence, using such constructs has been pointed out as a problematic side-effect of aspects affecting exception-dependent code. For example, Coelho et al. [14] characterised a bug pattern named *"Unstable Exception Interface"* which states that aspect have the "ability" of destabilising the exception interface of advised methods. Therefore, even considering the benefits possibly achieved with the use of aspects for exception handling scenarios, specially regarding separation of concerns [30], we noticed that aspects as exception handlers harm the robustness of a system because: (i) several faults were associated with the related constructs (see Table III); and (ii) previous research has pointed out a large number of uncaught exceptions crosses the system's boundaries of AOP implementations [14].

In regard to faults of type F3,3, they also represent 14% of all faults collected from the three systems. In iBATIS, similarly to types F1.7 and F2.6, they are mostly associated with a specific concern: error context. This concern has as its main characteristic the high interactivity between base program and

aspects, what includes the exposure of context variables and their manipulation. One particular characteristic of faults of type F3.3 is that their isolation in both systems (iBATIS and HW) required more in-depth analysis than faults of types F1.7 and F2.6. While some faults of these last two types could be detected through static analysis of the code (e.g. to identify JP matchings), F3.3 instances could only be detected by means of testing and debugging.

The fragile pointcut problem: Faults of type F4.2 represent 26% of all faults documented in our study. They were mainly caused by evolutionary changes within the base program, what resulted in mismatching PCDs. This problem is known as the *fragile pointcut problem* [32], which leads to faults during software maintenance activities. It was particularly noticed in the iBATIS system due to one specific reason: iBATIS was the only system which experienced pre-release testing; its development strategy required that every noticed fault should be fixed before the code was committed to the CVS repository. These faults were identified in releases 01.5 and 02 of iBATIS due to the intensive code refactoring required for the aspectisation of persistence-related concerns, which in iBATIS represent a set of database-related functional requirements.

One might argue that faults related to fragile PCDs should be assigned to Group 1 (i.e. PCD-related faults). However, given that a PCD `p` is correctly defined according to the specification, any modification of the base program that breaks `p`'s functionality should also imply a revision of `p`. If this revision is not carried out, a fault may arise in the system. However, it should not be "blamed" on `p` but on the base program itself, since `p` still conforms to the original specification.

Static versus dynamic crosscutting: We noticed significant differences between the fault counts associated with elements that implement the two models of crosscutting in the analysed systems: static and dynamic crosscutting [33]. While the former is generally realised through ITDs (e.g. class member introductions), the latter is implemented within advices that are triggered at runtime.

Faults related to static crosscutting range from types F2.1 to F2.5. Only 2 occurrences were identified in the iBATIS system. Such low number can be explained by the fact that, in general, those fault types can be detected at compilation time (e.g. a missing class member that should have been introduced by an aspect). Advice-related faults, on the other hand, account for 23% of the total (24 out of 104 faults). Differently from faults related to static crosscutting, they can mainly be detected during the software execution.

Advice execution order: The aspect interference analysis performed in the MM system revealed 4 faults of type F2.7 and 4 of type F3.1. They are all related to the advice execution order. One characteristic of this system is that independent features implemented within aspects (e.g. photo and SMS) share common JPs in the base program. In iBATIS and HW, on the other hand, despite the occurrences of shared JPs, they are usually shared between aspects that implement inter-related concerns, e.g. exception handling and error context in iBATIS, and design patterns in HW. These awareness among concerns might have reduced the frequency of faults of this nature.

VI. Recurring Faulty Scenarios and Examples

This section presents examples of recurring faults we identified during the course of this study. We selected representatives of all fault groups defined in Section III, extracted from the three analysed systems. Additionally, we enumerate the main steps that might have led to such faults and, therefore, should be double-checked in order to reduce the risks of new fault introductions.

A. Pointcut-Related Faults

As described in Section III, PCD-related faults typically come from unintended JPs matchings. Despite the visual aid provided by existing IDEs w.r.t. this (e.g. AJDT[5]), the accuracy of the displayed information is not fully reliable thus requiring developer's interference (e.g. frequent "refresh" actions and manual inspection). As a consequence, faults can still remain in the code. Some PCD-based testing approaches have been recently proposed [6, 34, 35], however they still require proper evaluation.

Figure 3 shows a F1.7 instance that was identified in iBATIS. The `declare soft` statement in line 5 softens[6] `Exception` instances that may be thrown at JPs selected by the PCD defined in lines 2-3. However, the `after throwing` advice still expects to advise JPs where `Exception` instances may be thrown, hence this advice is never triggered.

```
1 public pointcut afterPropertyAccessPlanSetProperties():
2    execution(public void
3      PropertyAccessPlan.setProperties(Object, Object[]));
4
5 declare soft : Exception :
6      afterPropertyAccessPlanSetProperties();
7
8 after() throwing(Exception e) throws NestedRuntimeException:
9      afterPropertyAccessPlanSetProperties(){
10   throw new NestedRuntimeException(
11         "Error setting properties. Cause: " + e, e);
12 }
```

Fig. 3. Example of a faulty PCD-related implementation scenario.

Fault-leading steps: (1) a PCD `p` is defined; (2) an exception softening is implemented and bound to `p`; (3) an exception-handling advice is defined and bound to `p`, however it relies on an inaccurate list of possibly thrown exceptions.

B. ITD-Related Faults

Figure 4 shows an example of fault classified as F2.6, extracted from the HW system. It consists in a typical mistake made by programmers while aspectising exception handlers. The `HWTransactionManagement` aspect softens exceptions (lines 7-10) that might be thrown by itself (method calls in lines 13, 17 and 21). However, no handler is defined for the softened exceptions, neither within the same aspect nor in some higher-level class in the method call chain.

Fault-leading steps: (1) a PCD `p` is defined; (2) an exception softening is implemented and bound to `p`; (3) an exception-handling advice is expected to handle the softened exception though it is not implemented.

[5]http://www.eclipse.org/ajdt/ - accessed on 28/07/2010

[6]In AspectJ, *softening* an exception means that it is converted into an unchecked, `SoftException` instance.

```
1  public aspect HWTransactionManagement {
2
3   pointcut transactionalMethods(): execution(*
4                              HealthWatcherFacade.*(..)) &&
5                              ! execution(static * *.*(..));
6
7   declare soft: TransactionException :
8      call(void IPersistenceMechanism.beginTransaction())    ||
9      call(void IPersistenceMechanism.rollbackTransaction()) ||
10     call(void IPersistenceMechanism.commitTransaction());
11
12  before(): transactionalMethods() {
13     getPm().beginTransaction();
14  }
15
16  after() returning: transactionalMethods()  {
17     getPm().commitTransaction();
18  }
19
20  after() throwing: transactionalMethods()  {
21     getPm().rollbackTransaction();
22  }
23
24  public IPersistenceMechanism getPm() {
25     return HWPersistence.aspectOf().getPm();
26  }
27 }
```

Fig. 4. Example of a faulty exception softening implementation scenario.

Figure 5 shows another example of fault from Group 2 (type 2.7), now extracted from the MM system. In this case, two different aspects (namely, `SMSAspect` and `PhotoAndMusicAspect`) are advising the same JP (shadowed in grey), however no precedence order is defined. This arbitrary execution order may impact future error recovery actions in case one of the advices presents abnormal behaviour.

```
public privileged aspect SMSAspect {
  ...
  pointcut startApplication(MainUIMidlet middlet):
          execution(public void MainUIMidlet.startApp())
          && this(middlet);

  after(MainUIMidlet middlet): startApplication(middlet) {
    ...
  }
  ...
}

public aspect PhotoAndMusicAspect {
  ...
  pointcut startApp(MainUIMidlet midlet):
          execution( public void MainUIMidlet.startApp() )
          && this(midlet);

  after(MainUIMidlet midlet): startApp(midlet) {
     ...
  }
  ...
}
```

Fig. 5. Example of unintended advice execution order .

Fault-leading steps: (1) Two PCDs p_1 and p_2 are defined to match a common JP; (2) The advices that are bound to p_1 and p_2 are implemented and the affected JPs are possibly verified within each aspect; (3) The affected JPs are not verified from the base-program side, thus they are advised in arbitrary order.

C. Advice-Related Faults

Figure 6 shows an example of fault type 3.3, which accounts for the highest fault count within Group 3 (i.e. advice-related faults). The example was extracted from iBATIS. The `executeQueryWithCallback` method (lines 1-26) is affected by the advice listed in lines 35-37. This advice implements Error Context behaviour and affects `executeQueryWithCallback` at several points (e.g. lines 7-8, 9 and 13-14). The two variables highlighted in grey (i.e. `errorContext` and `executeUpdateErrorContext`) are used inside the affected class and the aspect, respectively, to together store the execution trace of the `executeQueryWithCallback` method. Note that `executeUpdateErrorContext` is initialised with `errorContext`'s value in a previous class-aspect interaction (not listed in the figure). However, an exception is thrown at line 9 and all execution context information stored in the `executeUpdateErrorContext` variable is lost.

```
1  protected List executeQueryWithCallback( parameter list )
2                                  throws SQLException {
3    ErrorContext errorContext = request.getErrorContext();
4    try {
5      validateParameter(parameterObject);
6      Sql sql = getSql();
7      ParameterMap parameterMap =
8                                  sql.getParameterMap(arguments);
9      ResultMap resultMap = sql.getResultMap(arguments);
10     request.setResultMap(arguments);
11     request.setParameterMap(arguments);
12     List resultList = new ArrayList();
13     Object[] parameters =
14          parameterMap.getParameterObjectValues(arguments);
15     ... some other code also affected by the aspect below
16     return resultList;
17   } catch (SQLException e) {
18     errorContext.setCause(e);
19     throw new NestedSQLException(errorContext.toString(),
20                   e.getSQLState(), e.getErrorCode(), e);
21   } catch (Exception e) {
22     errorContext.setCause(e);
23     throw new NestedSQLException(
24                   errorContext.toString(), e);
25   }
26 }
27
28 privileged aspect ErrorContextAspect
29                                  perthis( pointcut name ) {
30   ...
31   ErrorContext executeUpdateErrorContext;
32   ...
33   pointcut callGetResultMap(): pointcut expression  ;
34
35   before(): callGetResultMap(){
36     executeUpdateErrorContext.setMoreInfo( info );
37   }
38   ...
39 }
```

Fig. 6. Example of a faulty advice-related implementation scenario.

Fault-leading steps: (1) context information is shared between and updated by the base code and advices; (2) the joint execution (base program - advice behaviour) is broken in the event of an exception and no error recovery action prevents the partial loss of shared, updated information

D. Base Program-Related Faults

We identified F4.2 as the most representative fault type from Group 4, i.e. base program-related faults. Figure 7 shows a typical scenario for this problem, which was extracted from iBATIS. The `callSqlExecuteUpdate` PCD picks out calls to the `sqlExecuteUpdate` method that occur inside the `executeUpdate` or `executeQueryWithCallback` methods. Therefore, the call to `sqlExecuteUpdate` in the original method (highlighted in grey) is matched by this PCD. However, `sqlExecuteUpdate` was refactored to expose a JP during the aspectisation of the Connection concern. As a result, a call to the `executeUpdatePrepend` replaced the original method call, thus breaking the PCD and hence the Error Context functionality.

Pointcut (ErrorContextAspect **aspect**)

```
public pointcut callSqlExecuteUpdate():
 call(protected int sqlExecuteUpdate(..)) &&
 (withincode(public int GeneralStatement.executeUpdate(..)) ||
  withincode( protected List
              GeneralStatement.executeQueryWithCallback(.. )));
```

Original method (GeneralStatement **class)**

```
public int executeUpdate(parameters) throws SQLException {
   ...
   rows = sqlExecuteUpdate(request,
                conn, sqlString, parameters);
   ...
}
```

Modified method (refactored):

```
public int executeUpdate(parameters) throws SQLException {
   ...
   rows = executeUpdatePrepend(request,
              conn, parameters, sqlString);
   ...
}
```

Fig. 7. Example of a broken pointcut scenario.

Fault-leading steps: (1) a maintenance task (e.g. evolutionary or perfective) is performed in the base code that is matched by a PCD `p`; (2) no revision of `p`'s JP matching is performed.

VII. Study Limitations and Related Work

The refactoring-driven approach adopted for the development of the analysed systems is a possible limitation of this study. However, this has been a common practice for the majority of AO systems developed and investigated so far [20, 21, 36, 37], what may have acted in favour of the AO version (i.e. less introduced faults) as it was being developed from an already-stable architecture. On the other hand, it may also have acted against, as the architecture in the Java version may have restricted the AO solution as found in a case study that refactored a legacy application from Java to AspectJ [38].

Another threat to the validity of this study regards the size and representativeness of the evaluated systems, hence limiting the generalisation of the results. However, they come from heterogeneous domains and are all implemented in AspectJ, which is the most representative language in the state of AOP practice. iBATIS is a widely-used framework. Moreover, despite the size of HW and MM (i.e. smaller applications), they are also heavily based on industry-strength technologies. Therefore, the characteristics of these systems, when contrasted with the state of practice, represent a first step towards the generalisation of the achieved results.

In regarding to the testing strategy applied to the systems, in particular to iBATIS and HW, it consisted in regression testing due to the fact that during the development, each AO release should include exactly the same set of functionalities of its OO counterpart. Therefore, we assumed there would not be necessary extra tests once we achieved a good coverage for the OO versions.

As introduced in Section V, this paper extends our previous research [19] that investigated the fault-proneness of the major AOP mechanisms. The main difference between the two investigations is: when considering a coarse-grained fault classification (i.e. based on the major mechanisms), we came up with similar fault counts for the three groups of faults [19]. However, in this paper we considered the fine-grained classification presented in Section III, thus looking at particular fault types that stood out w.r.t the fault frequency.

So far, we could identify a single other quantitative study of faulty behaviour in AO programs [14]. The authors evaluated a set of AO systems – which includes some releases of HW and MM – and quantified exception-dependent paths that are likely to be problematic (e.g. resulting either in uncaught or swallowed exceptions). In their study, a single implementation fault may lead to several problematic exception flows. In our study, on the other hand, we quantified the root causes of problems (e.g. a missing handler) instead of their consequences (e.g. several uncaught exceptions).

When fault characterisation is concerned, we can identify a few attempts in the context of AO programs. For instance, Alexander et al. [7] were the first authors who defined a candidate fault taxonomy for AO programs. They used simple examples to illustrate how the fault types included in their taxonomy can occur in AO programs. Similar work has been done by Zhang and Zhao [13], also based on small examples. Finally, Coelho et al. [14] defined a set of bug patterns specific for exception handlers implemented in AspectJ. Their discussion was supported by some examples extracted from the HW and MM systems.

Most of the reported studies on the characterisation of software faults are based on the C language (and previous technologies), and are therefore built upon the procedural development approach (i.e. it does not take into account AOP or even OO programming). For instance, Endress [2] was one of the first authors to classify software faults occurring in real software projects. The fault classification is based on primary activities of designing and implementing algorithms in an operating system. Ostrand and Weyuker [3], on the other hand, collected and categorised software fault data from an interactive, special-purpose editor system. Basili and Perricone [4] analysed a medium scale system to provide a classification of fault data.

VIII. Conclusions and Future Work

This paper presented the results of a study that quantified and categorised faults in AO programs according to a fine-grained fault taxonomy that accounts for the main AOP mechanisms. The faults were extracted from several releases of three AO systems and then classified according to the varied fault types that are described in the taxonomy. The fault taxonomy itself is also a contribution of this work. I was introduced in our previous research [6] and refined herein. The refinement consisted of a more detailed description of each fault type together with the description of generic examples.

The results show that a subset of fault types stood out within each coarse-grained fault category (e.g. pointcut- or advice-related faults). For instance, faults that are caused by incorrect join point matchings based on exception throwing patterns were the most recurrent type of faults inside the pointcut-related category. Another example regards pointcuts that are broken due to the evolution of the base program, therefore classified as base program-related faults. For each fault category, we presented real examples extracted from

the analysed systems. They are representatives of the most recurring fault types and illustrate the steps that led the developer to introduce the faults into the code.

The results of this study can support the definition of testing strategies that enforce the testing of parts of the code that are involved in particular fault-prone scenarios, described herein. We intend to tackle this issue in our upcoming research. In a more general view, the results can be useful for programming language designers, who can further investigate how to improve AOP features.

ACKNOWLEDGEMENTS

We would like to thank Andrew Camilleri from Lancaster University (UK) and Eduardo Figueiredo from UFMG (Brazil) for their help while analysing the MobileMedia system with the CIF framework. We also thank Nélio Cacho from UFRN (Brazil) for his support in the HealthWatcher system analysis.

The authors received full of partial funding from the following agencies and projects: *Fabiano Ferrari*: FAPESP (grant 05/55403-6), CAPES (grant 0653/07-1) and EC Grant AOSD-Europe (IST-2-004349); *Rachel Burrows*: UK EPSRC grant; *Otávio Lemos*: FAPESP (grant 2008/10300-3); *Alessandro Garcia*: FAPERJ (distinguished scientist grant E-26/102.211/2009), CNPq (productivity grant 305526/2009-0 and Universal Project grant number 483882/2009-7), and PUC-Rio (productivity grant) ; *José Maldonado*: EC Grant QualiPSo (IST-FP6-IP-034763), FAPESP, CAPES and CNPq.

REFERENCES

[1] J. Offutt, R. Alexander, Y. Wu, Q. Xiao, and C. Hutchinson, "A fault model for subtype inheritance and polymorphism," in *ISSRE'01*. IEEE Computer Society Press, 2001, pp. 84–93.

[2] A. Endress, "An analysis of errors and their causes in systems programs," *IEEE Transactions on Software Engineering*, vol. 2, pp. 140–149, 1978.

[3] T. J. Ostrand and E. J. Weyuker, "Collecting and categorizing software error data in an industrial environment," *Journal of Systems and Software*, vol. 4, no. 4, pp. 289–300, 1984.

[4] V. R. Basili and B. T. Perricone, "Software errors and complexity: An empirical investigation," *Communications of the ACM*, vol. 27, no. 1, pp. 42–52, 1984.

[5] G. Kiczales et al., "Aspect-oriented programming," in *ECOOP'97*. Springer, 1997, pp. 220–242 (LNCS 1241).

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

[7] R. T. Alexander, J. M. Bieman, and A. A. Andrews, "Towards the systematic testing of aspect-oriented programs," Dept. of Comp, Science, Colorado State Univ., Report CS-04-105, 2004.

[8] J. S. Bækken, "A fault model for pointcuts and advice in AspectJ programs," Master's thesis, School of Electrical Engineering and Computer Science, Washington State Univ., USA, 2006.

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

[10] M. Eaddy, A. Aho, W. Hu, P. McDonald, , and J. Burger, "Debugging aspect-enabled programs," in *SC'07*, 2007.

[11] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero, and C. V. Lopes, "Testing aspect-oriented programming pointcut descriptors," in *WTAOP'06*. ACM Press, 2006, pp. 33–38.

[12] 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, The Netherlands, Report SEN-R0507, 2005.

[13] S. Zhang and J. Zhao, "On identifying bug patterns in aspect-oriented programs," in *COMPSAC'07*, 2007, pp. 431–438.

[14] R. Coelho et al., "Assessing the impact of aspects on exception flows: An exploratory study," in *ECOOP'08*. Springer, 2008, pp. 207–234 (LNCS v.5142).

[15] "Demoiselle framework," Online, http://www.frameworkdemoiselle.gov.br/.

[16] F. Munoz, B. Baudry, R. Delamare, and Y. L. Traon, "Inquiring the usage of aspect-oriented programming: An empirical study," in *ICSM'09*. IEEE Computer Society, 2009, pp. 137–146.

[17] R. Burrows, F. Ferrari, A. Garcia, and F. Taïani, "An empirical evaluation of coupling metrics on aspect-oriented programs," in *ICSE WETSoM Workshop*, 2010, pp. 53–58.

[18] N. Cacho, F. Dantas, A. Garcia, and F. Castor Filho, "Exception flows made explicit: An exploratory study," in *SBES'09*. IEEE Computer Society, 2009, pp. 43–53.

[19] F. C. Ferrari et al., "An exploratory study of fault-proneness in evolving aspect-oriented programs," in *ICSE'10*. ACM Press, 2010, pp. 51–58.

[20] E. Figueiredo et al., "Evolving software product lines with aspects: An empirical study on design stability," in *ICSE'08*. ACM Press, 2008, pp. 261–270.

[21] P. Greenwood et al., "On the impact of aspectual decompositions on design stability: An empirical study," in *ECOOP'07*. Springer, 2007, pp. 176–200 (LNCS 4609).

[22] E. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.

[23] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An overview of AspectJ," in *ECOOP'01*. Springer-Verlag, 2001, pp. 327–353 (LNCS v.2072).

[24] N. McEachen and R. T. Alexander, "Distributing classes with woven concerns: An exploration of potential fault scenarios," in *AOSD'05*. ACM Press, 2005, pp. 192–200.

[25] F. C. Ferrari and J. C. Maldonado, "A systematic review on aspect-oriented software testing," in *WASP'06*. Brazilian Computer Society, 2006, pp. 101–110, (in Portuguese).

[26] F. C. Ferrari, E. N. Höhn, and J. C. Maldonado, "Testing aspect-oriented software: Evolution and collaboration through the years," in *LAWASP'09*. Brazilian Computer Society, 2009, pp. 24–30.

[27] 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.

[28] "iBATIS data mapper," Online, http://ibatis.apache.org/.

[29] A. Camilleri, G. Coulson, and L. Blair, "CIF: A framework for managing integrity in aspect-oriented composition," in *TOOLS'09*. Springer-Verlag, 2009, pp. 18–26 (LNBIP v.33).

[30] F. Castor Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. M. F. Rubira, "Exceptions and aspects: The devil is in the details," in *FSE'06*. ACM Press, 2006, pp. 152–162.

[31] F. Castor Filho, A. Garcia, and C. M. F. Rubira, "Extracting error handling to aspects: A cookbook," in *ICSM'07*. IEEE Computer Society, 2007, pp. 134–143.

[32] M. Stoerzer and J. Graf, "Using pointcut delta analysis to support evolution of aspect-oriented software," in *ICSM'05*. IEEE Computer Society, 2005, pp. 653–656.

[33] "The AspectJ programming guide," Online, http://www.eclipse.org/aspectj/doc/released/progguide/index.html - (27/04/2010).

[34] O. A. L. Lemos and P. C. Masiero, "Using structural testing to identify unintended join points selected by pointcuts in aspect-oriented programs," in *SEW'08*. IEEE Computer Society, 2008.

[35] 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.

[36] M. Marin, L. Moonen, and A. van Deursen, "An integrated crosscutting concern migration strategy and its application to JHotDraw," in *SCAM'07*. IEEE Computer Society, 2007, pp. 101–110.

[37] M. Mortensen, S. Ghosh, and J. M. Bieman, "Testing during refactoring: Adding aspects to legacy systems," in *ISSRE'06*. IEEE Computer Society, 2006, pp. 221–230.

[38] C. Kastner, S. Apel, and D. Batory, "A case study implementing features using AspectJ," in *Proceedings of the 11th International Software Product Line Conference (SPLC)*. IEEE Computer Society, 2007, pp. 223–232.

Appendix

C

Paper: Mutation Testing for Aspect-Oriented Programs

This appendix presents the full contents of a paper published in the Proceedings of the 1^{st} International Conference on Software Testing, Verification and Validation (ICST'08). An overview of this work was provided in Chapter 4 of this dissertation. A copyright notice in regard to it is next shown.

IEEE COPYRIGHT NOTICE[1]**:** "*Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.*"

[1] `http://www.ieee.org/portal/cms_docs/pubs/transactions/auinfo03.pdf` - last accessed on 17/08/2010.

Mutation Testing for Aspect-Oriented Programs

Fabiano Cutigi Ferrari*, José Carlos Maldonado
University of São Paulo - São Carlos/SP, Brazil
Computing Systems Department
{*ferrari,jcmaldon*}*@icmc.usp.br*

Awais Rashid
Lancaster University - Lancaster, UK
Computing Department
marash@comp.lancs.ac.uk

Abstract

Mutation testing has been shown to be one of the strongest testing criteria for the evaluation of both programs and test suites. Comprehensive sets of mutants require strong test sets to achieve acceptable testing coverage. Moreover, mutation operators are valuable for the evaluation of other testing approaches. Although its importance has been highlighted for Aspect-Oriented (AO) programs, there is still a need for a suitable set of mutation operators for AO languages. The quality of the mutation testing itself relies on the quality of such operators. This paper presents the design of a set of mutation operators for AspectJ-based programs. These operators model instances of fault types identified in an extensive survey. The fault types and respective operators are grouped according to the related language features. We also discuss the generalisation of the fault types to AO approaches other than AspectJ and the coverage that may be achieved with the application of the proposed operators. In addition, a cost analysis based on two case studies involving real-world applications has provided us feedback on the most expensive operators, which will support the definition of further testing strategies.

1. Introduction

Aspect-Oriented Programming (AOP) [19] has introduced new concepts and technologies into the software life cycle. Despite the claimed benefits AOP brings, it poses new challenges for software quality assurance [5], motivating the proposal of several approaches for aspect-oriented (AO) software testing in the last years [5, 21, 32, 33, 35].

Although mutation testing has been empirically shown to be one of the strongest testing criteria [24, 31], it has not been deeply explored in the context of AO software. Some incipient work has been proposed [6, 20, 26]. However, no comprehensive set of mutation operators for AO implementations has been proposed to date.

*He is currently a visiting student at Lancaster University.

Mutation testing is a valuable resource for the evaluation of software artefacts at different levels of abstraction (e.g. specification and source-code) and also test suites. In addition, the faults modelled by mutation operators may be useful in order to evaluate how sensitive general testing approaches are in order to reveal these faults. In this sense, identifying a set of fault types which cover most of the AO features and modelling these fault types within mutation operators represent a step forward in improving the quality of AO software and related testing approaches. In this context, this paper makes the following contributions:

- Identification of a comprehensive set of AO fault types based on previous works on AO testing;
- Design of a set of mutation operators for the AspectJ language [3], based on the identified fault types, also applicable to other similar languages;
- Analysis of identified fault types for generalisation to AO implementations other than AspectJ; and
- Application of the proposed operators and a cost estimate comprising two real-world case studies, demonstrating the potential cost of the operators.

Considering that usually research comprising AO faults types and testing approaches is somehow related to AspectJ features, we have analysed to what extent candidate fault models may be generalised to well disseminated or even general AO approaches. We have found that most of characterised fault types may also occur in other AO implementations (e.g. JBoss AOP [2] and CaesarJ [25]). We have also found that the proposed mutation operators cover a wide range of these fault types. In addition, our cost analysis shows that the most expensive operators are related to the quantification features of AspectJ. Such information is essential when defining testing strategies.

This paper is organised as follows: Section 2 presents a brief overview of AOP, AspectJ and mutation testing. Section 3 summarises the AO fault types we have identified and the proposed set of mutation operators. In Section 4, we discuss the generalisation of AO fault types beyond AspectJ and issues related to the proposed operators. Section 5 describes a cost analysis for application of the operators based

on two case studies. Related work is discussed in Section 6. Finally, Section 7 presents our conclusion and future work.

2. Background

2.1. Mutation testing

Mutation testing [13] is a fault-based testing criterion which relies on the *Competent Programmer* and the *Coupling Effect* hypotheses. These hypotheses state that a program under test contains only small syntactic faults and that complex faults result from the combination of them. Fixing the small faults will probably solve the complex ones.

Given an original program P, the criterion requires the creation of a set M of *mutants*, consisting of slightly modified versions of P. *Mutation operators* encapsulate the modification rules applied to P. Then, for each mutant $m, (m \in M)$, the tester runs the test suite T originally designed for P. If $m(t) \neq P(t),\ (t \in T)$, this mutant is considered *killed*. If not, the tester should improve 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.

Mutation testing has been shown to be strong amongst several other testing adequacy criteria, such as data flow based criteria [24, 31]. However, the quality of the mutation testing relies on the quality of the mutation operators, which must reflect realistic fault types. Considering this, in Section 3 we present of a set of AspectJ mutation operators designed according to several identified AO fault models. The next section briefly introduces AOP and AspectJ.

2.2. AOP and AspectJ

Aspect-Oriented Programming (AOP) [19] has arisen as a possible solution to improve software modularity. Its adoption is motivated by the fact that traditional software development approaches do not satisfactorily cope with crosscutting concerns; these are typically spread across or tangled with other concerns or even among themselves.

AOP provides means for modularising crosscutting concerns within new conceptual[1] software entities called *aspects*. The aspects are further *woven* into the base application supported by a quantification mechanism. This mechanism allows a simple piece of crosscutting behaviour to be executed in several well-defined points during software execution, called *join points*. Examples of join points are a method call, a method execution and a field access.

In AspectJ [3], which is currently the most consolidated Java-based AOP language, a *pointcut* expression selects a set of join points. It is generally formed by patterns (e.g. method signatures) and predicates. Pointcuts are bound to *advices*, which consist of portions of Java code implementing crosscutting behaviour. AspectJ also allows static modifications of class structure and hierarchy through inter-type declarations (ITDs). For a complete list of AspectJ features, the reader may refer to the AspectJ project Web site [3].

The concepts and elements introduced by AOP technologies result in new potential fault sources [5], hence posing new challenges for testing activities. For example, faults can arise from crosscutting behaviour or from the interactions between aspects and the base application. Regarding this, the next section introduces a set of mutation operators for AspectJ, based on a set of characterised AO fault types.

[1] *Conceptual* since symmetric AOP approaches ideally do not require new software entities to implement crosscutting behaviour.

3. Mutation testing for AspectJ programs

This section initially presents a set of AO-specific fault types. Next, we introduce a set of mutation operators for the AspectJ language based on these fault types.

3.1. AO Fault Types

Tables 1–4 present a set of AO fault types identified during our extensive survey on AO testing. Our analysis has drawn on AO candidate fault models [5, 7, 9, 15, 30], fault classifications [20] and bug patterns [34]. The faults are distributed in four groups related to: (F1) pointcut expressions; (F2) ITDs and other declarations; (F3) advice definition and implementation; and (F4) the base program. We refer to a specific fault type by its group plus its number (e.g. F1.4). We have added three new fault types which have not been included by previous works (italicised in tables 2 and 3).

Table 1. Pointcut related faults (F1)

#	Description
1	Selection of a superset of join points
2	Selection of a subset of join points
3	Selection of a wrong set of join points, including intended and unintended ones
4	Selection of a wrong set of join points, including only unintended ones
5	Incorrect use of primitive pointcut designators
6	Incorrect pointcut composition rules
7	Incorrect matching based on exception throwing patterns
8	Incorrect matching based on dynamic values and events

We state this set of fault types is as comprehensive as possible since it relies on the expertise of researchers and practitioners and also on our own experience on AOP. However, to date, it is not possible to find historical data regarding AO fault types from real projects. Further feedback from such projects may elicit new fault types to be included in the presented set. In addition, analysis of the inclusion relation amongst fault types and experimental studies may help us to minimise the set.

Table 2. ITD related faults (F2)

#	Description
1	Improper method introduction, resulting in inconsistencies in method overriding
2	Introduction of a method into an incorrect class
3	Incorrect changes in class hierarchy
4	Incorrect method introduction, resulting in unexpected method overriding
5	Omitted declared parent interface or introduced interface which breaks object identity
6	Incorrect changes in exception dependent control flow
7	Incorrect or omitted aspect precedence declaration
8	*Incorrect aspect instantiation rules and deployment*
9	*Incorrect policy enforcement rules supported by warning and error declarations*

Table 3. Advice related faults (F3)

#	Description
1	Incorrect advice type specification
2	Incorrect control or data flow due to execution of the original join point
3	Incorrect advice logic, violating invariants and failing to establish expected postconditions
4	Infinite loops resulting from interactions among advices
5	*Incorrect access to join point static information*
6	Advice bound to incorrect pointcut

Table 4. Base program related faults (F4)

#	Description
1	Base program does not offer required join points
2	Software evolution causes pointcut to break
3	Other problems related to maintenance of the base program (e.g. inconsistencies and duplicated crosscutting code)

Except from Bækken [7], none of the authors present details of possible instances of each fault type. We call *fault instance* a specific occurrence of a fault type. For example, an incorrect use of a wildcard in a pointcut expression is an instance of *"Selection of a superset of join points"* (F1.1). Most of such fault types focus on specific AspectJ structures and characteristics. Possible generalisations to other AO implementations are discussed in Section 4.1.

3.2. Mutation operators for AspectJ

This section introduces a set of mutation operators for AspectJ programs based on the set of AO fault types presented in Section 3.1. The operators model possible fault instances according to syntactic constructions allowed in AspectJ and similar languages. They are organised into three groups, summarised in tables 5, 6 and 7. Due to space limitations, for each group we only present a short description and examples of operator application.

The application of some operators is straightforward from their descriptions. For example, the application of an operator which simply removes `declare precedence` clauses is straight (or trivial). On the other hand, an operator which removes `proceed` statement calls must consider how such statement appears inside an advice (e.g. in a variable definition or in a return expression). The examples we next show consist of non-trivial operator applications. In the examples, the ◯ and Δ symbols indicate the original and the mutated code, respectively. The > symbol indicates a mutated line and each occurrence usually represents a different mutant (some operators require more than one mutated line). Notice that in the examples we use generic element names like a pointcut "`p`", a class "`C`", a method "`m`", an attribute type "`A`" and a return type "`R`".

3.2.1. Group 1 - Operators for pointcut expressions. Group 1 contains 15 mutation operators which model faults related to pointcut expressions. Such faults usually result in incorrect join point matchings or undue execution contexts. This group, shown in Table 5, is divided into four categories, according to the results obtained from the application of the respective operators:

(i) Pointcut Weakening operators: This group is compounded by the PWSR, PWIW and PWAR operators (see Table 5). The resulting mutants possibly *increase* the number of selected join points if compared to the original pointcut expression. Following is an example of how the PWIW operator inserts the "⋆" and "+" wildcards into the original expression `call(public R C.m(A))`.

```
◯ | pointcut p(): call(public R C.m(A));

Δ | > pointcut p(): call(public * C.m(A));
  | > pointcut p(): call(public R *.m(A));
  | > pointcut p(): call(public R C.*(A));
  | > pointcut p(): call(* C.m(A));
  | > pointcut p(): call(public R C.m(*));
  | > pointcut p(): call(public R* C.m(A));
  | > pointcut p(): call(public R C*.m(A));
  | > pointcut p(): call(public R C.m*(A));
  | > pointcut p(): call(public R C.m(A*));
  | > pointcut p(): call(public *R C.m(A));
  | > ... // and so on
```

(ii) Pointcut Strengthening operators: Operators of this group perform modifications to *decrease* the number of selected join points if compared to the original pointcut expression. The PSSR, PSWR and PSDR operators comprise this group (see Table 5). The following example shows how the PSSR operator replaces types present in the expression `call(* C.m(TypeA))` with their immediate subtypes.

```
◯ | pointcut p(): call(* C.m(A));

Δ | > pointcut p(): call(* SubtypeOfC.m(A));
  | > pointcut p(): call(* C.m(SubtypeOfA));
```

(iii) Pointcut Weakening or Strengthening operators: These operators produce mutants that may either *increase* or *decrease* the number of selected join points. They are the POPL, POAC and POEC operators (see Table 5). Next is an example where the POPL operator replaces items from the list `(TypeA,..,TypeB)` with the ".." wildcard.

Table 5. Mutation operators for pointcut expressions (Group 1).

Operator	Description/Consequences
PWSR	Pointcut weakening by replacing a type with its immediate supertype in pointcut expressions
PWIW	Pointcut weakening by inserting wildcards into pointcut expressions
PWAR	Pointcut weakening by removing annotation tags from type, field, method and constructor patterns
PSSR	Pointcut strengthening by replacing a type with its immediate subtype in pointcut expressions
PSWR	Pointcut strengthening by removing wildcards from pointcut expressions
PSDR	Pointcut strengthening by removing `declare @` statements, used to insert annotations into base code elements
POPL	Pointcut weakening or strengthening by changing parameter lists of primitive pointcut designators
POAC	Pointcut weakening or strengthening by changing `after [retuning|throwing]` advice clauses
POEC	Pointcut weakening or strengthening by changing exception throwing clauses
PCTT	Pointcut changing by replacing a `this` pointcut designator with a `target` one and vice versa
PCCE	Context changing by switching `call/execution/initialization/preinitialization` pointcuts designators
PCGS	Pointcut changing by replacing a `get` pointcut designator with a `set` one and vice versa
PCCR	Pointcut changing by replacing individual parts of a pointcut composition
PCLO	Pointcut changing by changing logical operators present in type and pointcut compositions
PCCC	Pointcut changing by replacing a `cflow` pointcut designator with a `cflowbelow` one and vice versa

Table 6. Mutation operators for AspectJ declarations (Group 2).

Operator	Description/Consequences
DAPC	Aspect precedence changing by alternating the order of aspects involved in `declare precedence` statements
DAPO	Arbitrary aspect precedence by removing `declare precedence` statements
DSSR	Unintended exception handling by removing `declare soft` statements
DEWC	Unintended control flow execution by changing `declare error/warning` statements
DAIC	Unintended aspect instantiation by changing `perthis/pertarget/percflow/percflowbelow` deployment clauses

Table 7. Mutation operators for advice definitions and implementations (Group 3).

Operator	Description/Consequences
ABAR	Advice kind changing by replacing a `before` clause with an `after [retuning|throwing]` one and vice versa
APSR	Advice logic changing by removing invocations to `proceed` statement
APER	Advice logic changing by removing guard conditions which surround `proceed` statements
AJSC	Static information source changing by replacing a `thisJoinPointStaticPart` reference with a `thisEnclosingJoinPointStaticPart` one and vice versa
ABHA	Behaviour hindering by removing implemented advices
ABPR	Changing pointcut-advice binding by replacing pointcuts which are bound to advices

```
○ | pointcut p(): call(* C.m(TypeA,..,TypeB));

Δ | > pointcut p(): call(* C.m(TypeA,TypeB));
  | > pointcut p(): call(* C.m(..,..,TypeB));
  | > pointcut p(): call(* C.m(TypeA,..,..));
```

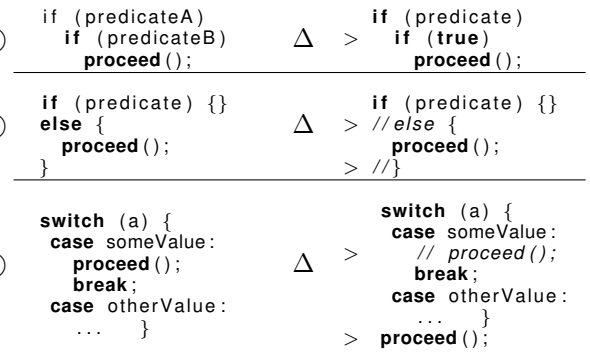
(iv) Pointcut Changing operators: Operators from this group perform a variety of changes in pointcut expression, as shown in Table 5. As a result, we have: (i) partial or complete changes in the number of selected join points with the PCTT, PCGS, PCCR and PCLO operators; (ii) changes in execution context with the PCCE operator; and (iii) changes in the number of advice executions with the PCCC operator. The following example shows how the PCLO operator changes logical operators in a pointcut composition.

```
○ | pointcut composite(): pA() || pB();

Δ | > pointcut composite(): pA() && pB();
  | > pointcut composite(): !pA() || pB();
  | > pointcut composite(): pA() || !pB();
```

3.2.2. Group 2 - Operators for general declarations. Group 2, summarised in Table 6, contains five operators that model faults related to general AspectJ declarations. Related faults lead to unintended control flow executions and object/aspect state. For example, the DAPC operator modifies aspect precedence declarations, varying among all possible precedences for involved aspects. The DAPO operator is applied to omit `declare precedence` statements. Other declarations targeted by operators of Group 2 are `declare error`, `declare warning`, `declare soft` and aspect instantiation rules. Next is an example of how the DEWC operator suppresses the `declare error` statement and replaces it with a `declare warning` one.

```
○ | declare error: p(): "a message...";

Δ | > //declare error: p(): "a message...";
  | > declare warning: p(): "a message...";
```

3.2.3. Group 3 - Operators for advice definitions and implementations. Table 7 summarises a set of six operators related to advice definition and implementation. Faults modelled by operators from this group comprise incorrect advice kind (ABAR operator), incorrect advice logic (APSR, APER and AJSC operators) and incorrect advice execution (ABHA and ABPR operators). In the following example, the APER operator removes the guard conditions that surround the `proceed` statement in three different situations: when it is guarded by an `if`, an `else` or a `switch-case` condition.



4. Discussion

This section first addresses issues related to generalisation of AO fault types, i.e. if they are either specific for AspectJ programs or may be extended to other AO implementations. Following this, we discuss some limitations and implications for the proposed mutation operators.

4.1. May AO fault types be generalised?

AspectJ, JBoss AOP [2] and Spring AOP [18] are AO technologies which have a significant user base. CaesarJ [25] is another recent language which provides a number of features to enable AOP. Our discussion, summarised in Table 8, is focused on these technologies, but not restricted to them. However, a full and refined analysis regarding all AO implementations is out of the scope of this paper.

Group 1 - Pointcut related faults: Many pointcut faults are directly related to the quantification mechanism, which must somehow be present in all AO implementations. Specifically, these faults are related to pointcut definition and composition rules (F1.1, F1.2, F1.3, F1.4 and F1.6 in our grouping).

Faults arising from incorrect use of primitive pointcut designators (F1.5) (e.g. a method call or execution) occur in most, even all AO implementations, considering that they usually provide underlying join point models. These models enable the selection of desired join points usually through a set primitive pointcut designators, and misuses of these designators result in software faults.

Faults related to incorrect matching based on exception throwing patterns (F1.7) depend on the availability of specific supporting mechanisms. For example, while AspectJ provides the `handler` pointcut designator and the `after throwing` advice type, JBoss AOP may handle exception throwing scenarios through its interceptor advice model.

Finally, all AO implementations that perform some kind of runtime type checking (e.g. for binding or advice execution purposes) are prone to faults related to matching based on dynamic circumstances (F1.8). These faults may also occur due to runtime checking of control flow context, as implemented in AspectJ, JBoss AOP and Spring AOP.

Group 2 - ITD related faults: Many AO implementations allow modifications of both structure and hierarchy of the base application. A modification may consist of a member introduction (method or attribute), an inheritance change or new requested interface.

Fault instances of types F2.1 to F2.5 rely on the availability of mechanisms that support these modifications. For example, AspectJ allows static changes in class structures by the insertion of members and by the redefinition of class hierarchy and required interfaces. JBoss AOP and Spring AOP only support the introduction of new required interfaces, which can be realised through specific annotations and XML tags. In CaesarJ, support for member introduction is provided by *virtual classes*, *collaborating classes* and *mixins composition*. All these specific features are potential fault sources like AspectJ ITDs.

AspectJ provides the `declare soft` statement which converts a checked exception into a runtime one. Fault type F2.6 comprises this kind of scenario. We have not identified a similar mechanism in other AO implementations. However, the wrapping approach used to control join point executions, which is present in several of them, may be used to catch and rethrow checked and runtime exceptions.

Incorrect execution order of aspects may be present in most AO artefacts. Thus, faults related to aspect precedence order (F2.7) occur in any AO implementation. AspectJ and CaesarJ, for example, allow aspect precedence definition through `declare precedence` clauses. Other implementations provide specific clauses for the same purpose.

A number of options are available for aspect deployment in AO implementations, which may result in faults related to aspect instantiation (F2.8). For example, in AspectJ and JBoss AOP, aspects may be deployed as singleton, per target or this object and so on. CaesarJ, on the other hand, requires explicit aspect instantiation during execution flow.

Fault type F2.9 depends exclusively on error and warning declaration mechanisms, as provided by AspectJ and JBoss AOP. If incorrectly used (e.g. the compiler signalling a warning instead of an error message), they may result in faults related to policy enforcement rules.

Table 8. Relationship between AO implementations and fault types.

	Fault Types																								
	F1								F2									F3						F4		
AO Implem.	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	1	2	3
AspectJ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Spring AOP	✓	✓	✓	✓	✓	✓	✓	✓			✓		✓		✓	✓		✓	✓	✓	✓	✓	✓	✓	✓	✓
JBoss AOP	✓	✓	✓	✓	✓	✓	✓	✓			✓		✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
CaesarJ	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓

Group 3 - Advice related faults: After a pointcut has selected a set of join points, one or more advices are set to run usually before, after or around them, supported by some kind of weaving mechanism [19]. Thus, faults related to advice type (F3.1), which defines the moment when a crosscutting behaviour should run, may occur in any AO supporting technology.

Around advice usually wraps the execution of the original join point, which may or may not be activated. This feature may lead to incorrect control or data flow due to the execution of unintended behaviour (F3.2). AspectJ, CaesarJ and Spring AOP provide the `proceed` statement to enable the execution of the original join point. In JBoss AOP, on the other hand, all advice is around (called advice methods), which is used to simulate before and after advices.

Regarding faults related to incorrect logic implemented in advices (F3.3), we might consider these faults as non AO-specific, since they are independent of paradigm and technology. However, after weaving, they may lead to violations of state invariants and failures to accomplish postconditions which were not present before weaving.

Faults related to infinite loops resulting from interactions among advices (F3.4) may rely on mechanisms which enable the crosscutting of advice execution. AspectJ and CaesarJ provide the `adviceexecution` pointcut designator. Other AO implementations support the definition of crosscutting behaviour as advice methods. Therefore, they are also prone to similar problems, given the possibility of cyclic interaction among advice methods.

Faults related to incorrect access to static information of join points (F3.5) depend on the availability of specific infrastructure. For example, some Java-based AO implementations, like AspectJ, Spring AOP and CaesarJ, have specific reflection APIs to provide access to static join point information during program execution.

Finally, we consider that all AO implementations are prone to faults related to incorrect binding between pointcut and advice (F3.6). Even if both pointcut definition and advice implementation are correct, the rules that define the binding among these elements may be incorrectly defined.

Group 4 - Base program related faults: Fault types F4.1 and F4.2 are related to software reuse and maintenance, respectively. These fault types arise due to reliance on *naming conventions* and *pointcut fragility* [29]. In AspectJ-like approaches, for example, pointcuts explicitly select join points by matching elements in the base application through their names (e.g. a class or a method name). This high coupling results in pointcut fragility and a simple method renaming may cause pointcuts to break. These fault types may be generalised to AO implementations for paradigms other than OO (e.g. procedural and functional languages [11, 16]), which make use of naming conventions to identify join points. Similar faults are present even in query languages[2] of general composition mechanisms as the one proposed by Harrison et al. [17].

The remaining fault type is related to duplicated code after refactoring activities and inconsistencies between aspects and base program (F4.3). These faults may occur in products resulting from general AO implementations.

4.2. Mutation operator issues

Differently from traditional mutation, an ordinary mutation of a pointcut expression may directly affect many points in the woven application. In addition, a mutation in an advice may imply similar but indirect effect, since this advice may run many times during software execution. However, the underlying concepts of the mutant analysis are the same, i.e. one must observe the behaviour of the mutated code against the provided test cases and decide if a mutant is dead or alive. Regarding equivalent mutants, a mutant pointcut expression may be considered equivalent to the original one when both match the same set of join points [6]. In general, a mutant must be set as equivalent when the observed behaviour is the same for all executions.

Three conditions must be considered for the design of effective mutation operators: *reachability*, *necessity* and *sufficiency* [14]. Given an original product P and a mutant M of P which contains a fault f inserted by a mutation operator: (i) f must be reached and executed (reachability); (ii) the state of M must be infected after the execution of f (necessity); and (iii) the difference between the states of P and M after the execution of the mutated portion of code must propagate to the end of P and M (sufficiency).

Bækken [7] defines the necessity condition for a faulty pointcut as the difference between the sets of the intended

[2] A query language is used in AOP implementations to formulate expressions for identifying sets of correspondences in the base application (i.e. join points) in which specific behaviour will execute [17].

and the actually captured join points. The activation of an advice at an unintended join point represents the sufficiency condition. For advice faults, the execution of a fault which implies differences in subsequent control or data flow (and propagates to the end of the execution) represents the necessity and sufficiency conditions. Finally, for other fault types related to general declarations, the infection may be seen as a side-effect of faulty elements. For example, an incorrect aspect precedence may lead to incorrect order of advice execution, with in turn may lead to observable failures at the end of the execution. Some issues involved in the design and application of the operators are next discussed.

Non-modelled fault types: Table 9 shows the relation between AO fault types and the proposed mutation operators. The table only includes direct effects of operators in relation to the target elements and resulting mutants. We can observe that some fault types of groups 2 and 4 have not been modelled. Regarding the former, ITD mutations would probably result in non-compilable mutants since the expected introduced member would be invoked or accessed in some part of the woven code. Regarding fault types of Group 4, the base program is commonly implemented using OO or procedural approaches. Therefore, traditional mutation operators for unit [4], interface [12] and class [22] levels might be employed on the base program. These operators may also be employed to improve testing quality of advice logic. In addition, considering some AspectJ-specific statements and object references (e.g. `proceed` and `thisJoinPointStaticPart`) we proposed the APSR, AJSC and APER operators.

Finally, we did not propose any mutation operator to switch elements in parameter lists in a similar fashion to Interface Mutation operators [12]. These operators may be adapted to model faults related to incorrect order of parameters identified by Bækken [7].

Inconsistency of object and aspect states: We can consider that all proposed operators influence aspect and object states, from the point of view of the necessity property [14]. For example, operators that change the selected join points may indirectly lead to incorrect or inconsistent states, since extra behaviour will be executed where it should not or some behaviour will not be executed where it should. However, we consider that operators like DAIC, ABAR, APSR and APER are more directly related to incorrect state issues.

Exception throwing problems: Some operators may produce mutants that always throw exceptions when the mutated code runs. One could argue that these kinds of mutants are not representative since they are considered as trivial ones [27]. However, the examples presented in Figures 1-a and 1-b show a case where mutants generated by the PCLO may either throw an

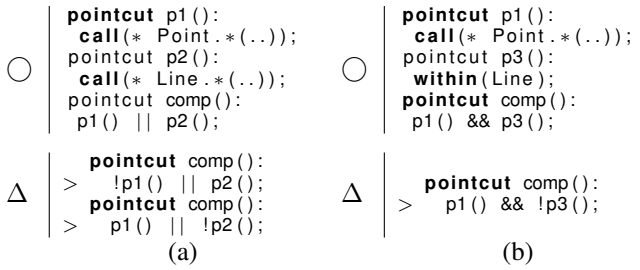


Figure 1. Trivial and non-trivial mutants.

`ExceptionInInitializerError` exception for any execution (1-a) or generate significant mutants (1-b), i.e. which do not throw such exception when they run.

Non-compilable mutants: According to the *Competent Programmer Hypothesis* [13], a program that is under test is correct or almost correct. Thus, probably each modification performed on this program would insert a single fault in it. Besides, such modification should consist of a semantic fault and not a syntactic one, since syntactic faults are revealed at compile time. Some operators proposed in this work are prone to generating non-compilable mutants. In particular, this is the case for the POAC, DSSR, DEWC, ABAR and ABPR operators, which mutate structures that may involve context exposure, exception softening and error and warning declarations.

Approaches to dealing with non-compilable mutants vary among: (i) generating all possible mutants and further marking non-compilable ones as anomalous or stillborn [27]; (ii) generating only compilable mutants (conservative approach); or (iii) generating mutants after a fine-grained analysis to detect compilation problems. Moreover, this issue may be tackled during the automation phase, as done in the *Proteum/IM* [12] and $\mu Java$ [23] tools for C and Java programs mutation testing, respectively.

Mutation applied to anonymous pointcuts: AspectJ allows the definition of *anonymous pointcuts*, which are directly bound to advices. However, they cannot be reused by other pointcut or advice expressions. Mutation operators for named pointcuts can also be applied to anonymous ones, with similar limitations when applied to named pointcuts.

Mutation applied to annotation-based pointcuts: AspectJ 5 allows matchings based on Java annotated code [3], and some specific primitive pointcuts are provided, namely: `@args`, `@this`, `@target`, `@within`, `@withincode` and `@annotation`. In this context, the mutation operators related to pointcuts may also be applied to expressions that include annotation-based pointcut designators, as well as in patterns that include annotated elements. The PWAR and PSDR act specifically over annotation-related code. Figure 2 shows examples of mutants obtained from the PCTT (a) and PCCE (b) operators.

Table 9. Relationship between proposed AspectJ mutation operators and AO fault types.

	Fault Types																								
	F1								F2									F3						F4		
Operator	1	2	3	4	5	6	7	8	1	2	3	4	5	6	7	8	9	1	2	3	4	5	6	1	2	3
PWSR	✓																									
PWIW	✓																									
PWAR	✓																									
PSSR		✓																								
PSWR		✓																								
PSDR		✓																								
POPL	✓	✓																								
POAC	✓	✓	✓	✓																						
POEC	✓	✓	✓	✓			✓																			
PCTT	✓	✓	✓	✓	✓																					
PCCE					✓																					
PCGS				✓	✓																					
PCCR	✓	✓	✓	✓		✓															✓					
PCLO	✓	✓	✓	✓		✓																				
PCCC	✓	✓						✓																		
DAPC															✓											
DAPO															✓											
DSSR														✓												
DEWC																	✓									
DAIC																✓										
ABAR																		✓								
APSR																			✓	✓						
APER																			✓	✓						
AJSC																						✓				
ABHA																				✓						
ABPR																							✓			

○
```
pointcut b():
 call(* *.print())
 &&
 @this(Employee);
```

○
```
pointcut a():
 call(
  @Ann *.set*(..));
```

Δ
```
  pointcut b():
   call(* *.print())
   &&
>  @target(Employee);
```

Δ
```
  pointcut a():
>  execution(
     @Ann *.set*(..));
```

(a) (b)

Figure 2. Mutations applied to annotation-based pointcuts.

5. Cost analysis

This section presents a cost analysis for the application of the proposed set of mutation operators. Although the costs might be estimated based on the number of test cases required to achieve a certain mutant coverage, different complexities of test cases impact directly on the estimates. Therefore, the following analysis is based on the number of mutants generated from two AspectJ applications, which are following briefly described.

Target applications: The first application, called *Toll System Demonstrator* (TSD), includes a subset of requirements of a real-world tolling system. TSD was developed in the context of the AOSD-Europe Project [1] and contains functional and non-functional concerns (e.g. charging variabilities, distribution and logging) implemented within aspects. The TSD implementation includes 25 aspects, 37 compound pointcuts (i.e. pointcut expressions formed by one or more simple expressions), 37 advices and 2 aspect deployment clauses. The second, called *Health Watcher* (HW) [28], is a Web-based application that allows citizens to register complaints regarding health issues. Some aspectised concerns in HW are distribution, persistence and exception handling. HW code includes 11 aspects, 16 compound pointcuts, 14 advices and 6 `declare` clauses.

Obtained results: We have applied all operators manually, totalling to 981 TSD mutants and 388 HW mutants, as shown in Table 10. As expected, operators from Group 1 (third column) resulted in the largest number of mutants (86% for TSD and 88% for HW). These numbers mainly result from the PWIW operator due to the number of possibilities for wildcard insertions into a pointcut expression.

Table 10. TSD and HW mutants.

App	Total	Gr. 1	%	Gr. 2	%	Gr. 3	%
TSD	981	847	86	8	1	126	13
HW	388	342	88	8	1.5	121	10.5

The PCLO and PSSR also contributed to the large number of mutants obtained for Group 1. This was due to: (i) the frequent use of primitive pointcut expression to build compound expressions; and (ii) the extensive use of class hierarchies in both applications (e.g. a simple interface that is implemented by several classes).

We can also observe from Table 10 that operators from Group 2 result in a small number of mutants due to the low use of `declare` statements in both applications. Specifically, only the DAIC and DSSR operators produced mutants for TSD and HW, respectively.

Regarding operators from Group 3 (see Table 10), the ABAR, ABHA and ABPR operators together contributed for 80% and 90% of TSD and HW mutants, respectively. However, the number of mutants for each of them is close to the number of implemented advices.

Table 11. Mutants average per element type.

App.	Aspect	Pointcut	Declaration	Advice
TSD	39	23	4	3
HW	35	21	1	3

Table 11 shows the average of mutants obtained from each group of operators in relation to the targeted elements. For groups which result in the largest numbers of mutants (i.e. pointcuts and advices - see Table 10), the average per element are similar for both applications. For example, while for TSD we have 23 mutants per pointcut and 3 per advice, we have 21 and 3 for the same elements in HW.

The achieved results may be considered as reasonable cost estimates for the proposed operators, since they have been obtained from two real-world applications. Regarding operators for pointcut expressions, despite the large numbers of mutants, they offer a wide coverage of related fault types, systematically but not exhaustively generating mutants for these elements. Moreover, the identification of equivalent mutants may reduce testing efforts, which will be addressed in our forthcoming research.

6. Related work

Mortensen and Alexander [26] propose a hybrid AO testing approach combining coverage and mutation testing, based on a candidate AO fault model [5]. They define three mutation operators based on AspectJ constructs, which involve pointcut expressions (namely PCS-*pointcut strengthening* and PCW-*pointcut weakening*) and aspect precedence declarations (namely PRC-*precedence changing*). However, they do not provide details of syntactic changes and implications of each operator as we present in this paper.

Anbalagan and Xie [6] implement the PCS and PCW operators in an automated tool which exhaustively generates mutants for AspectJ pointcut expressions. The tool also helps the tester to identify the most representative mutants and the equivalent ones through some heuristics. Our set of operators produces a lower number of mutant pointcuts and covers a broader set of AspectJ features.

Lemos et al. [20] present another hybrid approach which involves structural and mutation testing of pointcut expressions. The former consists of composing control flow graphs in order to detect unintended join point selection, while the latter is used to increase the sets of matched join points. However, the authors only highlight the need for a comprehensive set of mutation operators to make the approach more effective. This paper addresses the design of such mutation operators.

The problem of fragile pointcuts [29] leads to faults during software maintenance activities. Some approaches try to deal with this problem, for example, by the application of delta analysis in refactored code [29]. However, even with approaches like that, the problem is not completely solved, since references to elements in the base program still remain in some software artefacts. Chitchyan et al. [10] propose the definition of semantics-based composition at the requirements level, based on the grammatical semantics of the natural language aided by natural language processing tools. This approach, if extended to later stages of the software life cycle, might reduce the coupling between aspects and base concerns and hence pointcut fragility, possibly reducing the number of related faults.

7. Conclusion and future work

The increasing interest in aspect-oriented programming both in academy and industry demands specific testing approaches in order to improve the quality of resulting products. Although mutation testing has been claimed to be important for AO programs evaluation [7, 20, 26], a comprehensive set of mutation operators that model realistic instances of AO fault types has been missing to date.

This paper addressed this issue by proposing a set of mutation operators for AspectJ, which is currently the most consolidated Java-based AOP language. These operators model fault instances devised from a set of fault types we have identified from previous works on AO testing, mainly regarding AO fault models. We have found that most AO fault types, initially characterised upon AspectJ features, actually may occur in general AO implementations and approaches other than AspectJ. In addition, the proposed operators cover many AO fault types for AspectJ-like programs, except faults related to incorrect changes in class structures.

We have also estimated the costs for the application of the proposed operators through two case studies. The targeted systems have been developed in the context of academic and industrial research [1, 28]. We have found that operators for pointcut expressions result in a large number of mutants if compared with operators for other AspectJ features. However, these operators cover a wide but not exhaustive range of pointcut faults with a reasonable cost if compared to a previous mutation generation approach [6].

Our forthcoming research comprises the assessment of test sets considering different testing criteria (e.g. white-box criteria) and the proposed mutation operators. We aim

at assessing the effectiveness of these operators in simulating faults which cannot be revealed by previously proposed AO testing approaches. Moreover, previous results obtained from studies comprising cost reduction techniques (e.g. the identification of sufficient mutation operator sets [8]) may serve as a basis for the definition of cost reduction strategies. Such studies depend on automated tools which are also targeted in our future work.

Acknowledgements: Thanks for the financial support received from the AOSD-Europe Project [1], CAPES (Process 0653/07-1) and FAPESP (Process 05/55403-6).

References

[1] AOSD-Europe Project Home Page, 2007. http://www.aosd-europe.net/ (accessed 08/10/2007).

[2] JBoss AOP Reference Documentation, 2007. http://labs.jboss.com/jbossaop/docs/index.html (accessed 08/10/2007).

[3] The AspectJ Project, 2007. http://www.eclipse.org/aspectj/ (accessed 06/09/2007).

[4] R. Agrawal et al. Design of mutant operators for the C programming language. Report SERC-TR41-P, S.E. Research Center, Purdue University, West Lafayette-USA, 1989.

[5] R. T. Alexander, J. M. Bieman, and A. A. Andrews. Towards the systematic testing of aspect-oriented programs. Report CS-04-105, Dept. of Computer Science, Colorado State University, Fort Collins-USA, 2004.

[6] P. Anbalagan and T. Xie. Efficient mutant generation for mutation testing of pointcuts in aspect-oriented programs. In *Mutation'2006*, pages 51–56. Kluwer, 2006.

[7] J. S. Bækken. A fault model for pointcuts and advice in AspectJ programs. Master's thesis, School of Electrical Engineering and Computer Science, Washington State University, Pullman-USA, 2006.

[8] E. F. Barbosa, J. C. Maldonado, and A. M. R. Vincenzi. Toward the determination of sufficient mutant operators for C. *Soft. Testing, Verif. and Reliability*, 11(2):113–136, 2001.

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

[10] R. Chitchyan, A. Rashid, P. Rayson, and R. Waters. Semantics-based composition for aspect-oriented requirements engineering. In *AOSD'2007*, pages 36–48. ACM Press, 2007.

[11] D. S. Dantas, D. Walker, G. Washburn, and S. Weirich. PolyAML: A polymorphic aspect-oriented functional programming language. In *ICFP'2005*, pages 306–319. ACM Press, 2005.

[12] M. E. Delamaro, J. C. Maldonado, and A. P. Mathur. Interface Mutation: An approach for integration testing. *IEEE Transactions on Soft. Eng.*, 27(3):228–247, 2001.

[13] R. A. DeMillo. Hints on test data selection:Help for the practicing programmer. *IEEE Computer*, 11(4):34–43, 1978.

[14] R. A. DeMillo and A. J. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Soft. Eng.*, 17(9):900–910, 1991.

[15] M. Eaddy, A. Aho, W. Hu, P. McDonald, and J. Burger. Debugging aspect-enabled programs. In *Software Composition'2007*, 2007.

[16] M. Gong, V. Muthusamy, and H. Jacobsen. AspeCt-Oriented C tutorial, 2006. http://research.msrg.utoronto.ca/ACC/ Tutorial (accessed 23/08/2007).

[17] W. Harrison, H. Ossher, and P. L. Tarr. General composition of software artifacts. In *Software Composition'2005*, pages 194–210 (LNCS v.4089). Springer-Verlag, 2006.

[18] R. Johnson et al. Spring - Java/J2EE application framework. Reference Manual Version 2.0.6, Interface21 Ltd., 2007.

[19] G. Kiczales et al. Aspect-oriented programming. In *ECOOP'1997*, pages 220–242 (LNCS v.1241). Springer-Verlag, 1997.

[20] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero, and C. V. Lopes. Testing aspect-oriented programming pointcut descriptors. In *WTAOP'2006*, pages 33–38. ACM Press, 2006.

[21] O. A. L. Lemos, A. M. R. Vincenzi, J. C. Maldonado, and P. C. Masiero. Control and data flow structural testing criteria for aspect-oriented programs. *Journal of Systems and Software*, 80(6):862–882, 2007.

[22] Y. S. Ma, Y. R. Kwon, and J. Offutt. Inter-class mutation operators for Java. In *ISSRE'2002*, pages 352–366. IEEE Computer Society, 2002.

[23] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. $\mu Java$: A mutation system for java. In *ICSE'2006,* p.827-830, ACM Press 2006.

[24] A. P. Mathur and W. E. Wong. An empirical comparison of data flow and mutation based test adequacy criteria. *Soft. Testing, Verif. and Reliability*, 4(1):9–31, 1994.

[25] M. Mezini and K. Ostermann. Conquering aspects with Caesar. In *AOSD'2003*, pages 90–99. ACM Press, 2003.

[26] M. Mortensen and R. T. Alexander. An approach for adequate testing of AspectJ programs. In *WTAOP'2005*, 2005.

[27] A. J. Offutt, J. Voas, and J. Payne. Mutation operators for Ada. Report ISSE-TR-96-06, Dept. Inf. and Soft. Systems Eng., George Mason Universit, Fairfax-USA, 1996.

[28] S. Soares, P. Borba, and E. Laureano. Distribution and persistence as aspects. *Software - Practice & Experience*, 36(7):711–759, 2006.

[29] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *ICSM'2005*, pages 653–656. IEEE Computer Society, 2005.

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

[31] W. E. Wong. *On Mutation and Data Flow*. PhD thesis, Dept. of Computer Science, Purdue University, USA, 1993.

[32] T. Xie and J. Zhao. A framework and tool supports for generating test inputs of AspectJ programs. In *AOSD'2006*, pages 190–201. ACM Press, March 2006.

[33] D. Xu and W. Xu. State-based incremental testing of aspect-oriented programs. In *AOSD'2006*, pages 180–189. ACM Press, 2006.

[34] S. Zhang and J. Zhao. On identifying bug patterns in aspect-oriented programs. In *COMPSAC'2007*, pages 431–438. IEEE Computer Society, 2007.

[35] J. Zhao. Data-flow-based unit testing of aspect-oriented programs. In *COMPSAC'2003*, pages 188–197. IEEE Computer Society, 2003.

Appendix

D

Paper: Automating the Mutation Testing of Aspect-Oriented Java Programs

This appendix presents the full contents of a paper published in the Proceedings of the 5^{th} International Workshop on Automation of Software Test (AST'10). An overview of this work was provided in Chapter 5 of this dissertation. A copyright notice in regard to it is next shown.

[1] `http://www.acm.org/publications/policies/copyright_policy#Retained` - last accessed on 06/07/2010.

Automating the Mutation Testing of Aspect-Oriented Java Programs

Fabiano Cutigi Ferrari
Computer Systems Department
University of São Paulo - Brazil
ferrari@icmc.usp.br

Elisa Yumi Nakagawa
Computer Systems Department
University of São Paulo - Brazil
elisa@icmc.usp.br

Awais Rashid
Computing Department
Lancaster University - UK
marash@comp.lancs.ac.uk

José Carlos Maldonado
Computer Systems Department
University of São Paulo - Brazil
jcmaldon@icmc.usp.br

ABSTRACT

Aspect-Oriented Programming has introduced new types of software faults that may be systematically tackled with mutation testing. However, such testing approach requires adequate tooling support in order to be properly performed. This paper addresses this issue, introducing a novel tool named *Proteum/AJ*. *Proteum/AJ* realises a set of requirements for mutation-based testing tools and overcomes some limitations identified in previous tools for aspect-oriented programs. Through an example, we show how *Proteum/AJ* was designed to support the main steps of mutation testing. This preliminary use of the tool in a full test cycle provided evidences of the feasibility of using it in real software development processes and helped us to reason about the current functionalities and to identify future needs.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*

General Terms

Reliability, Verification, Measurement

Keywords

Mutation testing, Aspect-Oriented Programming, test automation, testing tools.

1. INTRODUCTION

In the last years, Aspect-Oriented Programming (AOP) [18] has been widely investigated as an approach for enhancing software modularity. It has introduced new concepts and programming mechanisms that allow software developers to implement different system functionalities separately. Once implemented, they can be combined together to produce a single executable system that is expected to present enhanced modularity and maintainability [19].

Despite the benefits claimed by the AOP community, the associated programming mechanisms lead to specific types of faults [13, 14]. In this context, mutation testing [12] is a fault-based testing criterion that can systematically explore common mistakes made by aspect-oriented (AO) software developers. It has been widely investigated in several development phases and technologies [17] and recently in AOP field [8, 10, 13].

Mutation testing strongly relies on automated testing tools in order to be introduced in real software development environments. Moreover, testing tools are invaluable resources for research and education, as highlighted by Horgan and Mathur [16]. Examples of mutation-based testing tools that have been used with success by software practitioners in both industry and academia are the *Mothra* [9] and *Proteum* [11] tools. In regard to AO software, we can also identify some attempts to provide tooling support for mutation testing [8, 10]. They all focus on programs written in AspectJ [7], which is the most widely adopted AOP language. Despite that, none of these tools currently provide adequate support to the basic steps performed in mutation testing.

In our previous research we investigated how fault types may occur in AO software and how this can be addressed with mutation testing [13]. In this paper we present *Proteum/AJ*, a novel tool for mutation testing of AspectJ AO programs. *Proteum/AJ* automates a set of AO-specific mutation operators [13] and supports the basic steps of mutation testing [12]. It leverages previous knowledge on developing *Proteum* (**Pro**gram **Te**sting **U**sing **M**utants) [21], a family of tools for mutation testing developed by the Software Engineering group at the University of São Paulo, Brazil.

Proteum/AJ aims at filling the gap of limited support for mutation testing of AO programs by providing a set of functionalities not yet available in current tools. The development of *Proteum/AJ* was guided by a set of basic requirements for mutation tools mostly identified from previous research (Section 2). For instance, it allows the tester to perform incremental testing by evolving the selection of mutation operators, target aspects and the test suite. Equivalent mutants can be automatically identified, and live mutants can be individually executed and analysed.

The tool implements a reference architecture for software testing tools named *RefTEST* [22], from which the main functional modules were derived (Section 3). The preliminary use of the tool (Section 4) evinces the feasibility of using *Proteum/AJ* in real-world software development processes. Moreover, using the tool provided us with valuable feedback with respect to the implemented functionalities and issues to be addressed in our future research (Sections 5 and 6).

AST '10, May 2-8 2010, Cape Town, South Africa

Table 1: Requirements for mutation-based testing tools.

Requirement	Description
1. Test case handling*	Execution, inclusion/exclusion, activation/deactivation of test cases.
2. Mutant handling*	Creation, selection, execution and analysis of mutants.
3. Adequacy analysis*	Computation of mutation score generation of statistical reports.
4. Reducing test costs†	Auto-generation of test cases and minimisation of test sets.
5. Unrestricted program size†	Size of the target application should not be restricted by the tool.
6. Support for testing strategies⊙	Different testing strategies (e.g. ordered application of mutation operators) should be supported by the tool.
7. Independent test configuration	The test configuration (e.g. test inputs, outputs and executing environment) should not be restricted by the tool.

*From Delamaro and Maldonado [11]. ⊙From Vincenzi et al. [23]. †From Horgan and Mathur [16].

2. BACKGROUND AND RELATED WORK

Mutation testing is a fault-based testing criterion which relies on common mistakes practitioners make during software development [12]. Such mistakes are modelled into *mutation operators* as a set of transformation rules. Given an original product[1] P, the criterion requires the creation of a set M of *mutants* of P, resulting from the application of the mutation operators to it. 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 improve 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.

A recent survey undertaken by Jia and Harman [17] showed that mutation testing has been extensively and increasingly investigated in the last three decades. According to their survey, over 30 tools have been implemented to automate mutation testing at several levels of software abstraction, ranging for formal specification to source code level testing. Specifically regarding source code testing, *Mothra* [9] for Fortran programs and *Proteum* [11] for C programs are the two most widely investigated and used tools.

From the experience in developing and using *Proteum*, Delamaro and Maldonado [11] listed a minimal set of requirements that should be provided by mutation-based testing tools. Basically, their list regards test case handling, mutant handling and adequacy analysis. We enhanced this list of requirements with some common features observed by Horgan and Mathur [16] in a suite of testing tools used in their experiments, including *Mothra*. The resulting list is shown in Table 1.

Recently, Vincenzi et al. [23] proposed the *Muta-Pro* process to support mutation-based testing. It aims at synergically integrating several approaches to reduce the high cost of mutant execution and analysis tasks. The process was designed to support experimentation in software testing, for instance, supporting incremental testing strategies along the process. From *Muta-Pro* we identified an extra requirement that is listed in Table 1. Moreover, we included the *Independent test configuration* feature, which we consider fundamental for testing modern software systems. For such systems (e.g. enterprise information systems, frameworks and software product lines), the tool should not restrict, for instance, the test input and test output formats, neither the configuration of the executing environment. Such properties should be delegated for specific test execution mechanisms whose outputs (e.g. results and reports) could be extracted and analysed by the testing tool.

[1] P can be a program specification, source code or any other executable software artefact [23].

2.1 Current Mutation Tools for AO Programs

We identified two implementations of mutation tools for AO software, both for AspectJ programs. Anbalagan and Xie [8] implemented a tool that performs mutations of pointcut expressions. Mutants are produced through the use of wildcards as well as by using naming parts of original pointcuts and join points identified from the base code. Based on heuristics, the tool automatically ranks the most representative mutants pointcuts, which are the ones that more closely resemble the original pointcuts. If a mutant selects the same set of join point as does the original expression, it is automatically classified as equivalent. The final output is a list of the ranked mutants.

Anbalagan and Xie [8]'s tool has some limitations[2] with respect to the requirements presented in Table 1. It does not support all steps of mutation testing. Basically, the tool is limited to the creation and classification of mutants based on a very small set of mutation operators. No support for test case and mutant handling is provided. Other particular limitation regards the equivalent mutant detection. In AspectJ, pointcuts can be reused in different modules, hence several join point matchings across the system may be affected by a single pointcut mutation. However, the tool analyses pointcuts individually, potentially overlooking the impact of a mutation on other modules.

More recently, Delamare et al. [10] presented the *AjMutator* tool that implements a subset of the operators for pointcut expressions proposed in our previous research [13]. *AjMutator* parses pointcut expressions from aspects individually and performs the mutations. The modified expressions are reinserted into the code, generating the mutants. The automatic detection of equivalent mutants relies on join point matching information provided by the abc compiler [1], an alternative compiler for AspectJ programs. *AjMutator* allows the tester to run JUnit test cases and identifies non-compilable and dead mutants. The tool outputs an XML file that contains information about every mutant handled (e.g. mutant status, pointcut ID and aspect ID).

Although *AjMutator* tackles some of the limitations observed in Anbalagan and Xie's tool [8], it still misses some basic functionalities to properly support mutation testing. For instance, it does not allow for mutation operator selection, hence hindering testers to apply different strategies. The mutant analysis itself is limited to the automatic detection of equivalent mutants; other mutant handling features such as individual mutant execution and manual classification of mutants are not available. Specific implementation issues include the lack of support to Java 5 features [6] in the

[2] Since the tool is not available for download, our analysis is based on the description provided by the authors [8].

Table 2: Limitations of current tools for mutation testing of AO programs.

Req.	AjMutator [10]	Anbalagan and Xie's tool [8]
#1	Supports test case execution. However, tests cannot be incrementally added, activated or deactivated.	No support for test case management and execution.
#2	All implemented operators are applied at once. Does not support manual mutant inspection and individual mutant execution.	All implemented operators are applied at once. No support for mutant handling is provided.
#3	Equivalent mutants are automatically identified. Calculates mutation score but does not create statistical reports.	Equivalent mutants are automatically identified. Does not compute mutation score neither creates statistical reports.
#4	No support for test case generation or minimisation of test suites.	No support for test case generation or minimisation of test suites.
#5	n/a	n/a
#6	No support for testing strategies (e.g. incremental use of operators or addition of test cases).	No support for testing strategies (e.g. incremental use of operators or addition of test cases).
#7	Test execution is configured through Ant tasks (i.e. it is delegated to external tools).	No support for test execution.

abc compiler, as well as the lack of support for incremental testing since the results are reset in every test run.

A summary of the addressed requirements and limitations of these tools is presented in Table 2. The requirement numbers are presented in the first column and "n/a" means that we could not find such information in the original work. In the next sections we describe how we designed the *Proteum/AJ* tool to overcome some of the current limitations.

3. THE *PROTEUM/AJ* TOOL

Proteum/AJ is a tool for mutation testing of AO Java programs written in AspectJ. Table 3 summarises up front how *Proteum/AJ* addresses the requirements for mutation tools listed in Section 2. It also compares our tool with the two previous tools described in Section 2.1. In short, *Proteum/AJ* supports the four main steps of mutation testing, as originally described by DeMillo et al. [12]: (i) the original program is executed on the current test set and test results are stored; (ii) the mutants are created based on a mutation operator selection that may evolve in new test cycle iterations; (iii) the mutants can be executed all at once or individually, as well as the test set can be augmented or reduced based on specific strategies; and (iv) the test results are evaluated so that mutants may be set as dead or equivalent, or mutants may remain alive.

Table 3: Addressed requirements.

Requirement	*Proteum/AJ*	*AjMutator*	Anbalagan
1. Test case handling	partial	partial	no
2. Mutant handling	yes	partial	partial
3. Adequacy analysis	partial	partial	partial
4. Reducing test costs	no	no	no
5. Unrestricted program size	yes	n/a	n/a
6. Support for testing strategies	partial	no	no
7. Independent test configuration	yes	yes	no

More details about how requirements are fully or partially addressed by *Proteum/AJ* are further discussed in Section 5. Next we present an overview of the tool architecture (Section 3.1) and its data model (Section 3.2). The main supporting technologies and implementation details are presented in Sections 3.3 and 3.4, respectively.

3.1 Architecture

Figure 1 depicts the architecture of *Proteum/AJ*. The architecture is based on a reference architecture for software testing tools called *RefTEST* [22]. *RefTEST* is based on separation of concerns (SoC) principles, the Model-View-Controller (MVC) and three tier architectural patterns, and the ISO/IEC 12207 standard for Information Technology. *RefTEST* encourages the use of aspects as the mechanism for integrating the core activities of a testing tool with tools that automates supporting and organisational software engineering activities defined in ISO/IEC 12207 (e.g. planning, configuration management and documentation tools). Moreover, aspects are also encouraged for integrating services such as persistence and access control. *Proteum/AJ* benefited mainly from the reuse of domain knowledge contained in *RefTEST*. The instantiation of the architecture provided us with guidance on how we could structure the tool in terms of functionalities and module interactions.

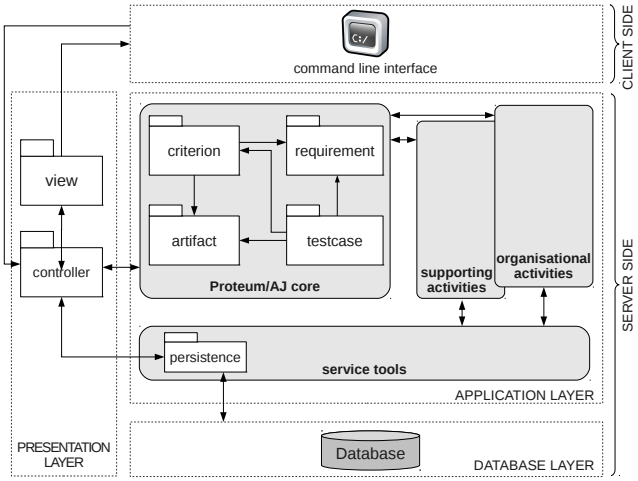


Figure 1: *Proteum/AJ* architecture.

The modules currently implemented in *Proteum/AJ* are shown as UML packages in Figure 1. The core of the tool comprises the four main concepts that should be handled by testing tools, as proposed in *RefTEST* [22]: *testing criterion*, *artifact*, *test requirement* and *test case*. Some of them map directly to the requirements presented in Table 1. For instance, testcase maps to the *"test case handling"* requirement; and criterion maps to both *"mutant handling"* and *"adequacy analysis"* requirements. The former provides functionalities for running and managing test cases in *Proteum/AJ*, while the latter is responsible for handling tasks related to testing criterion itself (e.g. generating, compiling and analysing mutants).

The artifact and requirement modules comprise, respectively, the artefacts under test (i.e. the AspectJ source code files) and the test requirements (i.e. the generated mutants). The controller module is in charge of receiving requests from the client and properly invoking the modules present in the application layer, which include core functionalities and database-related procedures. The controller

is also responsible for updating the view that is presented to the client. In *Proteum/AJ*, the view is basically formed by test execution feedback that is displayed to the users.

3.2 Data Model

Figure 2 shows the data model of *Proteum/AJ* using the Extended Entity-Relationship notation. In the *Proteum/AJ* database, a TestProject element represents a test project and belongs to a user (a User element in Figure 2). Each TestProject selects one or more AspectJ source code files (represented by the SourceCode) that are the targets of one or more mutation operators (MutationOperatorBean element). The mutations implemented by the mutant operators are performed on the SourceCode elements, resulting in a set of mutants (Mutant element). Each TestProject also executes a set of test cases (AntTestCase element), which are also executed against the mutants within the same test project.

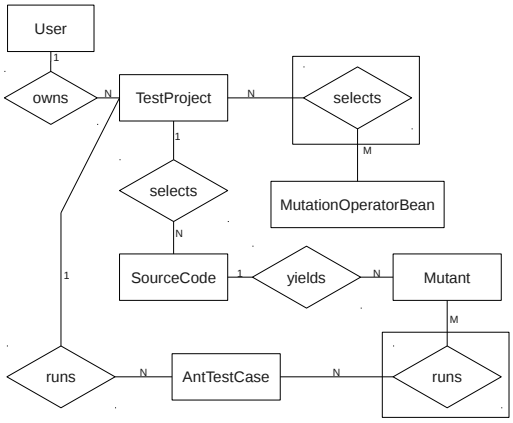


Figure 2: *Proteum/AJ* data model.

3.3 Supporting Technologies

We employed a set of technologies to implement *Proteum/AJ* functionalities. The main ones are:

AspectJ-front: In *Proteum/AJ*, parsing and pretty-printing of AspectJ source code are supported by the AspectJ-front tools suite [3]. The AspectJ-front parser converts AspectJ code into an abstract syntax tree (AST) represented in aterm notation. aterm is a format for exchanging structured data between tools [4]. Figure 3 exemplifies how an `after returning` AspectJ advice (top part) is represented in aterm notation (bottom part).

Original advice in AspectJ

```
after(Customer caller, Customer receiver, boolean iM)
   returning(Connection c) : createConnection(caller, receiver, iM) {
   ...
}
```

The same advice in aterm notation

```
AdviceDec([],
 After([Param([],ClassOrInterfaceType(TypeName(
          Id("Customer")),None()),Id("caller")),
        Param([],ClassOrInterfaceType(TypeName(
          Id("Customer")),None()),Id("receiver")),
        Param([],Boolean(),Id("iM"))],
        Some(Returning(Param([],ClassOrInterfaceType(
          TypeName(Id("Connection")),None()),Id("c"))))),
 None(),
 NamedPointcut(PointcutName(Id("createConnection")),[ ... ]),
 Block([ ... ])
)
```

Figure 3: Example of aterm representation.

iBATIS: We used the iBATIS data mapper framework [5] in *Proteum/AJ* to handle data storage. With iBATIS, persistent objects are mapped to database table by means of XML-based procedures that can be invoked with pure Java code. The framework also provides full support for database connection pooling, caching and transactions.

Ant: The Ant tool [2] provides facilities for compiling and manipulating files in build processes through a set of XML-based procedures called *tasks*. Such tasks can be programmatically invoked, what facilitates the integration of Ant with customised tools such as *Proteum/AJ*. Examples of Ant tasks used within *Proteum/AJ* are AspectJ source code compilation (iajc task), test case execution (junit task) and file decompression (unzip task).

3.4 Implementation Details

This section describes the main modules of the *Proteum/AJ* core: the criterion and the testcase modules. A simplified class diagram of these modules is depicted in Figure 4. Details are provided in the sequence.

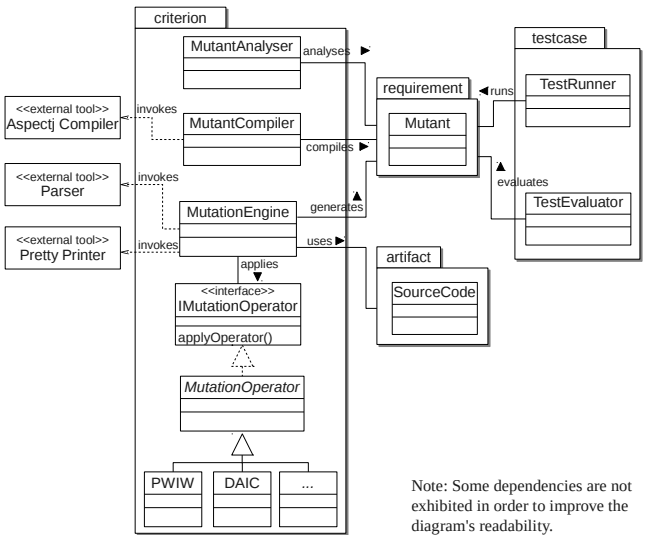


Figure 4: *Proteum/AJ* core.

3.4.1 *The* criterion *Module*

The criterion module includes functionalities for generating, compiling and analysing mutants. These functionalities are implemented within three main classes: `MutationEngine`, `MutantCompiler` and `MutantAnalyser`.

The `MutationEngine` class is responsible for generating mutants from AspectJ source code by controlling the application of the mutation operators. The operators implemented in *Proteum/AJ* were proposed in our previous research [13]. They model faults that are likely to occur in AspectJ-like programs. These operators are summarised in Table 4. They are split into three groups, according to the main AOP constructs[3] they are related to, namely: Group G1 – pointcut-related operators; Group G2 – `declare`-like-related operators; and Group G3 – advice-related operators. All mutation operators implement the `IMutationOperator` interface, what facilitates the inclusion of new mutation operators into the tool. Mutations are performed in the aterm representation of source code elements, which are manipulated as Java `String`

[3]More information about the main AOP constructs can be found in the AspectJ documentation [7]

Table 4: Mutation operators implemented in Proteum/AJ (adapted from [13]).

Operator		Description/Consequences
G1	PWSR†	Pointcut weakening by replacing a type with its immediate supertype in pointcut expressions
	PWIW*	Pointcut weakening by inserting wildcards into pointcut expressions
	PWAR*	Pointcut weakening by removing annotation tags from type, field, method and constructor patterns
	PSSR†	Pointcut strengthening by replacing a type with its immediate subtype in pointcut expressions
	PSWR*	Pointcut strengthening by removing wildcards from pointcut expressions
	PSDR*	Pointcut strengthening by removing `declare @` statements, used to insert annotations into base code elements
	POPL*?	Pointcut weakening or strengthening by changing parameter lists of primitive pointcut designators
	POAC*?	Pointcut weakening or strengthening by changing `after [retuning|throwing]` advice clauses
	POEC*	Pointcut weakening or strengthening by changing exception throwing clauses
	PCTT*?	Pointcut changing by replacing a `this` pointcut designator with a `target` one and vice versa
	PCCE	Context changing by switching `call/execution/initialization/preinitialization` pointcuts designators
	PCGS*	Pointcut changing by replacing a `get` pointcut designator with a `set` one and vice versa
	PCCR*?	Pointcut changing by replacing individual parts of a pointcut composition
	PCLO*	Pointcut changing by changing logical operators present in type and pointcut compositions
	PCCC*?	Pointcut changing by replacing a `cflow` pointcut designator with a `cflowbelow` one and vice versa
G2	DAPC	Aspect precedence changing by alternating the order of aspects involved in `declare precedence` statements
	DAPO	Arbitrary aspect precedence by removing `declare precedence` statements
	DSSR	Unintended exception handling by removing `declare soft` statements
	DEWC	Unintended control flow execution by changing `declare error/warning` statements
	DAIC	Unintended aspect instantiation by changing `perthis/pertarget/percflow/percflowbelow` deployment clauses
G3	ABAR	Advice kind changing by replacing a `before` clause with an `after [retuning|throwing]` one and vice versa
	APSR	Advice logic changing by removing invocations to `proceed` statement
	APER	Advice logic changing by removing guard conditions which surround `proceed` statements
	AJSC	Static information source changing by replacing a `thisJoinPointStaticPart` reference with a `thisEnclosingJoinPointStaticPart` one and vice versa
	ABHA	Behaviour hindering by removing implemented advices
	ABPR	Changing pointcut-advice binding by replacing pointcuts which are bound to advices

†Not implemented in the current version of *Proteum/AJ*.

*Considered for automatic detection of equivalent mutants. ?Impacts quantification of join point with dynamic residues.

objects by the `MutationEngine` class. The engine uses a set of tailor-made string handlers that enables such manipulation (e.g. code offset localising and replacement).

The `MutationEngine` class also invokes the AspectJ-front pretty-printer tool to build AspectJ code. This tool performs a first check that ensures each mutant code is syntactically correct when considered in isolation, i.e. before the aspect is woven into the base code. However, it cannot guarantee that base code and the mutant aspect can be successfully woven together. Such verification can only be performed by an AspectJ compiler such as ajc [7].

Invoking the ajc compiler is assigned to the `MutantCompiler` class. It is achieved through the iajc Ant task provided with the AspectJ API. Non-compilable mutants are identified at this step. Such mutants are classified as *anomalous* and are further discarded when the mutation score is calculated.

Finally, the `MutantAnalyser` class compares the original and the mutated code. More specifically, it automatically identifies equivalent mutants for some pointcut-related operators based in the weaving output produced by the compiler. This output is a valuable information that includes details of all matching between aspects and base code [7]. The `MutantAnalyser` also generates reports that show the modified portions of code for each mutant, for each target aspect. These reports are necessary to help the testers identify either equivalent mutants or the need for additional test cases.

3.4.2 *The* `testcase` *Module*

The role of the testcase module is managing test execution and evaluation. Within it, the `TestRunner` class implements test runner methods for the original and the mutant applications. In particular, it calls the `MutantCompiler` class to produce the executable mutant application. If the compilation succeeds, the automatic detection of equivalent mutants is performed by the `MutantAnalyser` class[4]. If the original application and the mutant are not equivalent, the tests are executed and the results are stored for further evaluation.

The `TestEvaluator` class evaluates the test results obtained from the execution of the original application and the mutants. It contrasts test case outputs, identifies the differences and decides whether a given mutant should be killed or not.

4. EXAMPLE

This section describes the use of *Proteum/AJ* based on the modules presented in the previous section. It starts presenting an example (Section 4.1) that is used along the section to exemplify the main steps performed with *Proteum/AJ*. In the sequence, we show the results obtained with this example along a full test session.

4.1 The Telecom Application

To introduce the *Proteum/AJ* functionalities, we selected an AspectJ application called Telecom. It is a telephony system simulator which is originally distributed with AspectJ [7]. In Telecom, timing and billing of phone calls are handled by aspects. The version we use in this example includes six classes and three aspects. It extends the original version to support a different type of charging for mobile calls [20].

Figure 5 shows the partial implementation of the three aspects present in Telecom. The `Timing` aspect measures the duration of the calls, which are logged by the `TimerLog` aspect. `Billing` implements the billing concern and ensures calls are charged accordingly. Note that Figure 5 only shows AspectJ code that is relevant for the mutation operators implemented in *Proteum/AJ*, i.e. pointcuts, advices and `declare`-like statements. The full implementation can be found at `http://www.labes.icmc.usp.br/~ferrari/proteumaj/`.

[4]More details about equivalent mutant detection in Section 4.

```
public aspect Timing {
  ...
  after(Connection c) returning : target(c) &&
        call(void Connection.complete()) {
    getTimer(c).start();
  }

  pointcut endTiming (Connection c) :target(c) &&
        call(void Connection.drop());

  after(Connection c) returning : endTiming(c) {
    getTimer(c).stop();
    c.getCaller().totalConnectTime += getTimer(c).getTime();
    c.getReceiver().totalConnectTime += getTimer(c).getTime();
  }
}


public aspect TimerLog {
  after(Timer t) returning : target(t) && call(* Timer.start()) {
    System.err.println("Timer started: " + t.startTime);
  }

  after(Timer t) returning : target(t) && call(* Timer.stop()) {
    System.err.println("Timer stopped: " + t.stopTime);
  }
}


public aspect Billing {
  declare precedence : Billing, Timing ;
  ...
  pointcut createConnection (Customer caller, Customer receiver,
           boolean iM):
       args(caller, receiver, iM) && call(Connection+.new(..) );

  after(Customer caller, Customer receiver, boolean iM) returning(Connection c):
        createConnection(caller, receiver, iM) {
    if(receiver.getPhoneNumber().indexOf("0800") == 0)
      c.payer = receiver;
    else
      c.payer = caller;
    c.payer.numPayingCalls += 1;
  }
  ...
  after(Connection conn) returning: Timing.endTiming(conn) {
    long time = Timing.aspectOf().getTimer(conn).getTime();
    long rate = conn.callRate();
    long cost = rate * time;
    if(conn.isMobile()) {
      if(conn instanceof LongDistance) {
        long receiverCost = MOBILE_LD_RECEIVER_RATE * time;
        conn.getReceiver().addCharge(receiverCost);
      }
    }
    getPayer(conn).addCharge(cost);
  }
  ...
  void around(boolean isMobile): execution(Connection.new(.., boolean) ) &&
        args(.., isMobile) {
    System.err.println("The established Connection includes a mobile device");
    if(isMobile)
      proceed(isMobile);
    else {
      proceed(isMobile);
    }
  }
} // end of Billing
```

Figure 5: Partial code of the Telecom application.

4.2 Testing with Proteum/AJ

As introduced early in Section 1 as well in this section, *Proteum/AJ* supports the main steps performed in a typical mutation-based testing process. Figure 6 depicts how this is achieved with the tool. It shows the execution sequence of the main modules and the inputs/outputs of each of them. The modules are invoked through parameterised scripts that are executed via command line in a shell console.

Pre-processing the original application

As shown in Figure 6, the target application must be a compressed file that is submitted to *Proteum/AJ*. This file contains all modules (classes, aspects and libraries) of the application under test. The Application Handler module then runs a pre-processing step, whose outputs are the decompressed original application and a list of target aspects. This step creates the test projects in the *Proteum/AJ* database, according to the schema presented in Figure 2. Such data is handled along the test process. The Application Handler also compiles the original application through the iajc Ant task.

Execution of the original application

The decompressed application is sent to the Test Runner module together with the test case files. The Test Runner executes the application on the available test set by invoking the junit Ant task. The results are stored for further evaluation of mutants.

We designed an initial test set that includes 18 test cases that target all modules of the Telecom system. This test set covers all control flow- and data flow-based requirements computed according to the approach for structural testing of AO programs proposed by Lemos et al. [20]. It took ∼1.6 second to execute the 18 tests in *Proteum/AJ*.

Generation of mutants

The Mutation Engine receives as input the list of target aspects identified by the Application Handler and the set of mutation operators selected by the tester. It produces the set of mutants that are passed to the Mutant Compiler.

The 24 operators implemented in *Proteum/AJ* produced a total of 111 mutants for the three aspects of Telecom in ∼1.8 second. Table 5 summarises the results. The mutants are grouped according to the nature of the mutation operators (see Table 4).

Table 5: Telecom mutants.

	Group G1	Group G2	Group G3	Total
`Billing`	30	2	15	47
`TimerLog`	26	0	6	32
`Timing`	26	0	6	32
Total	82	2	27	111

Execution of mutants

The execution of mutants requires a series of steps in *Proteum/AJ*. Initially, each mutant is sent to the Mutant Compiler module. This module invokes the ajc compiler through the iajc Ant task provided with the AspectJ API [7]. The Mutant Compiler detects non-compilable mutants which are classified as *anomalous*. For compilable mutants, the weaving information produced by the ajc compiler is collected at this stage and further used by the Mutant Analyser module.

The Mutant Analyser compares the ajc weaving output of the original application and the mutants. *Proteum/AJ* uses such information to decide whether mutants created by a subset of operators from Group G1 (tagged with the symbol "⋆" in Table 4) are equivalent to the original aspects. The join point quantification yielded by these operators can be checked at compilation time so that equivalence can be automatically obtained. However, some pointcut designators (e.g. `if` and `cflow`) results in dynamic tests being inserted into the affected join points during the weaving process. These tests, also called *dynamic residues* [15], imply that final join point matching can only be resolved at runtime.

We identified mutation operators that are likely to impact the quantification of join points that contain dynamic residues (tagged with the symbol "?" in Table 4). *Pro-*

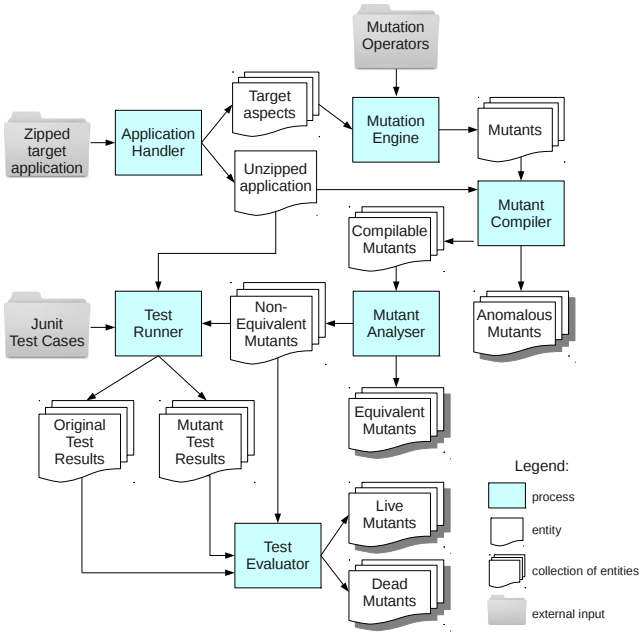


Figure 6: *Proteum/AJ* execution flow.

teum/AJ, in turn, gives the tester the option of deactivating the automatic equivalence detection by setting a flag sent to the Mutant Compiler module. In doing so, the tester becomes in charge of manually analysing the mutants created by those operators during the *Analysis of mutants* step.

As for the original application, the Test Runner module runs all compilable, non-equivalent mutants on the current test set using the iajc Ant task. The results are stored and sent to the Test Evaluator module. The evaluator contrasts the results with the original results and identifies which mutants should be killed. Mutants can also be killed by execution timeout. In this case, the timeout is parameterised either within the junit Ant task or via console script.

Table 6: Results for Telecom after running the tests.

	Alive	Dead	Equivalent*	Anomalous	Total
`Billing`	8	13	14	12	47
`TimerLog`	16	0	16	0	32
`Timing`	10	6	16	0	32
Total	34	19	46	12	111

*automatically identified.

According to the numbers presented in Table 6, 99 mutants of Telecom compiled with success and 12 were classified as anomalous. The execution of the original test set took ∼3 minutes (or ∼1.62 second per mutant), resulting in 19 dead mutants and 46 automatically identified as equivalent.

Analysis of mutants

This is the most costly step performed in mutation testing since it requires manual intervention from the tester. To support it, the Mutant Analyser module creates a set of plain text reports that show the differences between the original and mutated code. These reports show the modified parts of the code for each mutant, hence providing guidance for the tester while analysing the changes. The tester then decides which mutants should be set as equivalent and updates their status through the Mutant Analyser module.

The Mutant Analyser also computes the mutation score using the formula below, where dm is the number of dead mutants, cm is the number of compilable mutants and em is the number of equivalent mutants:

$$MS = (dm)/(cm - em)$$

The initial tests yielded in a mutation score of 0.3585. We analysed the 34 live mutants, what resulted in 12 mutants identified as equivalent. The remaining ones were killed by five extra test cases that we added to the initial test set. The final results are shown in Table 7. The final test set includes 23 test cases that lead to a mutation score of 1.0.

Table 7: Final results for Telecom.

	Alive	Dead	Equivalent	Anomalous	Total
`Billing`	0	19	16	12	47
`TimerLog`	0	12	20	0	32
`Timing`	0	12	20	0	32
Total	0	43	56	12	111

5. DISCUSSION

This section analyses how *Proteum/AJ* addresses the requirements presented in Section 2. We also discuss limitations and issues related to the implementation of the tool.

How does Proteum/AJ fulfil the requirements?

Proteum/AJ allows the tester to manage mutants in several ways. For example, mutants can be created, recreated and individually selected for execution. The execution can also be restricted to live mutants only, and these can be manually set as equivalent and vice versa, that is, equivalent mutants can be reset as alive.

The size of the application under test is not constrained by the tool. Decompression and compilation tasks are delegated to third-party tools configured through Ant tasks. We have been experimenting *Proteum/AJ* with larger applications (e.g. nearly 200 classes and 40 aspects [14]) and the tool has shown to be able to handle larger sets of mutants (∼3,600 in total), despite the compilation issues discussed in the sequence. The test setup is also fully configurable through junit Ant tasks that are invoked by *Proteum/AJ*. In this way, there are only minor dependencies between the test execution configuration and the tool.

Proteum/AJ also enables the tester to import and execute new test cases within an existing test project, although it does not support test case activation/deactivation. Despite that, the *Proteum/AJ* database was designed to support such features in the future. The mutation score can be computed at any time after the first tests have been executed; however, the creation of statistical reports is another functionality that is planned for the upcoming releases.

Proteum/AJ implementation issues and limitations

Since *Proteum/AJ* runs JUnit test cases to evaluate the mutants, we can consider the tool implements the *firm mutation* approach. JUnit test cases have the ability of configuring partial program runs (e.g. a single method execution) and performing assertions in the course of the execution. Firm mutation is defined by Woodward and Halewood [24] as "*the situation where a simple error is introduced into a program and which persists for one or more executions, but not for the entire program execution*". Thus, similarly to the *AjMutator* tool [10], the evaluation of mutants based on partial program executions implemented in JUnit tests characterises firm mutation in *Proteum/AJ*.

Regarding the mutant compilation step, the ajc compiler allows for two types of weaving [7]: compile-time and post-

compile weaving[5]. The former is the simplest approach and is performed when the source code is available. The latter, on the other hand, is carried out for existing class files. In *Proteum/AJ*, the compiler directives are provided by the tester through the iajc Ant task. That is, the Ant task defines how the application will be compiled. So far we have only experienced *Proteum/AJ* with applications configured for weaving based on source code (compile-time). However, the compilation time may become a bottleneck for larger systems. Therefore, reducing the compilation time (e.g. through post-compile weaving) is one of the enhancements we planned for the next releases of the tool. Nevertheless, full weaving would still be required in the occurrence of inter-type declarations in the system [7], what is a very recurring situation we have noticed in complex AO systems [14].

To run *Proteum/AJ*, the compressed file submitted to *Proteum/AJ* is expected to include an Ant build file named build.xml. This build file must include three tasks that will be used by *Proteum/AJ*: (i) compile, which specifies how to compile the application; (ii) full-test, which specifies how to run the full test set (possibly) included with the application; and (iii) single-test, which specifies how to run a single JUnit test file. The tester can also provide a file named targets.lst within the compressed file. It contains a list of target aspects that is parsed by the Application Handler module (see Section 4). Optionally, target aspects and test case files can be added to the test project after it has been created.

6. FINAL REMARKS

In this paper we presented *Proteum/AJ*, a tool that automates the mutation testing of aspect-oriented AspectJ programs. The tool was planned to address some limitations of previous tools with the same intent [8, 10]. Its development was guided by a reference architecture for software testing tools [22] and by a set of requirements mostly identified from previous research on test automation [11, 16, 23]. The reference architecture established the main modules that compose the tool. On the other hand, the requirements defined the set of functionalities implemented within those modules.

We described *Proteum/AJ*'s characteristics through an example of use. The tool implements a set of mutation operators [13] that subsumes previous implementations [8, 10]. *Proteum/AJ* is able to generate and manage mutants for multiple aspects within a single test project. The mutant sets can be augmented or reduced along the test cycles, as well as the test suites that can be evolved in order to achieve higher test coverages (i.e. enhanced mutation scores). The example of use showed that employing *Proteum/AJ* in real software development processes is an achievable goal.

Our next research steps include the conduction of extra case studies that will comprise larger aspect-oriented systems. We also intend to fulfil the requirements not yet addressed in *Proteum/AJ*, specially the ones that regards test case handling and adequacy analysis. Besides, considering the complementary nature of testing approaches, we also aim to investigate how the tool can be integrated with other testing tools in order to share common resources such as test projects and test suites.

Acknowledgments

We would like to thank Martin Bravenboer from the Stratego/XT Team for the ready replies to doubts and required fixes in the AspectJ-front toolset. We also thank the financial support received from FAPESP (grant 05/55403-6), CAPES (grant 0653/07-1), EC Grant AOSD-Europe (IST-2-004349), and CPNq - Brazil.

[5]A third type, the load-time weaving, is basically the post-compile weaving postponed to the moment classes are loaded to the JVM [7].

References

[1] abc: The aspectbench compiler for AspectJ. http://abc.comlab.ox.ac.uk/ - accessed on 01/03/2010.

[2] Ant. http://ant.apache.org/ - accessed on 01/03/2010.

[3] AspectJ-front. http://strategoxt.org/Stratego/AspectJFront - accessed on 01/03/2010.

[4] ATerm format. http://www.program-transformation.org/Tools/ATermFormat - accessed on 01/03/2010.

[5] iBATIS data mapper. http://ibatis.apache.org/ - accessed on 01/03/2010.

[6] J2SE 5.0: New features and enhancements, 2004. http://java.sun.com/j2se/1.5.0/docs/relnotes/features.html - accessed on 01/03/2010.

[7] AspectJ documentation, 2010. http://www.eclipse.org/aspectj/docs.php - accessed on 01/03/2010.

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

[9] B. J. Choi et al. The Mothra tool set. In *HICSS'89*, volume 2, pages 275–284, 1989.

[10] R. Delamare, B. Baudry, and Y. Le Traon. AjMutator: A tool for the mutation analysis of aspectj pointcut descriptors. In *Mutation'09 Workshop*, pages 200–204. IEEE Computer Society, 2009.

[11] M. E. Delamaro and J. C. Maldonado. Proteum: A tool for the assessment of test adequacy for C programs. In *PCS'96*, pages 79–95, 1996.

[12] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–43, 1978.

[13] F. C. Ferrari, J. C. Maldonado, and A. Rashid. Mutation testing for aspect-oriented programs. In *ICST'08*, pages 52–61. IEEE Computer Society, 2008.

[14] F. C. Ferrari et al. An exploratory study of fault-proneness in evolving aspect-oriented programs. In *ICSE'10*. ACM Press, 2010. (to appear).

[15] E. Hilsdale and J. Hugunin. Advice weaving in AspectJ. In *AOSD'04*, pages 26–35. ACM Press, 2004.

[16] J. Horgan and A. Mathur. Assessing testing tools in research and education. *IEEE Software*, 9(3):61–69, 1992.

[17] Y. Jia and M. Harman. An analysis and survey of the development of mutation testing. Tech.Report TR-09-06, CREST Centre, King's College, London - UK, 2009.

[18] G. Kiczales et al. Aspect-oriented programming. In *ECOOP'97*, pages 220–242 (LNCS v.1241). Springer-Verlag, 1997.

[19] R. Laddad. Aspect-oriented programming will improve quality. *IEEE Software*, 20(6):90–91, 2003.

[20] O. A. L. Lemos, A. M. R. Vincenzi, J. C. Maldonado, and P. C. Masiero. Control and data flow structural testing criteria for aspect-oriented programs. *Journal of Systems and Software*, 80(6):862–882, 2007.

[21] J. C. Maldonado et al. Proteum: A family of tools to support specification and program testing based on mutation. In *Mutation 2000 Symposium (Tool Session)*, pages 113–116. Kluwer, 2000.

[22] E. Y. Nakagawa, A. S. Simão, F. C. Ferrari, and J. C. Maldonado. Towards a reference architecture for software testing tools. In *SEKE'07*, pages 157–162, 2007.

[23] A. M. R. Vincenzi, A. S. Simão, M. E. Delamaro, and J. C. Maldonado. Muta-Pro: Towards the definition of a mutation testing process. *Journal of the Brazilian Computer Society*, 12(2):49–61, 2006.

[24] M. Woodward and K. Halewood. From weak to strong, dead or alive? An analysis of some mutation testing issues. In *Workshop on Soft. Testing, Verification, and Analysis*, pages 152–158. IEEE Computer Society, 1988.

Appendix

E

Paper: Towards the Practical Mutation Testing of Aspect-Oriented Java Programs

This appendix presents the full contents of a paper submitted to the Science of Computer Programming journal. The sections that describe the two case studies were replicated in Chapter 6 of this dissertation.

Towards the Practical Mutation Testing of Aspect-Oriented Java Programs

Fabiano Cutigi Ferrari[a,*], Awais Rashid[b], José Carlos Maldonado[a]

[a] *Departamento de Sistemas de Computação, Universidade de São Paulo (ICMC/USP)*
São Carlos - SP - Brazil
[b] *Computing Department, Lancaster University*
Lancaster - United Kingdom

Abstract

Mutation testing is a widely explored test selection criterion that focuses on recurring faults observed in the software. It helps to demonstrate the absence of prespecified faults and sounds to be an adequate mean to deal with testing-related specificities of contemporary programming techniques such as Aspect-Oriented Programming. However, to date the few initiatives for customising the mutation testing for aspect-oriented (AO) programs show either limited coverage with respect to the range of simulated faults, or a need for both adequate tool support and proper evaluation in regard to properties like application cost and effectiveness. This paper tackles these limitations by describing a comprehensive mutation-based testing approach for AO programs. The approach encompass the definition of a set of mutation operators for programs written in AspectJ and the implementation of a tool that automates the approach. The results of two evaluation studies with different sets of AO applications show that the mutation operators are able to simulate faults that may not be revealed by pre-existing, non-mutation-based test suites. Furthermore, the effort required to augment the test suites to provide adequate coverage of mutants does not tend to overwhelm the testers. This demonstrates the feasibility of the proposed approach and represents a step towards the practical fault-based testing of AO programs.

*Corresponding author
Email addresses: ferrari@icmc.usp.br (Fabiano Cutigi Ferrari), marash@comp.lancs.ac.uk (Awais Rashid), jcmaldon@icmc.usp.br (José Carlos Maldonado)

Keywords: software testing, mutation testing, aspect-oriented programming, testing aspect-oriented programs, test evaluation

1. Introduction

Aspect-Oriented Programming (AOP) [1] is a contemporary software development technique that aims to tackle modularisation issues faced by traditional approaches such as object-oriented and procedural programming. It has introduced the *aspect* as a new conceptual implementation unit that ideally encapsulates all behaviour required to realise the so-called *crosscutting concerns*. Such a concern may regard either a stakeholder's need or a non-functional requirement that appears either interwoven with other concerns or requirements, or scattered over several modules in the software. Once the aspects are implemented, they are combined with the other system modules – the *base modules* or *base code* – in order to produce the final, executable system. Classical examples of such crosscutting concerns are logging, exception handling, concurrency and certain design patterns.

The benefits that are likely to be achieved in aspect-oriented (AO) software comprise, for instance, enhanced maintainability [2, 3], augmented reusability [4] and facilitated software evolution [5]. In spite of such benefits, new challenges with respect to (w.r.t.) testing of AO software (or simply *AO testing*) have also been introduced [6, 7]. Recent empirical evaluations have indeed shown that AO implementations may have some drawbacks such as undesired effects on the event on changes [8, 9] and unintended flow of exceptions across the system and its boundaries [10]. That is, AOP together with its implementation mechanisms represent new potential sources of software faults that should be taken into consideration by existing and upcoming AO testing approaches.

In this context, mutation testing – originally named Mutant Analysis [11] – represents a useful mean to explore potential faulty scenarios within the AOP context. Mutation testing relies on recurring mistakes made by programmers and develops upon well-characterised faults which are simulated through mutation operators [11]. Furthermore, mutation testing has been shown to be an important tool for the assessment of other testing approaches within the experimental Software Engineering context [12, 13]. In spite of that, recent surveys [14, 15] reveal that mutation testing has not been properly investigated in the context of AO software. To date, the few approaches

to support mutation testing of AO programs are either not comprehensive enough w.r.t. the range of mechanisms introduced by AOP or still lack proper automated support and assessment studies.

This paper presents a comprehensive approach for mutation testing of AO programs. It tackles the aforementioned limitations by describing a set of mutation operators [16] for programs written in AspectJ, which happens to be the most investigated and used AOP language [17]. The design of the mutation operators is based on the characterisation of possible faults in AO software, all summarised in a fault taxonomy [16, 18]. We also present a tool that automates the application of the mutation operators and provides adequate support for the mutation testing of AspectJ programs [19].

We also perform two case studies in order to evaluate the proposed mutation testing approach. The goal of the first case study is to compare the mutant-related coverage provided by adequate test suites w.r.t. widely used functional-based test derivation criteria. Besides that, it measures the effort required to evolve such test suites to fully cover the generated mutants. The second case study, on the other hand, aims to estimate the cost for applying the mutation operators in larger AO systems. The estimation is based on both the generated set of mutants and the results of the first case study.

The results show that the mutation operators are able to simulate faults that may not be revealed by pre-existing, non-mutation-based test suites. Furthermore, the effort required to augment the test suites to provide adequate coverage of mutants does not tend to overwhelm the testers. This demonstrates the feasibility of the proposed approach and represents a contribution towards the practical application of fault-based testing to AO programs.

The remainder of this paper is organised as follows: Section 2 presents the background for the research. Section 3 describes an example of an AO application that is used throughout the paper. Section 4 summarises the fault taxonomy for AO software upon which we proposed our mutation testing approach. The set of mutation operators that underlies the approach is described in Section 5. Section 6 brings an overview of the provided tool support. The two case studies are described in Sections 7 and 8. A summary of the limitations of the conducted studies and the related research are presented in Sections 9 and 10, respectively. Finally, Section 11 brings our conclusions and future work.

2. Background

2.1. Mutation testing

The basic idea behind mutation testing [11] consists in creating several versions of the original program (i.e. the program under test), each one containing a simple fault. These modified versions are called *mutants* and are all expected to behave differently from the original program. In this way, any mutant that is executed against the test data should 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, a fault in P is 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 $P(t) \neq S(t), (t \in T)$ [20]. In other words, t shows that the original program does not conform to the specification.

In the second case, mutation testing evaluates how sensitive T is in order to identify the faults simulated by mutants. In this case, the quality of T is measured by the achieved *mutation score* (MS), which shows the rate of mutants that have been killed by T w.r.t. to the total of non-equivalent mutants [20].

We next introduce the basic concepts of Aspect-Oriented Programming and the AspectJ language. Then, in the subsequent sections, we investigate mutation testing in the context of AO programs, starting from the definition of a fault taxonomy for AO software, then moving to the definition of mutation operators, tool support and practical evaluation.

2.2. Aspect-Oriented Programming and AspectJ

Aspect-Oriented Programming (AOP) [1] is strongly based on the idea of *separation of concerns* (SoC), which claims that computer systems are

better developed if their several concerns are specified and implemented separately [21]. Concerns, in general, can address both functional requirements (e.g. business rules) and non-functional requirements (e.g. synchronisation or transaction management) [22]. They can be defined as *"anything a stakeholder may want to consider as a conceptual unit, including features, non-functional requirements, and design idioms"* [23].

In the context of AOP, a concern is typically handled as a coarse-grained feature that can be modularised within well-defined implementation units. They are split into two categories: *non-crosscutting concerns* and *crosscutting concerns* (or simply *cc-concerns*). The non-crosscutting concerns compose the *base code* of an application and comprise the set of functionalities that can be modularised within conventional implementation units (e.g. classes and data structures). Cc-concerns, on the other hand, cannot be properly modularised within conventional units [1]. As a consequence, in traditional programming approaches such as procedural and object-oriented programming (OOP), code that realises a cc-concern usually appears scattered over several modules and/or tangled with other concern-specific code, thus hardening software evolution and maintenance.

To deal with cc-concerns, AOP has introduced the *aspects*. An aspect can be either a conceptual programming unit represented as an ordinary class (as in industry-strength AOP frameworks like Spring AOP [24] and JBoss AOP [25]) or a concrete, specific unit named *aspect* (as in widely investigated languages such as AspectJ [17] and CaesarJ [26]). Once both aspects and base code are developed, they are combined together through a process called *weaving* [1] in order to produce the final, executable system.

Dynamic crosscutting: 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 of implicitly modifying the behaviour of a program at specific points during its execution. This is known as *dynamic crosscutting* [17]. Each of the points at which aspectual behaviour is activated is called *join point* (JP). They are characterised as well-defined points in a program execution in which aspects can insert additional functionalities as well as replacing the existing ones.

A set of JPs is identified by means of a *pointcut descriptor* (PCD), or simply *pointcut* [17]. A PCD is typically a language-based matching expression that identifies a set of JPs that share some common characteristic (e.g. based on properties or naming conventions). This selection ability is called

quantification and is a fundamental concept of AOP [27].

Once a JP is identified using a PCD during the program execution, behaviour that is encapsulated within method-like constructs named *advices* starts to run. Advices can be of different types depending on the supporting technology. They can generally be defined to run at three different moments when a JP is reached: *before*, *after* or in place of (*around*) it, as implemented in AspectJ [17].

Static crosscutting: AspectJ can also perform structural modifications of modules that compose the base code. This is known as *static crosscutting* [28]. These modifications are achieved by the so-called *intertype declarations* (ITDs) or simply *introductions* [28]. Examples of intertype declarations are the introduction of a new member (e.g. an attribute or a method) into a base module or a change in the class' inheritance (e.g. a newly implemented interface).

Other static modifications result from the use of AspectJ `declare`-like expressions [28]. Examples of such expressions are `declare precedence`, which alters the execution order of aspects that share common JPs, and `declare soft`, which specifies that an exception, if thrown at a JP, is converted to an unchecked exception.

The next section introduces an example of an AO application which is going to be used in the remaining of this paper. It shows how the main AOP concepts and programming constructs are realised in AspectJ.

3. Running example: An AO banking system

This section describes an example application we are going to use along this paper. The example was first presented by Laddad in his book about AOP with AspectJ [29]. It consists in a banking application – hereafter called BankingSystem – that includes aspects to manage business rules and logging-related functionalities.

The class diagram of the system is depicted in Figure 1. We used the ≪`aspect`≫ stereotype to represent that aspects in the system and ≪`crosscuts`≫ to represent the aspect-class dependencies. Such dependencies indicate which base modules are affected by aspects.

The `LogInsufficientBalanceException` aspect is responsible for logging exceptions that might raise during account debit operations. The `IndentedLogging` abstract aspect defines generic logging behaviour (e.g. indentation) that can be made concrete by aspects that extend it. An example

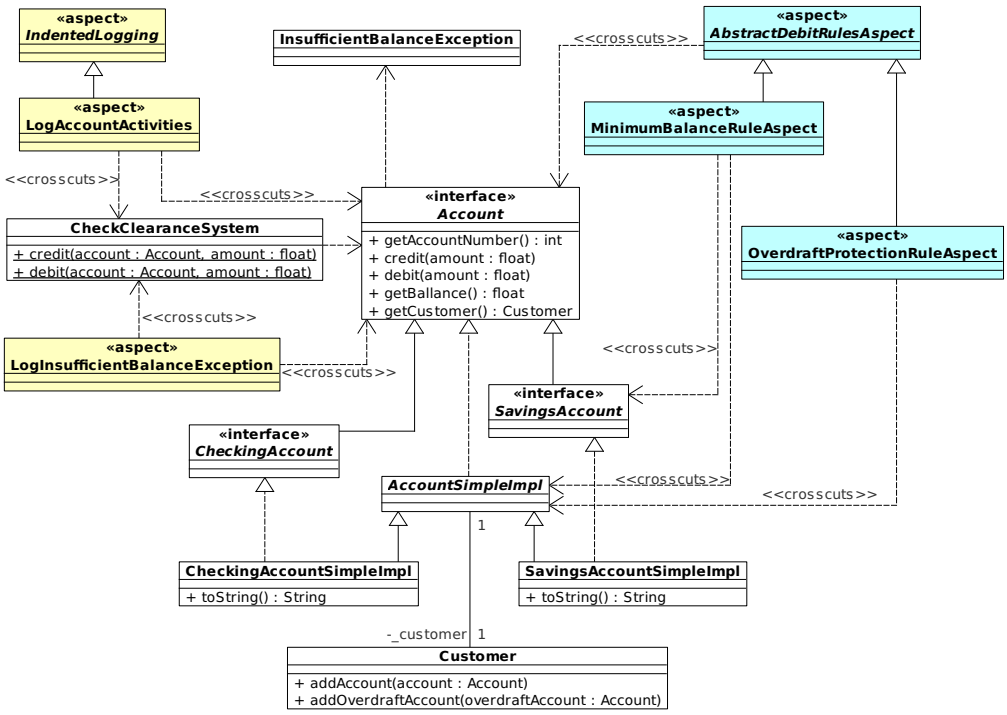


Figure 1: BankingSystem class diagram.

is the `LogAccountActivities` aspect, which logs information about the execution of debit and credit operations. Basically, `LogAccountActivities` displays the account balance before and after each requested operation.

The business rules are realised by the `AbstractDebitRulesAspect`, `MinimumBalanceRuleAspect` and `OverdraftProtectionRuleAspect` aspects. `AbstractDebitRulesAspect` defines a PCD that matches executions of account debit operations. This aspect is extended by `MinimumBalanceRuleAspect`, which implements a rule that enforces a minimum balance for savings accounts. The `OverdraftProtectionRuleAspect` aspect also extends `AbstractDebitRulesAspect` and performs overdraft checkings. If there is a debit request for a specific checking account and such account has no sufficient funds, `OverdraftProtectionRuleAspect` will look for funds in other accounts registered for the current customer.

Figures 2, 3 and 4 display the AspectJ implementation of two logging and a business rule aspects which compose the BankingSystem application.

They include representatives of the main AOP constructs. For instance, PCD definitions can bee seen at Figures 2 (line 10[1]), 3 (lines 5–7 and 9[2]) and 4 (lines 9–10[3]), while advice are found in lines 10–13, 11–17 and 12–17 of the three figures, respectively. Furthermore, we can observe that the `MinimumBalanceRuleAspect` aspect introduces the `getAvailableBalance` method into the `SavingsAccount` class (lines 5–7 in Figure 4). Note that these aspect implementations are also used in the Section 5 to exemplify the use of some mutation operators.

```
1  public abstract aspect IndentedLogging {
2     protected int _indentationLevel = 0;
3
4     protected abstract pointcut loggedOperations();
5
6     before() : loggedOperations() { _indentationLevel++; }
7
8     after() : loggedOperations() { _indentationLevel--; }
9
10    before() : call(* java.io.PrintStream.println(..)) && within(IndentedLogging+) {
11       for (int i = 0, spaces = _indentationLevel * 4; i < spaces; ++i)
12          System.out.print(" ");
13    }
14 }
```

Figure 2: The `IndentedLogging` aspect.

4. AOP-specific fault types

This section investigates types of faults that are specific to AO software. We start defining a set of conditions that must be satisfied in order to produce effective mutants of AO programs (Section 4.1). Such conditions aim to ensure that a fault introduced by a mutation operator can be executed and its effect can be noticed after the program execution [30]. In the sequence, Section 4.2 presents a fault taxonomy for AO software. The taxonomy was developed in terms of researchers' expertise [16] as well as practical observations of AO programs [18].

4.1. Reachability, necessity and sufficiency conditions for AOP-specific faults

The design of effective mutation operators must take into consideration three fundamental conditions: *reachability*, *necessity* and *sufficiency* [30, 31].

[1]This is an example of an anonymous, non-reusable PCD allowed in AspectJ.

[2]It realises the `loggedOperations` abstract PCD defined within `IndentedLogging`.

[3]It realises the `debitExecution` abstract PCD defined within the `AbstractDebitRulesAspect` aspect, which was not listed due to space limitations.

```
1  public aspect LogAccountActivities extends IndentedLogging {
2
3     declare precedence : LogAccountActivities, *;
4
5     pointcut accountActivity(Account account, float amount) :
6        ((execution(void Account.credit(float)) || execution(void Account.debit(float
             )) ) && this(account) && args(amount))
7        || (execution(void CheckClearanceSystem.*(Account, float)) && args(account,
              amount));
8
9     protected pointcut loggedOperations() : accountActivity(Account, float);
10
11    void around(Account account, float amount) : accountActivity(account, amount) {
12       try {
13          System.out.println("[" + thisJoinPointStaticPart.getSignature().
14              toShortString() + "] " + account + " " + amount);
15          System.out.println("Before: " + account.getBalance());
16          proceed(account, amount);
17       } finally { System.out.println("After: " + account.getBalance()); }
18    }
19 }
```

Figure 3: The `LogAccountActivities` aspect.

```
1  public aspect MinimumBalanceRuleAspect extends AbstractDebitRulesAspect{
2
3     private static final float MINIMUM_BALANCE_REQD = 25;
4
5     public float SavingsAccount.getAvailableBalance() {
6        return getBalance() - MINIMUM_BALANCE_REQD;
7     }
8
9     pointcut savingsDebitExecution(Account account, float withdrawalAmount) :
10       debitExecution(account, withdrawalAmount) && this(SavingsAccount);
11
12    before(Account account, float withdrawalAmount) throws
           InsufficientBalanceException:
13       savingsDebitExecution(account, withdrawalAmount) {
14
15       if (account.getAvailableBalance() < withdrawalAmount)
16          throw new InsufficientBalanceException("Minimum balance condition not met"
                );
17    }
18 }
```

Figure 4: The `MinimumBalanceRuleAspect` aspect.

Given an original product P and a mutant m of P which contains a fault f: (i) the *reachability* condition requires that f must be reachable and executable; (ii) the *necessity* condition requires that the state of m must be incorrect after the execution of f when compared to the execution of P at the same point (i.e. the original statement); finally, (iii) the *sufficiency* condition requires that the difference between the states of P and m after the execution of the mutated portion of code must propagate to the end of the execution of P and m.

Defining and checking these conditions for AO programs slightly differs

from programs developed in traditional approaches such as OO and procedural programming (hereafter called *conventional programs*). Mutation operators are typically applied to executable statements in conventional programs (e.g. a variable definition, an object instantiation or a language-specific statement [32, 33, 34]). Therefore, identifying the modified parts of the code is straightforward in order to collect and check information at runtime or even through symbolic evaluation [35].

However, mutations that are specific to AO programs can be performed, for example, in PCDs, ITDs and `declare`-like expressions, which consist in "atypical" executable entities. We can consider that a PCD is "executed" at weaving time since it is responsible for identifying the JPs at which the base code is advised by aspects. Similarly, mutations of ITDs and `declare`-like expressions impact the static structure of the system. Differently from `declare`-like expressions, ITD-specific faults can be mostly detected at compilation time, as discussed in our previous research [16, 18].

Given the aforementioned particularities of AO programs, we following define the reachability, necessity and sufficiency conditions for the three main groups of AOP constructs, namely: (i) PCDs; (ii) ITDs and `declare`-like expressions; and (iii) advices:

PCD-related faults: Similarly to Bækken [36], we define the reachability and necessity conditions for a faulty PCD as the difference between the sets of intended and actually selected JPs after the aspects are woven into the base code. The activation (or non-activation) of an advice at an unintended (or intended) JP that results in unexpected output at the end of the program execution satisfies the sufficiency condition.

ITD- and `declare`-like expression-related faults: For incorrect ITDs (e.g. an incorrect method overriding which resulted from a method introduction), the original definitions for the three conditions [30, 31] are applicable: the fault can be reached (e.g. the overriding method can be executed) and can produce an incorrect state that can propagate until the end of the system execution. Similarly, for `declare`-like expressions the necessity condition may be seen as a side-effect of faulty elements. For example, an incorrect aspect precedence may lead to incorrect order of advice execution, with in turn may lead to observable failures at the end of the execution.

Advice-related faults: Faults in advice may occur either in the advice body or in its signature. The former occurrence consists in typical faults

that can also be present method bodies in OO programs. The latter, on the other hand, as AOP-specific so that we define the three conditions as follows: the fault is reachable if there is at least one feasible path in the program execution which leads to the advice activation; the advice execution itself satisfies the necessity condition, since at this point unexpected control flow would be running; and sufficiency is satisfied when an incorrect state due to the execution of the advice propagates until the end of the system execution.

For all cases described above, according to Offutt and Pan's definition [30], a test case is said to be *effective* w.r.t. killing a mutant if and only if it shows that the execution of the affected code results in abnormal behaviour. In other words, effective test cases have the ability to show that the introduced fault satisfies the reachability, necessity and sufficiency conditions.

4.2. AOP-specific fault taxonomy

We herein present the fault taxonomy for AO software we defined and further refined in our previous research [16, 18]. A preliminary quantitative evaluation showed that the taxonomy was comprehensive enough in order to classify faults that have been documented from real-world AO systems [18].

Tables 1–4 summarise the list all fault types and gives a short description for each of them. The fault types are distributed across four main categories which were defined based on the main elements involved in an AO software, namely: (1) pointcut expressions; (2) ITDs and `declare`-like expressions; (3) advice definitions and implementations; and (4) the base program. More details and generalisations to mainstream AO technologies can be found elsewhere [16, 18].

5. Mutation operators for AspectJ programs

This section introduces a set of mutation operators for AspectJ programs. The design of the operators was based on the set of AO fault types presented in the previous section [16]. The syntactic changes modelled by the operators aim to reproduce mistakes that are likely to be made by programmers. With minor adaptations, they may become applicable to AOP languages that follow the AspectJ implementation model (e.g. CaesarJ [26], and AspectC++ [37]).

Table 1: Faults related to PCDs – adapted from Ferrari et al. [18].

ID	Description
F1.1	Selection of a superset of intended JPs.
F1.2	Selection of a subset of intended JPs.
F1.3	Selection of a wrong set of JPs, which includes both intended and unintended items.
F1.4	Selection of a wrong set of JPs, which includes only unintended items.
F1.5	Incorrect use of a primitive PCD[1].
F1.6	Incorrect PCD composition rules.
F1.7	Incorrect JP matching based on exception throwing patterns.
F1.8	Incorrect JP matching based on dynamic circumstances.

[1] Primitive PCDs are predefined PCDs available in AOP languages (e.g. in AspectJ [28]).

Table 2: Faults related to ITDs and declare-like expressions – adapted from Ferrari et al. [18].

ID	Description
F2.1	Improper method introduction, resulting in unanticipated method overriding or not resulting in anticipated method overriding.
F2.2	Introduction of a method into an incorrect class.
F2.3	Incorrect change in class hierarchy through parent declaration clauses V (e.g. `declare parents` statements), resulting in unintended inherited behaviour for a given class.
F2.4	Incorrect method introduction, resulting in unexpected method overriding.
F2.5	Omitted declared interface or introduced interface which breaks object identity.
F2.6	Incorrect changes in exception-dependent control flow, resulting from aspect-class interactions or from clauses that alter exception severity.
F2.7	Incorrect or omitted aspect precedence expression.
F2.8	Incorrect aspect instantiation rules and deployment, resulting in unintended aspect instances.
F2.9	Incorrect policy enforcement rules supported by warning and error declarations.

Table 3: Faults related to advices – adapted from Ferrari et al. [18].

ID	Description
F3.1	Incorrect advice type specification.
F3.2	Incorrect control or data flow due to incorrect aspect-class interactions.
F3.3	Incorrect advice logic, resulting in invariants violations or failures to establish expected postconditions.
F3.4	Infinite loops resulting from interactions among pieces of advice.
F3.5	Incorrect access to JP static information.
F3.6	Advice bound to incorrect PCD.

Tables 5–7 summarise the designed operators. They are organised into three groups, namely G1, G2 and G3. The groups correspond to the first

Table 4: Faults related to the base program – adapted from Ferrari et al. [18].

ID	Description
F4.1	The base program does not offer required JPs in which one or more foreign aspects were designed to be applied.
F4.2	The software evolution causes PCDs to break.
F4.3	Other problems related do base programs such as inconsistent refactoring or duplicated crosscutting code.

three groups of fault types included in the taxonomy presented in Section 4. Note that for base program-related faults (i.e. group F4 – see Table 4), existing mutation operators are applicable (e.g. for unit level [32] and class level [34]). The next sections describe the groups, providing an overview of each operator and sound examples of their application. The examples are based on the Banking System early introduced in Section 3.

Table 5: Mutation operators for PCDs – adapted from Ferrari et al. [16].

Oper.	Description/Consequences
PWSR	PCD weakening by replacing a type with its immediate supertype in PCDs.
PWIW	PCD weakening by inserting wildcards into it.
PWAR	PCD weakening by removing annotation tags from type, field, method and constructor patterns.
PSSR	PCD strengthening by replacing a type with its immediate subtype.
PSWR	PCD strengthening by removing wildcards from it.
PSDR	PCD strengthening by removing "`declare @`" statements from the aspect code.
POPL	PCD weakening or strengthening by modifying parameter lists of primitive PCDs.
POAC	PCD weakening or strengthening by modifying "`after [retuning| throwing]`" advice clauses.
POEC	PCD weakening or strengthening modifying exception throwing clauses.
PCTT	PCD changing by replacing "`this`" PCDs with "`target`" ones and vice versa.
PCCE	Context changing by switching "`call/execution/initialization/ preinitialization`" PCDs.
PCGS	PCD changing by replacing "`get`" PCDs with "`set`" ones and vice versa.
PCCR	PCD changing by replacing individual parts of a PCD composition.
PCLO	PCD changing by varying logical operators present in type and PCD compositions.
PCCC	PCD changing by replacing "`cflow`" PCDs with "`cflowbelow`" ones and vice versa.

5.1. Group G1: operators for PCDs

This is the largest group which includes 15 operators which model faults related to PCDs. Such faults usually result in incorrect JP matchings or

Table 6: Mutation operators for declare-like expressions – adapted from Ferrari et al. [16].

Oper.	Description/Consequences
DAPC	Aspect precedence changing by alternating the order of aspects involved in `declare precedence` expressions.
DAPO	Arbitrary aspect precedence by removing "`declare precedence`" expressions.
DSSR	Unintended exception handling by removing "`declare soft`" expressions.
DEWC	Unintended control flow execution by changing "`declare error/warning`" expressions.
DAIC	Unintended aspect instantiation by changing "`perthis/pertarget/percflow/percflowbelow`" deployment clauses.

Table 7: Mutation operators for advices – adapted from Ferrari et al. [16].

Oper.	Description/Consequences
ABAR	Advice kind changing by replacing a "`before`" clause with an "`after [retuning|throwing]`" one and vice versa.
APSR	Advice logic changing by removing invocations to "`proceed`" statement.
APER	Advice logic changing by removing guard conditions which surround "`proceed`" statements.
AJSC	Static information source changing by replacing a "`thisJoinPoint-StaticPart`" reference with a "`thisEnclosingJoinPointStaticPart`" one and vice versa.
ABHA	Behaviour hindering by removing implemented advices.
ABPR	Changing PCD-advice binding by replacing PCDs which are bound to advices.

undue execution contexts. This group, shown in Table 5, is divided into four categories, according to the results obtained from the application of the respective operators:

(i) PCD weakening operators. This category includes the PWSR, PWIW and PWAR operators. Mutants produced by these operators possibly *increase* the number of selected JPs if compared to the original PCD. Figure 5 shows an example of how the PWIW operator inserts the "`*`" and "`+`" wildcards into the original anonymous PCD extracted from the `IndentedLogging` aspect (see Figure 2, line 10). Note that for this example and the others presented in the sequence, the original code is listed in the first line while the remaining lines – starting with "`>`" – show its mutants.

(ii) PCD strengthening operators. This category includes the PSSR, PSWR and PSDR operators. Mutants produced by these operators possibly *decrease* the number of selected JPs if compared to the original PCD. Figure 6 shows an example of how the PSWR operator removes the "`+`" wildcard from the

```
   before() : call(* java.io.PrintStream.println(..)) && within(IndentedLogging+) {

>  before() : call(* *.io.PrintStream.println(..)) && within(IndentedLogging+) {
>  before() : call(* java.*.PrintStream.println(..)) && within(IndentedLogging+) {
>  before() : call(* java.io.*.println(..)) && within(IndentedLogging+) {
>  before() : call(* java.io.PrintStream.*(..)) && within(IndentedLogging+) {
>  before() : call(* java.io.PrintStream.println*(..)) && within(IndentedLogging+) {
>  before() : call(* java.io.PrintStream.*println(..)) && within(IndentedLogging+) {
>  before() : call(* java.io.PrintStream*.println(..)) && within(IndentedLogging+) {
>  before() : call(* java.io.*PrintStream.println(..)) && within(IndentedLogging+) {
>  before() : call(* java.io.PrintStream+.println(..)) && within(IndentedLogging+) {
>  before() : call(* java.io*.PrintStream.println(..)) && within(IndentedLogging+) {
>  before() : call(* java.*io.PrintStream.println(..)) && within(IndentedLogging+) {
  ... // and so on
```

Figure 5: Examples of mutants produced by the PWIW operator.

original PCD. Note that this PCD is the same as used in the example presented in Figure 5.

```
   before() : call(* java.io.PrintStream.println(..)) && within(IndentedLogging+) {

>  before() : call(* java.io.PrintStream.println(..)) && within(IndentedLogging) {
```

Figure 6: Example of mutant produced by the PSWR operator.

(iii) PCD weakening or strengthening operators. This category includes the POPL, POAC and POEC operators. They produce mutants that may either *increase* or *decrease* the number of selected JPs. Figure 7 exemplifies the application of the POAC operator to modify `after` advice clauses. The original advice definition was extracted from the `IndentedLogging` aspect (see Figure 2, line 8).

```
   after() : loggedOperations() { _indentationLevel--; }

>  after() returning : loggedOperations() { _indentationLevel--; }
>  after() throwing : loggedOperations() { _indentationLevel--; }
```

Figure 7: Examples of mutants produced by the POAC operator.

(iv) PCD changing operators. Operators from this group perform a variety of changes in PCDs, as shown in Table 5. As a result, we have: (i) partial or complete changes in the number of selected JPs with the PCTT, PCGS, PCCR and PCLO operators; (ii) changes in execution context with the PCCE

operator; and (iii) changes in the number of advice executions with the PCCC operator.

Figure 8 shows some mutants that can be generated by the PCLO operator. The original code is part of a PCD from the `LogAccountActivities` aspect (see Figure 3, line 6). In this example, the changes consists in replacing the "`||`" logical operator with "`&&`", and adding the "`!`" operator to each part of the composition. Note that similar rules can be applied to the remaining parts of the PCD as well as to the whole expression.

```
    ((execution(void Account.credit(float)) || execution(void Account.debit(float))
        ) && this(account) && args(amount))

>   ((execution(void Account.credit(float)) && execution(void Account.debit(float))
     ) && this(account) && args(amount))
>   ((!execution(void Account.credit(float)) || execution(void Account.debit(float))
      ) && this(account) && args(amount))
>   ((execution(void Account.credit(float)) || !execution(void Account.debit(float))
      ) && this(account) && args(amount))
```

Figure 8: Examples of mutants produced by the PCLO operator.

5.2. Group G2: operators for ITDs and `declare`-like expressions

This group contains five operators that model faults related to AspectJ `declare`-like expressions. These operators are listed in Table 6. Faults that are modelled by them may lead to unintended control flow executions, hence may result in erroneous object/aspect state. For example, the DAPC operator modifies `declare precedence` expressions, swapping amongst all aspects involved in the precedence definition. The DAPO operator is applied to omit `declare precedence` expressions. Other `declare`-like expressions targeted by operators of G2 are: `declare soft` (operator DSSR), `declare error` and `declare warning` (operator DEWC), and aspect instantiation rules (operator DAIC).

Figure 9 exemplifies the application of the DAPC operator that modifies `declare precedence` statements. The original advice definition was extracted from the `LogAccountActivities` aspect (see Figure 3, line 3).

```
    declare precedence : LogAccountActivities, *;

>   declare precedence : *, LogAccountActivities;
```

Figure 9: Example of mutant produced by the DAPC operator.

5.3. Group G3: operators for advices

Six operators related to advice definition and implementation compose this group. They are described in Table 7. Faults modelled by them comprise incorrect advice kind (ABAR operator), incorrect advice logic (APSR, APER and AJSC operators) and incorrect advice execution (ABHA and ABPR operators). Figure 10 exemplifies the application of the ABAR operator that replaces the kind of advices. The original advice definition was extracted from the `MinimumBalanceRuleAspect` aspect (see Figure 4, line 12).

```
    before(Account account, float withdrawalAmount) throws
        InsufficientBalanceException:

>   after(Account account, float withdrawalAmount) throws
      InsufficientBalanceException:
>   after(Account account, float withdrawalAmount) throwing throws
      InsufficientBalanceException:
>   after(Account account, float withdrawalAmount) returning throws
      InsufficientBalanceException:
```

Figure 10: Example of mutant produced by the ABAR operator.

5.4. Analysing AO mutants

Analysing mutants of AO programs (hereafter called *AO mutants*) differs from analysing mutants of conventional programs (or simply *conventional mutants*). For example, while the analysis of conventional mutants is typically unit-centred, i.e. the task is concentrated on the mutated statement and perhaps on its surrounding statements, detecting equivalent AO mutants may require a broader analysis of the woven code[4]. This is due to the quantification property [27] that is inherent to AOP constructs such as PCDs and `declare`-like expressions.

For mutations that affect the quantification of JPs, a possible approach to identify equivalent mutants is through the analysis of *JP static shadows*. A JP static shadow consists of the set of implicit calls to advice-methods, i.e. bytecode methods that result from the compilation of advices [38]. Such calls are inserted into the base code during the weaving process. Nevertheless, the advice execution itself may only be resolved at runtime, that is, not all

[4]The inter-class mutation operators for Java programs proposed by Ma et al. [34] pose similar challenge, given that a mutation that affects a member inheritance or a polymorphic type also requires broad analysis of the resulting compiled application.

JP included in a JP static shadow will in fact trigger the advice execution. Furthermore, given that named PCDs are eligible to be reused by other aspects, a complete JP matching analysis should be performed in order to decide if a PCD-related mutant is equivalent or not.

6. Tool support

We implemented a tool called *Proteum/AJ* to support the application of the mutation operators presented in the previous section [19]. The tool leverages previous knowledge on developing *Proteum* (**Pro**gram **Te**sting **U**sing **M**utants) [39], a family of tools for mutation testing developed by the Software Engineering group at the University of São Paulo, Brazil.

Proteum/AJ supports the four main steps of mutation testing, as originally described by DeMillo et al. [11]: (i) the original program is executed on the current test set and test results are stored; (ii) the mutants are created based on a mutation operator selection that may evolve in new test cycle iterations; (iii) the mutants can be executed all at once or individually, as well as the test set can be augmented or reduced based on specific strategies; and (iv) the test results are evaluated so that mutants may be set as dead or equivalent, or mutants may remain alive. Along this section we provide an overview of *Proteum/AJ*'s structure and functionalities. In Sections 7 and 8 we present the results of two case studies that have been performed with *Proteum/AJ*'s support.

Proteum/AJ is composed by a set of functional modules that automate the mutation testing steps. Such modules are invoked through parameterised command lines. Figure 11 displays an example of a command line that creates a test project (first 2 lines) together with some logging messages (remaining lines). In this example, the user ferrari creates a test project named exampleBankingSystem, which includes the PWIW, DAPC and APSR operators. banking.zip is the compressed file that contains the application to be tested.

The full mutation testing process supported by *Proteum/AJ* is next described and depicted in Figure 12, which shows the execution sequence of the main modules and the inputs/outputs of each of them. Command lines scripts, as the one presented in Figure 11, can be executed to activate the other *Proteum/AJ*'s functionalities represented in Figure 12.

Pre-processing the original application: Initially, *Proteum/AJ* receives a compressed file which includes the application and possibly an ini-

```
java br.usp.icmc.labes.amt.ui.cmdtool.CreateProject -u ferrari -a
         "/home/ferrari/tmp/banking.zip" -o "PWIW DAPC APSR" -p exampleBankingSystem


Timer started...

... Creating a new test project. Only the required parameters will be considered...
... Creating the project...
... Test project successfully created.
... Mutation operators successfully inserted.

uploadFile:
     [echo] Uploading the target application...
     [copy] Copying 1 file to /home/ferrari/tmp/mutation/executionFiles/uploads

... some other messages

BUILD SUCCESSFUL
Total time: 4 seconds
... Target application successfully decompressed and compiled...
... Test project successfully created.

Timer stopped...Total spent time: 00:00:07:04
```

Figure 11: Example of a shell command for creating a test project.

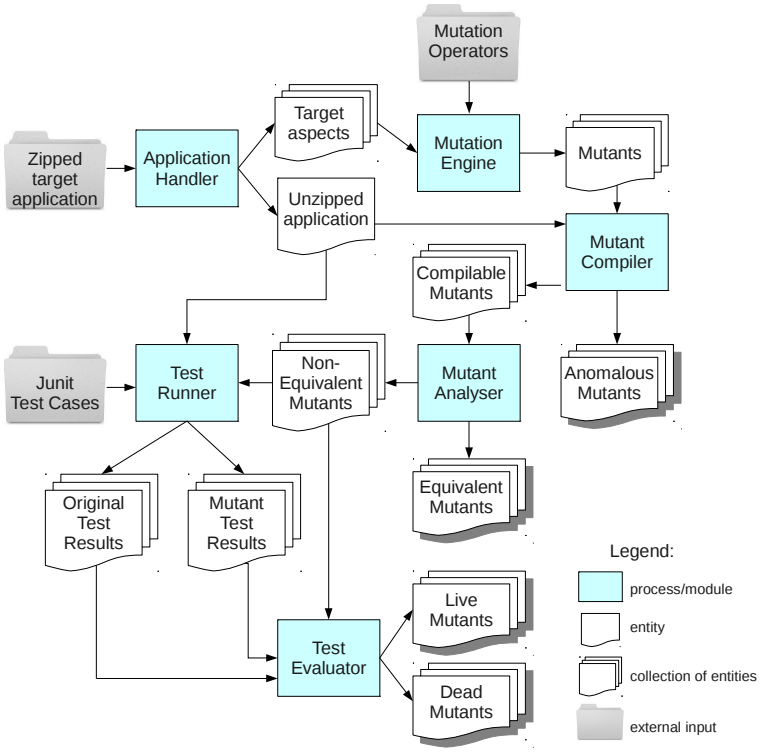


Figure 12: *Proteum/AJ*'s execution flow – adapted from Ferrari et al. [19].

tial set of JUnit test cases. The Application Handler module then runs a pre-processing step, whose outputs are the decompressed original application and a list of target aspects. This step creates the test projects in the *Proteum/AJ* database. The Application Handler also compiles the original application through the iajc Ant task [28].

Execution of the original application: The decompressed application is sent to the Test Runner module together with the test case files. The Test Runner executes the application on the available test set by invoking the junit Ant task. The results are stored for further evaluation of mutants.

Generation of mutants: The Mutation Engine includes 24[5] out of 26 mutation operators proposed in Section 5. It receives as input the list of target aspects identified by the Application Handler and the set of mutation operators selected by the tester. It produces the set of mutants that are passed to the Mutant Compiler.

Execution of mutants: The execution of mutants requires a series of steps in *Proteum/AJ*. Initially, each mutant is sent to the Mutant Compiler module, which invokes the AspectJ ajc compiler [28] through the iajc Ant task. The Mutant Compiler detects non-compilable mutants which are classified as *anomalous*. For compilable mutants, the weaving information produced by the compiler is collected at this stage and further used by the Mutant Analyser module. This output is a valuable information that includes details of all matching w.r.t. JPs and static crosscutting between aspects and base code [28]. The weaving output information of the original application and of the mutants is compared in order to decide whether mutants created by a subset of operators from Group G1 are equivalent to the original aspects. Details about the automatic equivalent mutant detection performed by *Proteum/AJ* are not described here due to space limitations although they can be found elsewhere [19].

As for the original application, the Test Runner module runs all compilable, non-equivalent mutants on the current test set using the junit Ant task. The results are stored and sent to the Test Evaluator module. The evaluator contrasts the results with the original results and identifies which mutants should be killed.

[5] Automating the PWSR and PSSR operators depends on class hierarchy analyses based on the Java reflection API and is currently not available in Proteum/AJ.

Analysis of mutants: The Mutant Analyser module creates a set of plain text reports that show the differences between the original and mutated code. These reports show the modified parts of the code for each mutant, hence providing guidance for the tester while analysing the changes. The tester then decides which mutants should be set as equivalent and updates their status through the Mutant Analyser module. This module also computes the mutation score.

7. First case study: Evaluating the usefulness and required effort of the proposed approach

This section describes a study that evaluates the mutation-based testing approach proposed in this paper. The main goal of this study is checking the feasibility of the approach in terms or its usefulness and required effort. The evaluation procedures were planned and conducted in order to answer the two following questions: (i) *do the operators have the ability of simulating faults that cannot be detected by pre-existing, systematically derived test suites?* and (ii) *can the proposed mutation operators be applied at a feasible cost?*

To answer these questions, we selected and thoroughly tested a set of AO applications based on a well-established functional-based testing approach. We evaluated the coverage obtained with this functional-based test data in regard to the generated mutants for each application. This initial phase addressed the first defined question. We then evolved the test sets to make them adequate to cover all non-equivalent mutants of the selected applications, thus addressing our second question. All procedures are described in the sequence, starting from an overview of the evaluated AO applications.

7.1. Target applications

We selected 12 small AO applications upon which we performed our evaluation. All applications were identified from previous papers that describe AO testing approaches and evaluation. A short description of each of them is following presented. A summary of the size-related metrics of the selected applications is shown in Table 8.

The BankingSystem application manages transactions for bank accounts [29]. It has been the running example throughout this paper. Aspects in BankingSystem implement logging, minimum balance control and overdraft operations.

Table 8: Values of metrics for the selected applications (first case study).

Application	LOC[1]	Classes[2]	Aspects	PCDs	Advices	declare
1. BankingSystem	199	9	6	11	7	1
2. Telecom	251	6	3	6	7	1
3. ProdLine	537	8	8	10	10	4
4. FactorialOptimiser	39	1	1	2	3	0
5. MusicOnline	150	7	2	4	3	0
6. VendingMachine	64	1	3	6	6	1
7. PointBoundsChecker	44	1	1	4	4	0
8. StackManager	77	4	3	3	3	0
9. PointShadowManager	66	2	1	3	3	0
10. Math	53	1	1	1	1	0
11. AuthSystem	89	3	2	2	2	0
12. SeqGen	205	8	4	3	3	2
TOTAL	1774	51	35	55	52	9

[1]It considers only real lines of code, excluding comments and blank lines.

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

Telecom is a telephony system simulator which is originally distributed with AspectJ [28]. In Telecom, timing and billing of phone calls are handled by aspects. The version evaluated in this section extends the original implementation in order to support a different type of charging for mobile calls [40].

ProdLine consists in a software product line for graph applications that includes a set of common functionalities of the graph domain [41]. Typical algorithms for graph manipulation are included, e.g. shortest-path between vertices, identification of strongly connected components and cycle checking. Aspects are used in this application to introduce the features selected for a specific SPL instance. Each feature is implemented through one ore more aspects. AspectJ ITDs are intensively used together with a few PCD-advice pairs.

FactorialOptimiser is a math utility application that implements optimised factorial calculation [7]. The calculation is managed by an aspect; if the factorial for a given number has already been calculated, the aspects retrieve it to reduce overhead. Every calculated factorial is cached for reuse purposes.

The MusicOnline application manages an online music store that allows customers to play songs and playlists [42]. A customer needs to pay for each played song or playlist. Aspects in this application manage the customer's accounts and the billing system. Once a customer exceeds his credit limit, his account is suspended until he or she proceeds either a total or a partial

payment.

VendingMachine consists in an application for a vending machine into which the customer inserts coins in order to get drinks [43]. The aspects are responsible for controlling the sales operations (e.g. number of inserted coins and number of available drinks).

PointBoundsChecker is a two-dimension point constraint checker [44]. An aspect checks if the point coordinates conform to a specific range of values. If not, exceptions are raised.

StackManager implements a simple stack that provides the basic push and pop operations, which are supervised by aspects [45]. The aspects avoid the insertion of negative values into the stack, perform audit on the stored elements and count the number of push operations.

PointShadowManager is an application for managing two-dimension point coordinates [6]. An aspect creates and manages shadows for point objects. When a point is created, its shadow has exactly the same coordinates. When a point coordinated is updated, the respective shadow's coordinate is added by a fixed offset.

Math is a math utility application that calculates the probability of successes in a sequence of n independent yes/no experiments (Bernoulli trial), each yielding success with probability p [42]. The aspect logs exponentiation operations, identifying the type of the exponent (integer or real).

AuthSystem is a simplified version of a banking system that requires user authentication before the execution of certain operations like debit and balance retrieval [46]. Furthermore, it monitors amount transfer between accounts by means of atomic transactions. The aspects are responsible for authentications and transaction management.

SeqGen implements a sequence generator of integers and chars values [47]. It includes two aspects that modularise the generation policy (random and Fibonacci sequences) and the logging concerns. Similarly to the ProdLine application, AspectJ ITDs are intensively used in SeqGen.

7.2. Building the initial test set

We applied the Systematic Functional Testing (SFT) criterion [48] to build initial test sets for each selected application. The choice for the SFT criterion was motivated by the significant results reported in previous research [48]: test sets that covered all SFT-derived requirements – hereafter called *SFT-adequate* test sets or simply T_{SFT} – yielded high coverage of mutants generated with conventional mutation operators for C programs [32].

SFT combines two widely used functional-based testing criteria: Equivalence Partitioning and Boundary-Value Analysis [49]. Basically, the main difference between SFT and the other two is that SFT requires two test cases for each equivalence class. In this way, one can avoid coincidental correctness possibly observed for a test input which "masks" a fault that could be uncovered by another test input from the same domain partition [48].

For each of the target applications, we developed a test plan that specifies the equivalence classes and boundaries that should be covered at this initial testing phase. We defined the test requirements for every public operation from the classes that compose the application. Moreover, we defined test requirements for aspectual behaviour according to the description of the systems available in the original papers and reports [6, 7, 29, 40, 41, 42, 43, 44, 45, 46, 47].

Figure 13 and Table 9 illustrate the definition of the equivalence classes and boundary values for a method extracted from the BankingSystem application. The `debit` method belongs to the `AccountSimpleImpl` class and is intended to perform debit operations in bank accounts, having as a constraint the current account balance that should never be lower than zero.

```
1 public void debit(float amount) throws InsufficientBalanceException {
2   if (_balance < amount) {
3      throw new InsufficientBalanceException("Total balance not
           sufficient");
4   }
5   else {
6     _balance = _balance - amount;
7   }
8 }
```

Figure 13: Source code of the `debit` method (`AccountSimpleImpl` class).

Table 9: Equivalence classes and boundary values for the `debit` method.

Input Condition	Valid class	Invalid Class
`amount` parameter	(C1) amount <= current balance	(C2) amount > current balance
Output Condition	**Valid class**	**Invalid Class**
Resulting balance	(o1) balance = previous balance − debited amount	n/a
	(o2) balance = previous balance	n/a
Boundary values		
(C1) amount = current balance		
(C2) amount = current balance + 0.01		

Building SFT-adequate test sets requires designing at least two test cases to cover each of the equivalence classes, as well as at least one test case that covers each of the boundary values. As suggested by Myers et al. [49], tests for valid classes can cover one or more of such classes, whereas individual tests should be created for each invalid class. Considering the test plan for the `debit` method presented in Table 9, a SFT-adequate test set w.r.t. such method requires at least six test cases: four[6] test cases to cover the equivalence classes and two others to cover the boundary values. The results obtained with the execution of the SFT-adequate test sets in this case study are detailed in the sequence.

Table 10 summarises the results achieved during this initial phase. Column 2 lists the number of test requirements derived for each application according to the Equivalence Partitioning criterion, and likewise does column 3 for Boundary-Value Analysis. Columns labelled with $|T_{EB}|$ and $|T_{SFT}|$ list the size of test sets which are adequate w.r.t., respectively, the two aforementioned traditional functional-based criteria and the SFT criterion (i.e. *SFT-adequate* test sets). Column 6 lists the increase percentage w.r.t. the size of the test sets when we evolved T_{EB} to T_{SFT}. Finally, Column 7 shows the number of faults revealed in each application. Such faults are related to either ordinary code (e.g. incorrect implemented logic) or AOP constructs (e.g. incorrect PCD definition) and have been all fixed before we started the mutation testing phase that is next described.

7.3. Applying Mutant Analysis to the target applications

We applied the 24 mutation operators implemented in the *Proteum/AJ* tool to the 12 applications of our case study. The summary of the mutant generation step is displayed in Table 11. Columns 2–4 represent the three groups of mutation operators, namely G1 (related to PCDs), G2 (related to `declare`-like expressions) and G3 (related to advices), together with the associated number of mutants per application. Table 11 also includes the number of equivalent mutants that have been automatically identified by *Proteum/AJ* (column 6), the number of anomalous – i.e. non-compilable – mutants (column 7) and the number of mutants that remained alive (column 8).

[6]Note that a single test case can cover classes C1 and o1, while another test case can cover classes C2 and o2. Therefore, two other test cases are sufficient to fulfil the SFT criterion w.r.t. the equivalence classes.

Table 10: Functional-based test requirements and respective adequate test sets.

Application	Equivalence Classes	Boundaries	$\|T_{EB}\|$	$\|T_{SFT}\|$	$T_{EB} \rightarrow T_{SFT}$	Faults
1. BankingSystem	36	10	34	58	71%	1
2. Telecom	49	2	37	63	70%	6
3. ProdLine	23	6	20	36	80%	0
4. FactorialOptimiser	10	6	15	19	27%	0
5. MusicOnline	43	2	25	46	84%	0
6. VendingMachine	13	5	12	18	50%	1
7. PointBoundsChecker	10	6	9	14	56%	0
8. StackManager	12	3	9	15	67%	2
9. PointShadowManager	12	2	6	12	100%	0
10. Math	26	38	41	53	29%	1
11. AuthSystem	16	6	12	17	42%	1
12. SeqGen	46	15	22	39	77%	0
TOTAL	296	101	242	390	61%	12

Table 11: Mutants generated for the 12 target applications.

Application	Mut. G1	Mut. G2	Mut. G3	Total	Autom. Equiv.	Anom.	Alive
1. BankingSystem	108	2	26	136	68	18	50
2. Telecom	82	2	27	111	46	12	53
3. ProdLine	158	0	41	199	125	16	58
4. FactorialOptimiser	14	0	15	29	8	6	15
5. MusicOnline	47	0	10	57	25	5	27
6. VendingMachine	82	2	29	113	58	8	47
7. PointBoundsChecker	46	0	24	70	32	10	28
8. StackManager	34	0	11	45	24	0	21
9. PointShadowManager	38	0	12	50	25	4	21
10. Math	16	0	4	20	13	0	7
11. AuthSystem	45	0	7	52	28	3	21
12. SeqGen	33	0	7	40	19	3	18
TOTAL	703	6	213	922	471	85	366

Note that *Proteum/AJ* allows the tester to enable or disable the option for automatic detection of equivalent mutants. The mutation operators that are eligible for such automatic procedure are all listed in Section 8 of this paper (see Table 14). They are related to PCDs and are expected to yield the largest mutant set amongst the three groups of operators [16], from which a high percentage represent equivalent ones[7]. Indeed, the figures presented in

[7]High percentages of PCD-related equivalent mutants have been observed by Delamare et al. [50].

Table 11 reveal that nearly 67% of the mutants produced by operators from G1 were automatically detected as equivalent. Besides that, around 9% of mutants are anomalous. We can also observe a very small number of mutants produced by operators from G2, mainly due to the rare use of `declare`-like expressions in the selected applications.

In order to evaluate the mutant coverage yielded by the T_{SFT} test sets and augment such coverage, for each application we performed the following sequence of steps: (1) execution of the live mutants on the respective T_{SFT} test set; (2) calculation of the initial mutation score; (3) manual identification of equivalent mutants; (4) calculation of the intermediate mutation score; (5) design of new test cases to kill the remaining live mutants, producing the mutation-adequate (T_M) test sets (i.e. mutation score of value 1.0).

The obtained results after performing the six steps just described are summarised in Table 12. In this table, columns 2–5 list, respectively, the initial number of live mutants, the number of mutants killed with the execution of T_{SFT}, the remaining number of live mutants after the execution of T_{SFT}, and the initially achieved mutation score. For those applications whose achieved mutation score was lower than one (i.e. at least one mutant was still alive), columns 6 and 7 show the number of equivalent mutants identified by hand and the updated mutation score. Finally, columns 8–10 present, respectively, the number of test cases added to T_{SFT} in order to obtain the T_M test set, the size of T_M and the final mutation score. Note that mutation scores of value 1.0 are not repeated in subsequent columns of the table. The results are analysed in the sequence.

Table 12: Mutation testing results for the 12 target applications.

Application	Initial Alive	Killed by T_{SFT}	Remain Alive	Initial MS	Man. Equiv.	Interm. MS	Added Tests	$\|T_M\|$	Final MS
1.BankingSystem	50	50	0	1.00	–	–	–	58	–
2.Telecom	53	31	22	0.58	10	0.72	4	67	1.00
3.ProdLine	58	58	0	1.00	–	–	–	36	–
4.FactorialOptimiser	15	14	1	0.93	1	1.00	–	19	–
5.MusicOnline	27	22	5	0.81	2	0.88	2	48	1.00
6.VendingMachine	47	23	24	0.49	13	0.68	5	23	1.00
7.PointBoundsChecker	28	28	0	1.00	–	–	–	14	–
8.StackManager	21	21	0	1.00	–	–	–	15	–
9.PointShadowManager	21	13	8	0.62	5	0.81	2	14	1.00
10.Math	7	4	3	0.57	2	0.80	1	54	1.00
11.AuthSystem	21	17	4	0.81	1	0.85	2	19	1.00
12.SeqGen	18	4	14	0.22	8	0.40	3	42	1.00
TOTAL	366	285	81	0.78	42	0.88	19	409	1.00

7.4. Analysis of the results

Comparing SFT with mutation testing: According to the results presented in the previous section, in particular in Table 12, the T_{SFT} test sets have yielded high[8] mutant coverage for five applications: $MS = 1.00$ for BankingSystem, ProdLine, PointBoundsChecker and StackManager, and $MS = 0.93$ for FactorialOptimiser.

The mutation scores yielded by the T_{SFT} test sets can be observed in Figure 14. The chart depicts the achieved mutant coverage at two different moments: before and after the manual identification of equivalent mutants. The column labelled with *Initial* represents the first case, i.e. the achieved coverage right after the execution of the T_{SFT} test sets. The *All Equiv* column represents the coverage after we performed the analysis of live mutants and classified all the equivalent ones. The chart also includes a column to represent the goal, i.e. the full mutant coverage.

These obtained results provide evidence on the relevance of the mutation operators' ability in simulating faults that cannot be easily revealed by existing, non-mutation-based test sets. In this case study, we applied the SFT criterion, which combines widely used functional-based testing criteria and has shown to be strong w.r.t mutant detection rate [48]. However, the *SFT-adequate* test sets were able to achieve high mutation score for only 5 out of 12 evaluated applications.

Effort required to achieve mutation-adequate test sets: In regard to the effort required to obtain the T_M test sets (i.e. test sets that result in mutation scores of 1.0), apart from the FactorialOptimiser application, whose mutation score reached the value of 1.0 after the manual equivalent mutant detection step, the other 7 applications required test set increments. This is depicted in Figure 15, which shows the differences in size of T_{SFT} and T_M test sets for all evaluated applications.

Considering the initial and final test set cardinalities (i.e. $|T_{SFT}|$ and $|T_M|$) (see Tables 10 and 12), in total, 19 test cases were designed to kill the mutants that remained alive after the mutant execution and identification of

[8]Note that the definition of a threshold value for the goodness of the mutation score (e.g. high or low) depends on the criticality of the program under evaluation. Nevertheless, mutation score values above 0.95 have typically been considered high in previous research [51, 52]. In this case study, we use a threshold value of 0.9; therefore, $MS \geq 0.9$ is considered high.

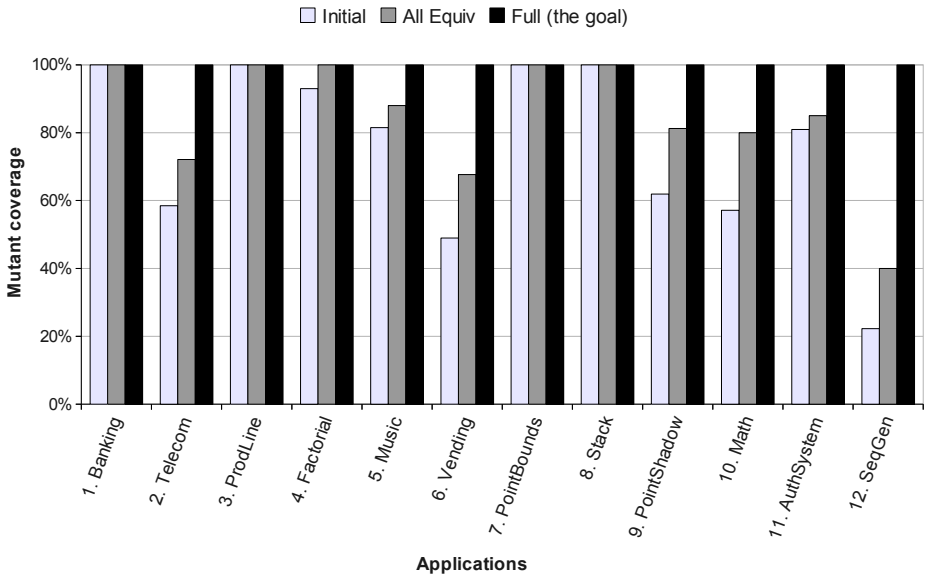


Figure 14: Coverage yielded by SFT test sets.

equivalent mutants steps. It means that, on average, the increase to produce the $|T_M|$ test sets was nearly 5%. From this viewpoint, we can conclude that the application of the proposed mutation operators does not overwhelm the testers while enhancing the existing, systematically derived test sets to achieve high mutant coverage.

7.5. Additional comments on the mutant analysis step

As highlighted in Section 5.4, a simple mutation can have wide impact on the woven application. Consequently, many times analysing a mutant required us to scan several modules of the application in order to realise the real impact of the mutation on the application. However, even though a mutation can impact on the quantification of JPs, the behaviour of the woven application may remain the same.

For example, Figure 16 shows a mutant produced by the PWIW for the MusicOnline application. The mutation consisted in replacing a naming part of the PCD (i.e. the `owed` attribute) with the "`*`" wildcard. This modification results in an additional JP selection in the based code, hence this mutant

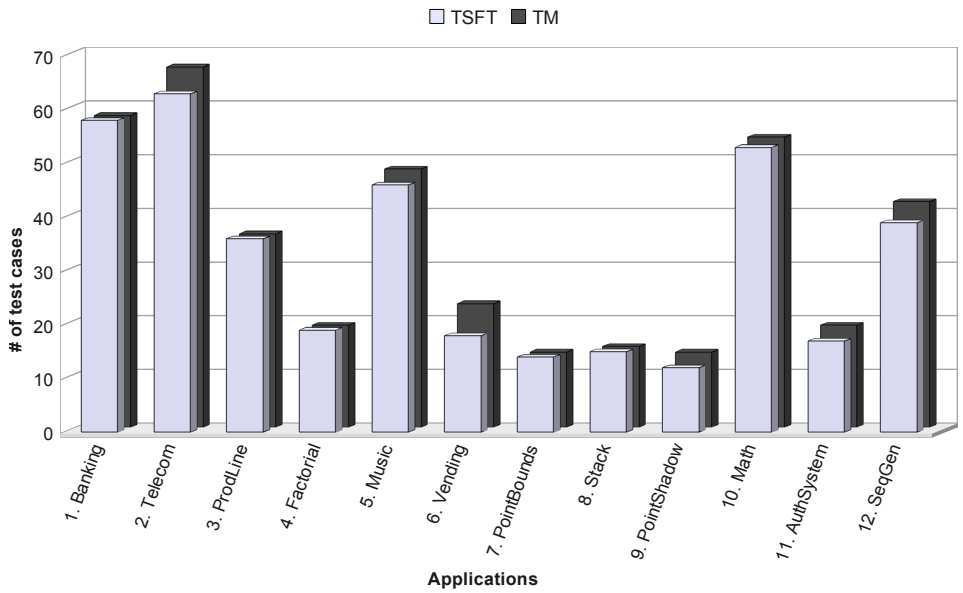


Figure 15: Effort required to derive the T_M test sets.

has not been automatically classified as equivalent by *Proteum/AJ*. Nonetheless, during the mutant analysis step, we noticed that the final behaviour of MusicOnline was not altered so that this mutant was manually classified as equivalent.

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

Figure 16: Example of an equivalent mutant of the MusicOnline application.

Note that the analysis of live mutants depends on the current configuration of the application under test [53]. That is, while a PCD p may be correct w.r.t. to a given base program P_1, p may "misbehave" (i.e. match a different set of JPs) when it is applied into a different base program P_2. This observation also holds for advices as well as for PCD-advice pairs.

8. Second case study: Estimating the cost of the approach with larger systems

This section presents a second case study that aims to estimate the cost of applying the AspectJ mutation operators in systems larger than the ones presented in the previous section. This second case study comprises four medium-sized AO systems from different application domains, and the achieved results are compared to the results presented in Section 7 of this paper. We start by describing the four analysed systems. The results are presented in the sequence.

8.1. Target systems

The four systems we evaluated in this case study are called iBATIS, TollSystemDemonstrator, HealthWatcher and MobileMedia. These systems have already been evaluated within the academic and industrial context [8, 9, 54, 55, 56] and employ mainstream industrial technologies in their implementations. Table 13 lists some size-related metric values for the four systems. Note that these implementations are significantly larger than the applications evaluated in Section 7. This allows us to better estimate the cost of applying the mutation operators, in particular w.r.t. the number of generated mutants, including equivalent and anomalous ones. A short description of each system is following presented.

Table 13: Values of metrics for the selected applications (second case study).

Application	Aprox. KLOC[1]	Classes	Aspects	PCDs	Advices	declare
iBATIS	11	207	41	97	95	69
TollSystemDemonstrator	4	98	25	37	37	6
HealthWatcher	7	137	26	57	47	14
MobileMedia	3	45	22	65	60	26
TOTAL	24	487	114	256	239	115

[1]It considers only real lines of code, excluding comments and blank lines.

iBATIS [57] is a Java-based open source framework for object-relational data mapping. The iBATIS AO versions [9] have some functional and non-functional concerns modularised within aspects (e.g. exception handling, concurrency and type mapping). The TollSystemDemonstrator (TSD) system includes a subset of requirements of a real-world tolling system. It has been developed in the context of the AOSD-Europe Project [54] and contains functional and non-functional concerns implemented within aspects

(e.g. charging variabilities, distribution and logging). HealthWatcher (HW) is a Web-based application that allows citizens to register complaints regarding health issues [8, 56]. Some aspectised concerns in HealthWatcher are distribution, persistence and exception handling. Finally, MobileMedia (MM) [55] is a software product line for mobile devices that allows users to manipulate image files in different mobile devices. In MM, aspects are used to configure the product line instances, enabling the selection of alternative and optional features.

Apart from TollSystemDemonstrator, which has a single release, the other three systems have several releases available for evaluation. In this case study, we selected versions 01, 10 and 06 of iBATIS, HealthWatcher and MobileMedia, respectively. For more information about each of them, the reader may refer to the respective placeholder websites or to previous reports of these systems [8, 9, 54, 55].

8.2. Generating mutants for the target systems

Applying the 24 mutation operators implemented in *Proteum/AJ* resulted in the numbers of mutants presented in Tables 14 and 15. Such tables focus on, respectively, the numbers of equivalent and anomalous mutants produced per operator. The two rightmost columns of the tables show the total number of generated mutants for all systems and the respective percentages of equivalent and anomalous mutants. Note that Table 14 only includes mutation operators which are eligible in regard to automatic detection of equivalent mutants. Furthermore, a blank cell in any of the tables means that no value could be assigned to that category since no mutant has been generated by the respective operator.

When we consider the overall number of mutants produced per operator, we can see in Table 15 that three mutation operators (namely PSDR, DEWC and AJSC) generated no mutants for any of the systems. This indicates that AspectJ constructs targeted by these operators are not present in any of the evaluated versions of those systems. In regard to mutants produced by the PCCC operator, all were automatically classified as equivalent. However, such mutants should be manually revised given that the JP selections by the `cflow` and `cflowbelow` PCDs are evaluated at runtime in order to decide whether every specific JP selection holds of not within the running control flow context.

In the next section, we compare the obtained numbers of equivalent and anomalous mutants with the results obtained in our first case study earlier

Table 14: Percentages of equivalent mutants generated for the target systems.

	iBatis AO01		TollSystem		HW AO10		MM AO06		All Systems	
Operator	Total	Equiv.	Total	Equiv.	Total	Equiv.	Total	Equiv.	Total	Equiv.
PWIW	1976	92%	449	92%	646	87%	1075	96%	4146	92%
PWAR	0	–	4	0%	0	–	0	–	4	0%
PSWR	4	0%	5	20%	37	68%	0	–	46	57%
PSDR	0	–	0	–	0	–	0	–	0	–
POPL	193	63%	60	77%	70	74%	117	91%	440	74%
POAC	185	0%	41	0%	63	0%	75	9%	364	2%
POEC	19	58%	0	–	0	–	0	–	19	58%
PCTT	15	100%	21	81%	21	86%	39	85%	96	86%
PCGS	1	0%	1	0%	0	–	0	–	2	0%
PCCR	121	2%	17	0%	43	0%	120	32%	301	13%
PCLO	222	0%	30	37%	24	0%	21	29%	297	6%
PCCC	0	–	8	100%	1	100%	0	–	9	100%
TOTAL	2736	72%	636	78%	905	72%	1447	84%	5724	76%

presented in Section 7. We also estimate the effort required for developing mutation-adequate test sets based on the previously observed results.

8.3. Contrasting the results with the first case study

From Tables 14 and 15, we can observe that for all systems the obtained numbers of equivalent and anomalous mutants exceed the averages in our first case study. This is graphically shown in Figure 17.

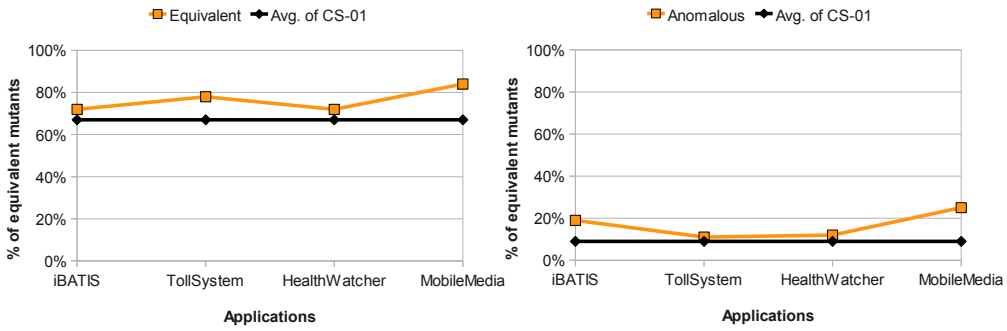


Figure 17: Percentages of equivalent and anomalous mutants for the target systems.

Considering the applications evaluated in Section 7, the average number of equivalent mutants was 67%, while this value for the four larger systems

Table 15: Percentages of anomalous mutants generated for the target systems.

	iBatis AO01		TollSystem		HW AO10		MM AO06		All Apps	
Operator	Total	Anom.	Total	Anom.	Total	Anom.	Total	Anom.	Total	Anom.
PWIW	1976	1%	449	1%	646	4%	1075	2%	4146	2%
PWAR	0	–	4	0%	0	–	0	–	4	0%
PSWR	4	25%	5	0%	37	5%	0	–	46	7%
PSDR	0	–	0	–	0	–	0	–	0	–
POPL	193	22%	60	3%	70	3%	117	6%	440	12%
POAC	185	96%	41	7%	63	5%	75	44%	364	59%
POEC	19	0%	0	–	0	–	0	–	19	0%
PCTT	15	0%	21	0%	21	0%	39	0%	96	0%
PCCE	153	50%	66	0%	52	23%	94	27%	365	31%
PCGS	1	0%	1	0%	0	–	0	–	2	0%
PCCR	121	7%	17	0%	43	5%	120	43%	301	21%
PCLO	222	40%	30	57%	24	17%	21	48%	297	40%
PCCC	0	–	8	0%	1	0%	0	–	9	0%
DAPC	0	–	0	–	1	0%	176	89%	177	88%
DAPO	0	–	0	–	1	0%	6	0%	7	0%
DSSR	69	81%	0	–	14	100%	20	100%	103	87%
DEWC	0	–	0	–	0	–	0	–	0	–
DAIC	8	0%	8	0%	0	–	0	–	16	0%
ABAR	123	48%	23	4%	52	4%	48	23%	246	30%
APSR	16	13%	25	24%	10	0%	41	32%	92	23%
APER	0	–	4	75%	0	–	4	0%	8	38%
AJSC	0	–	0	–	0	–	0	–	0	–
ABHA	95	0%	40	0%	47	0%	60	0%	242	0%
ABPR	405	37%	68	91%	146	52%	250	80%	869	56%
TOTAL	3605	19%	870	11%	1228	12%	2146	25%	7849	19%

analysed in this section represents 76% of the total. When it comes to anomalous mutants, the average amount for the larger systems is 19% against 9% for the smaller applications.

Let us consider a test set T_C that is adequate to an AO system as large as the ones evaluated in this case study. Moreover, T_C is adequate w.r.t. to a given criterion C other than Mutant Analysis (e.g. T_C is derived from either a functional-based or structural-based test selection criterion). Let us also consider (i) the observed number of equivalent and anomalous mutants listed in Tables 14 and 15, and; (ii) the effort required to create the T_M test sets in our previous case study. We can therefore assume that the effort testers need to dedicate to evolve T_C to cover all remaining live mutants after the execution of T_C shall be equal or less than 5% of $|T_C|$.

The above estimate relies on the fact that the systems evaluated in this section have a smaller proportion of live mutants to be handled than the

systems evaluated in the previous case study. Even though this a rough estimate, according to the results observed in our first case study we argue that well-designed test suites are able to reveal most of the faults simulated by the mutation operators. Nevertheless, a small but not less important test set increment is still needed in order to yield trustworthy results in regard to the Mutant Analysis criterion.

9. Study limitations

The main threat to the validity of our evaluation studies – and, consequently, to the achieved results – regards the representativeness of the selected applications, specially in our first case study (described in Section 7). Such study comprised a set of small AO applications upon which we performed the evaluation procedures and drew our conclusions. It required us to undertake a full – thus time consuming – test process, starting from the test plan design and ending up with the creation of adequate test suites that yielded full mutant coverage for all applications. The limited size of the applications allowed us to produce detailed analysis of the systems, comprehensive test plans and subsequent test data. Note that performing such a number of tasks from scratch for larger systems might have been prohibitive specially due to time constraints. Besides that, similar studies in regard to the size and the number of selected applications have been performed in order to evaluate other testing approaches for AO software [42, 58, 59].

Another possible threat to the validity of our first case study concerns the way we developed the initial test suites. Functional-based testing, as a black-box technique [49], relies on specification documents to derive the test requirements. However, the lack of detailed specifications for the applications evaluated in that case study led us to follow a "reverse" process to derive the *SFT-adequate* test sets. Instead of building our test plans and the respective test sets based exclusively on the specifications, we were forced to analyse the source code in order to comprehend the functionalities of each application. Therefore, the design of some test cases has also been guided by internal (structural) elements of the code rather than solely by functional requirements. As a consequence, the produced test sets may have led to higher mutant coverage given that, traditionally, structural-based (i.e. white-box) test suites [49] tend to outperform black-box ones [20].

In regard to our second case study (described in Section 8), our conclusions are based on estimates derived from the set of generated mutants and on

the results achieved in the previous case study. In spite of the observed trend regarding the number of generated mutants and on the estimated effort, more definite conclusions are only possible after running similar procedures with a larger set of applications. Exercising such applications with concrete test data will also help us draw more well-founded conclusions on the required effort to fulfil our testing approach requirements.

10. Related work

Research which is related to the work presented in this paper can be grouped in the following categories: (i) mutation testing of AO programs; and (ii) evaluation of AO testing approaches. Both categories are summarised in the sequence.

10.1. Mutation testing of AO programs

Recent surveys on mutation testing [15] and AO testing [14] identified a few approaches available in the literature [44, 50, 53, 60, 61] as well as automated supporting tools [60, 61]. The first initiative was presented by Mortensen and Alexander [44], which consists in a hybrid AO testing approach that combines coverage-based and mutation-based testing. It relies on a candidate fault model for AO programs defined by Alexander et al. [7]. Three mutation operators are defined to strengthen and weaken PCDs and to alter the advice precedence order. However, the authors do not provide details of syntactic changes and implications of each operator as we did for our mutation operators.

Lemos et al. [53] proposed another hybrid approach which involves structural and mutation testing of PCDs. The former consists of composing control flow graphs in order to detect unintended JP selection, while the latter is used to increase the sets of matched JPs. However, the authors only highlight the need for a comprehensive set of mutation operators to make the approach more effective. This paper addressed the design of such mutation operators.

Anbalagan and Xie [60] implemented a tool that automates the two PCD-related mutation operators defined by Mortensen and Alexander [44]. Mutants are produced through the use of wildcards as well as by using naming parts of original PCDs and JPs 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 PCDs. If a mutant selects the same set of JP as does the original PCD, it is automatically classified as equivalent. The final output is a list of the ranked mutants. Our set of operators covers a broader set of AspectJ features, while the *Proteum/AJ* tool provides support for other steps of mutation testing such as test execution and mutant evaluation.

Delamare et al. [50] proposed an approach based on test-driven development concepts and mutant analysis for testing AspectJ PCDs. The goal is to validate PCDs by means of test cases that explicitly define sets of JPs that should be affected by specific advices. In turn, PCDs that are bound to such advices are also validated. A mutation tool named *AjMutator* [61] implements seven of our PCD-related operators. The mutant PCDs are used to validate the effectiveness of their approach. The *AjMutator* tool automatically detects equivalent mutants based on JP matching information provided by the abc compiler [62], an alternative compiler for AspectJ programs. However, the tool misses some basic functionalities to properly support mutation testing. For instance, it does not allow for mutation operator selection, hence hindering testers to apply different strategies. The mutant analysis itself is limited to the automatic detection of equivalent mutants; other mutant handling features such as individual mutant execution and manual classification of mutants are not available. *Proteum/AJ* overcomes these main limitations of *AjMutator*.

10.2. Evaluation of AO testing approaches

Considering that Aspect-Oriented Programming is still a developing research topic, to date we can only find limited evidence w.r.t. evaluation and assessment of related testing approaches. In our survey [14], we were able to identify three pieces of work that regard evaluation of AO testing. They are both based on practical evaluations of small AO applications [42, 58].

Xie and Zhao [58] proposed a framework for automatic test generation based on state variables and method invocation wrappers. The quality of the generated test data is checked against a set of structural-based coverage criteria. The authors also defined some guidelines that require developer intervention in order to augment the initial coverage. To evaluate their approach, Xie and Zhao used a set of 12 small AO applications that partially overlaps with our set (e.g. Telecom, ProdLine and BankingSystem). Differently from our evaluation study, Xie and Zhao's test data was automatically

derived based on values of the input space. Furthermore, the adopted coverage measure does not rely on mutants but on internal structural elements such as control flow branches and unit interactions (e.g. method-method and method-advice).

Lemos et al. [42] presented an approach for structural integration testing of AO programs. They defined a set of control flow- and data flow-based coverage criteria for pairwise integration of communicating units (methods and advices). In their work, Lemos et al. estimate the cost for applying their criteria based on the number of test cases which are required to evolve the coverage from unit to integration level. Similarly to Xie and Zhao [58], they also used a set of small AO applications (seven in total) that partially overlaps with our set (e.g. Telecom, MusicOnline and StackManager). The average number of test cases per application evaluated by Lemos et al. was 14, number that increased 12.5% during the test sets evolution. We followed similar steps in our first case study; we compared the coverage SFT-adequate test sets yield w.r.t. to AOP-specific mutants. Differently from us, however, Lemos et al. estimate the costs based on requirements derived from two structural-based measures: unit coverage and pairwise coverage.

11. Conclusion and future research

To deal with testing-related specificities of contemporary programming techniques such as Aspect-Oriented Programming (AOP), at first software engineers need to understand how software faults can occur in practice so that they can be either avoided or rapidly localised within the software. In this context, mutation testing is a widely explored test selection criterion that focuses on recurring faults observed in the software. Starting from a well-characterised set of fault types, mutation testing helps to demonstrate the absence of such faults in the evaluated software products. However, when it comes to AOP, to date the few initiatives for customising the mutation testing for AO programs show either limited coverage with respect to the range of simulated faults or a need for proper evaluation in regard to attributes like application cost and effectiveness.

This paper contributes in this context. It revisited our previous research on fault-based testing of AO software [16, 18, 19] in order to define a comprehensive approach for mutation testing of AO programs. The contributions include (i) the definition a set of mutation operators for AspectJ programs that simulate a range of fault types; (ii) the implementation of a tool that

automates the application of the mutation operators; and (iii) evaluation of the proposed mutation operators by means of two case studies.

The evaluation results show that applying the mutation operators to AO programs does not impose high costs, even though it is likely to require increases of 5% in the test suites in order to achieve full mutant coverage. These studies helped to demonstrate the feasibility of fault-based testing for AO programs with adequate tool support.

Our future research includes: (i) possible refinements of the mutation operator set; (ii) improvements to the *Proteum/AJ* tool; and (iii) further empirical evaluation of the approach. We plan to analyse available fault reports for AO programs [18] in order to figure out new mutation operators needs. Besides that, we aim to evolve *Proteum/AJ* to support varied mutation testing strategies such as Constrained Mutation [63] and Selective Mutation [64]. Finally, we will configure other evaluation studies similar to the ones presented in Sections 7 and 8 of this paper, addressing different sets of applications.

Acknowledgements

We would like to thank Otávio Lemos from UNIFESP, Brazil, and Giuseppe Di Lucca from Università degli Studi del Sannio, Italy, for providing some of the AspectJ applications we analysed in our first case study.

The authors received full of partial funding from the following agencies and projects: *Fabiano Ferrari*: FAPESP (grant 05/55403-6), CAPES (grant 0653/07-1) and EC Grant AOSD-Europe (IST-2-004349); *Awais Rashid*: EC Grant AOSD-Europe (IST-2-004349); *José Maldonado*: EC Grant QualiPSo (IST-FP6-IP-034763), FAPESP, CAPES and CNPq.

References

[1] G. Kiczales et al., Aspect-Oriented Programming, in: Proceedings of the 11^{th} European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, Jyväskylä - Finland, 220–242 (LNCS v.1241), 1997.

[2] M. Mortensen, S. Ghosh, J. M. Bieman, Aspect-Oriented Refactoring of Legacy Applications: An Evaluation, IEEE Transactions on Software Engineering (in press).

[3] A. Rashid et al., Aspect-Oriented Software Development in Practice: Tales from AOSD-Europe, IEEE Computer 43 (2) (2010) 19–26.

[4] R. Laddad, Aspect-Oriented Programming Will Improve Quality, IEEE Software 20 (6) (2003) 90–91.

[5] Y. Coady, G. Kiczales, Back to the Future: A Retroactive Study of Aspect Evolution in Operating System Code, in: Proceedings of the 2^{nd} International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, Boston/MA - USA, 50–59, 2003.

[6] J. Zhao, Data-Flow-Based Unit Testing of Aspect-Oriented Programs, in: Proceedings of the 27^{th} Annual IEEE International Computer Software and Applications Conference (COMPSAC), IEEE Computer Society, Dallas/Texas - USA, 188–197, 2003.

[7] R. T. Alexander, J. M. Bieman, A. A. Andrews, Towards the Systematic Testing of Aspect-Oriented Programs, Tech. Report CS-04-105, Dept. of Computer Science, Colorado State University, Fort Collins/Colorado - USA, 2004.

[8] P. Greenwood et al., On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study, in: Proceedings of the 21^{st} European Conference on Object-Oriented Programming (ECOOP), Springer Berlin, Berlin - Germany, 176–200 (LNCS v.4609), 2007.

[9] F. C. Ferrari et al., An Exploratory Study of Fault-Proneness in Evolving Aspect-Oriented Programs, in: Proceedings of the 32^{nd} International Conference on Software Engineering (ICSE), ACM Press, Cape Town - South Africa, 65–74, 2010.

[10] R. Coelho et al., Assessing the Impact of Aspects on Exception Flows: An Exploratory Study, in: Proceedings of the 22^{nd} European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, Paphos - Cyprus, 207–234 (LNCS v.5142), 2008.

[11] R. A. DeMillo, R. J. Lipton, F. G. Sayward, Hints on Test Data Selection: Help for the Practicing Programmer, IEEE Computer 11 (4) (1978) 34–43.

[12] J. H. Andrews, L. C. Briand, Y. Labiche, Is Mutation an Appropriate Tool for Testing Experiments?, in: Proceedings of the 27^{th} International Conference on Software Engineering (ICSE), ACM Press, St. Louis/MO - USA, 402–411, 2005.

[13] H. Do, G. Rothermel, On the Use of Mutation Faults in Empirical Assessments of Test Case Prioritization Techniques, IEEE Transactions on Software Engineering 32 (9) (2006) 733–752.

[14] F. C. Ferrari, E. N. Höhn, J. C. Maldonado, Testing Aspect-Oriented Software: Evolution and Collaboration through the Years, in: Proceedings of the 3^{rd} Latin American Workshop on Aspect-Oriented Software Development (LAWASP), Brazilian Computer Society, Fortaleza/CE - Brazil, 24–30, 2009.

[15] Y. Jia, M. Harman, An Analysis and Survey of the Development of Mutation Testing, IEEE Transactions on Software Engineering (in press).

[16] F. C. Ferrari, J. C. Maldonado, A. Rashid, Mutation Testing for Aspect-Oriented Programs, in: Proceedings of the 1^{st} International Conference on Software Testing, Verification and Validation (ICST), IEEE Computer Society, Lillehammer - Norway, 52–61, 2008.

[17] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, W. G. Griswold, An Overview of AspectJ, in: Proceedings of the 15^{th} European Conference on Object-Oriented Programming (ECOOP), Springer-Verlag, Budapest - Hungary, 327–353 (LNCS v.2072), 2001.

[18] F. C. Ferrari, R. Burrows, O. A. L. Lemos, A. Garcia, J. C. Maldonado, Characterising Faults in Aspect-Oriented Programs: Towards Filling the Gap between Theory and Practice, in: Proceedings of the 24^{th} Brazilian Symposium on Software Engineering (SBES), Salvador/BA - Brazil, (to appear), 2010.

[19] F. C. Ferrari, E. Y. Nakagawa, A. Rashid, J. C. Maldonado, Automating the Mutation Testing of Aspect-Oriented Java Programs, in: Proceedings of the 5^{th} ICSE International Workshop on Automation of Software Test (AST), ACM Press, Cape Town - South Africa, 51–58, 2010.

[20] A. P. Mathur, Foundations of Software Testing, Addison-Wesley Professional, 2007.

[21] E. W. Dijkstra, A Discipline of Programming, Prentice-Hall, 1976.

[22] E. Baniassad, S. Clarke, Theme: An Approach for Aspect-Oriented Analysis and Design, in: Proceedings of the 26^{th} International Conference on Software Engineering (ICSE), IEEE Computer Society, Edinburgh - UK, 158–167, 2004.

[23] M. P. Robillard, G. C. Murphy, Representing Concerns in Source Code, ACM Transactions on Software Engineering and Methodology (TOSEM) 16 (1).

[24] R. Johnson, J. Hoeller, A. Arendsen, C. Sampaleanu, R. Harrop, T. Risberg, D. Davison, D. Kopylenko, M. Pollack, T. Templier, E. Vervaet, P. Tung, B. Hale, A. Colyer, J. Lewis, C. Leau, R. Evans, Spring - Java/J2EE Application Framework, Reference Manual Version 2.0.6, Interface21 Ltd., 2007.

[25] The JBoss Team, JBoss AOP Reference Documentation V2.0, Online, `http://docs.jboss.org/jbossaop/docs/index.html` - last accessed 15/09/2010, 2010.

[26] M. Mezini, K. Ostermann, Conquering Aspects with Caesar, in: Proceedings of the 2^{nd} International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, Boston/MA - USA, 90–99, 2003.

[27] R. E. Filman, D. Friedman, Aspect-Oriented Programming is Quantification and Obliviousness, in: R. E. Filman, T. Elrad, S. Clarke, M. Akşit (Eds.), Aspect-Oriented Software Development, chap. 2, Addison-Wesley, Boston, 21–35, 2004.

[28] The Eclipse Foundation, AspectJ Documentation, Online, `http://www.eclipse.org/aspectj/docs.php` - last accessed on 15/09/2010, 2010.

[29] R. Laddad, AspectJ In Action, Manning Publications, 2003.

[30] A. J. Offutt, J. Pan, Automatically Detecting Equivalent Mutants and Infeasible Paths, Journal of Software Testing, Verification, and Reliability 7 (3) (1997) 165–192.

[31] R. A. DeMillo, A. J. Offutt, Constraint-Based Automatic Test Data Generation, IEEE Transactions on Software Engineering 17 (9) (1991) 900–910.

[32] H. Agrawal et al., Design of Mutant Operators for the C Programming Language, Technical Report SERC-TR41-P, Software Engineering Research Center, Purdue University, West Lafayette/IN - USA, 1989.

[33] M. E. Delamaro, J. C. Maldonado, A. P. Mathur, Interface Mutation: An Approach for Integration Testing, IEEE Transactions on Software Engineering 27 (3) (2001) 228–247.

[34] Y. S. Ma, Y. R. Kwon, J. Offutt, Inter-class Mutation Operators for Java, in: Proceedings of the 13^{th} International Symposium on Software Reliability Engineering (ISSRE), IEEE Computer Society Press, Annapolis/MD - USA, 352–366, 2002.

[35] J. C. King, Symbolic execution and program testing, Communications of the ACM 19 (7) (1976) 385–394.

[36] J. S. Bækken, A Fault Model for Pointcuts and Advice in AspectJ Programs, Master's thesis, School of Electrical Engineering and Computer Science, Washington State University, Pullman/WA - USA, 2006.

[37] A. Gal, W. Schröder-Preikschat, O. Spinczyk, AspectC++: Language Proposal and Prototype Implementation, in: Proceedings of the Workshop on Advanced Separation of Concerns in Object-Oriented Systems, Tampa Bay/Florida - USA, 2001.

[38] E. Hilsdale, J. Hugunin, Advice weaving in AspectJ, in: Proceedings of the 3^{rd} International Conference on Aspect-Oriented Software Development (AOSD), ACM Press, Lancaster - UK, 26–35, 2004.

[39] J. C. Maldonado, M. E. Delamaro, S. C. P. F. Fabbri, A. S. Simão, T. Sugeta, P. C. Masiero, Proteum: A Family of Tools to Support Specification and Program Testing Based on Mutation, in: Mutation 2000 Symposium - Tool Session, Kluwer Academic Publishers, San Jose/CA - USA, 113–116, 2000.

[40] O. A. L. Lemos, A. M. R. Vincenzi, J. C. Maldonado, P. C. Masiero, Control and Data Flow Structural Testing Criteria for Aspect-Oriented Programs, Journal of Systems and Software 80 (6) (2007) 862–882.

[41] R. E. Lopez-Herrejon, D. Batory, Using AspectJ to Implement Product-Lines: A Case Study, Technical Report, Department of Computer Sciences, The University of Texas, Austin, Texas- USA, 2002.

[42] O. A. L. Lemos, I. G. Franchin, P. C. Masiero, Integration testing of Object-Oriented and Aspect-Oriented programs: A structural pairwise approach for Java, Science of Computer Programming 74 (10) (2009) 861–878.

[43] C.-H. Liu, C.-W. Chang, A State-Based Testing Approach for Aspect-Oriented Programming, Journal of Information Science and Engineering 24 (1) (2008) 11–31.

[44] M. Mortensen, R. T. Alexander, An Approach for Adequate Testing of AspectJ Programs, in: Proceedings of the 1^{st} Workshop on Testing Aspect Oriented Programs (WTAOP), Chicago/IL - USA, 2005.

[45] T. Xie, J. Zhao, D. Marinov, D. Notkin, Automated Test Generation for AspectJ Programs, in: Proceedings of the 1^{st} Workshop on Testing Aspect Oriented Programs (WTAOP), Chicago/IL - USA, 2005.

[46] Y. Zhou, D. J. Richardson, H. Ziv, Towards A Practical Approach to Test Aspect-Oriented Software, in: Proceedings of the Net.ObjectiveDays 2004 Workshop on Testing Component-based Systems (TECOS), Germany, 1–16, 2004.

[47] M. L. Bernardi, G. A. D. Lucca, Testing Aspect Oriented Programs: an Approach Based on the Coverage of the Interactions among Advices and Methods, in: Proceedings of the 6^{th} International Conference on Quality of Information and Communications Technology (QUATIC), IEEE Computer Society, Lisbon - Portugal, 65–76, 2007.

[48] S. Linkman, A. M. R. Vincenzi, J. C. Maldonado, An Evaluation of Systematic Functional Testing Using Mutation Testing, in: Proceedings of the 7^{th} International Conference on Empirical Assessment in Software Engineering (EASE), Keele - UK, 1–15, 2003.

[49] G. J. Myers, C. Sandler, T. Badgett, T. M. Thomas, The Art of Software Testing, John Wiley & Sons, 2nd edn., 2004.

[50] R. Delamare, B. Baudry, S. Ghosh, Y. Le Traon, A Test-Driven Approach to Developing Pointcut Descriptors in AspectJ, in: Proceedings of the 2^{nd} International Conference on Software Testing, Verification and Validation (ICST), IEEE Computer Society, Denver/CO - USA, 376–385, 2009.

[51] E. F. Barbosa, J. C. Maldonado, A. M. R. Vincenzi, Toward the Determination of Sufficient Mutant Operators for C, The Journal of Software Testing, Verification and Reliability 11 (2) (2001) 113–136.

[52] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, C. Zapf, An Experimental Determination of Sufficient Mutant Operators, ACM Transactions on Software Engineering and Methodology (TOSEM) 5 (2) (1996) 99–118.

[53] O. A. L. Lemos, F. C. Ferrari, P. C. Masiero, C. V. Lopes, Testing Aspect-Oriented Programming Pointcut Descriptors, in: Proceedings of the 2^{nd} Workshop on Testing Aspect Oriented Programs (WTAOP), ACM Press, Portland/Maine - USA, 33–38, 2006.

[54] AOSD Europe, Project Home Page, `http://www.aosd-europe.net/` - last accessed on 15/09/2010, 2010.

[55] E. Figueiredo et al., Evolving Software Product Lines with Aspects: An Empirical Study on Design Stability, in: Proceedings of the 30^{th} International Conference on Software Engineering (ICSE), ACM Press, Leipzig - Germany, 261–270, 2008.

[56] S. Soares, P. Borba, E. Laureano, Distribution and Persistence as Aspects, Software - Practice & Experience 36 (7) (2006) 711–759.

[57] iBATIS Development Team, Apache iBATIS home page, Online, `http://attic.apache.org/projects/ibatis.html` - last accessed on 15/09/2010, 2009.

[58] T. Xie, J. Zhao, A Framework and Tool Supports for Generating Test Inputs of AspectJ Programs, in: Proceedings of the 5^{th} International

Conference on Aspect-Oriented Software Development (AOSD), ACM Press, Bonn - Germany, 190–201, 2006.

[59] G. Xu, A. Rountev, Regression Test Selection for AspectJ Software, in: Proceedings of the 29^{th} International Conference on Software Engineering (ICSE), IEEE Computer Society, Minneapolis/MN - USA, 65–74, 2007.

[60] P. Anbalagan, T. Xie, Automated Generation of Pointcut Mutants for Testing Pointcuts in AspectJ Programs, in: Proceedings of the 19^{th} International Symposium on Software Reliability Engineering (ISSRE), IEEE Computer Society, Seattle/WA - USA, 239–248, 2008.

[61] R. Delamare, B. Baudry, Y. Le Traon, AjMutator: A Tool for the Mutation Analysis of AspectJ Pointcut Descriptors, in: Proceedings of the 4^{th} International Workshop on Mutation Analysis (Mutation), IEEE Computer Society, Denver/CO - USA, 200–204, 2009.

[62] abc Development Team, abc: The AspectBench Compiler for AspectJ, Online, `http://abc.comlab.ox.ac.uk/` - last accessed on 15/09/2010, 2009.

[63] A. P. Mathur, W. E. Wong, Evaluation of the Cost of Alternative Mutation Strategies, in: Proceedings of the 7^{th} Brazilian Symposium on Software Engineering (SBES), João Pessoa/PB - Brazil, 320–335, 1993.

[64] A. J. Offutt, G. Rothermel, C. Zapf, An Experimental Evaluation of Selective Mutation, in: Proceedings of the 15^{th} International Conference on Software Engineering (ICSE), IEEE Computer Society, Baltimore/MD - USA, 100–107, 1993.