8
Concluding Remarks and Future Work

This chapter summarizes the work described thorough this dissertation. It presents our conclusions, lists the contributions of our work and how they have been accomplished. Finally it presents directions for future research and briefly describes the PhD roadmap that culminated in the work detailed here.

8.1.
Conclusions

This work can be divided into three main parts that complement each other: (i) the exploratory study; (ii) the development of SAFE (*Static Analysis for the Flow of Exceptions*), an exception-flow analysis tool for AspectJ programs; and (iii) the definition of a verification approach, based on SAFE, for the exception handling code of AspectJ programs.

We have performed the exploratory study to evaluate the impact of aspects on the exceptional control flow of programs. In this study, we selected 3 systems that were implemented both in Java and AspectJ: Health Watcher (Soares, 2004; Greenwood et al., 2007), Mobile Photo (Figueiredo et al., 2008) and JHotDraw (Deursen et al., 2005). For the Health Watcher and the Mobile Photo systems different releases were investigated. Then, we compared the Java and AspectJ versions of each system release in terms of the number of *uncaught exceptions*, exceptions caught by *subsumption*, and exceptions caught with specialized handlers. In all the AspectJ versions, we observed a significant increase in the number of *uncaught* exceptions and exception *subsumptions*, and a decrease in the number of exceptions caught by specialized handlers. To find out what caused such negative effects in AspectJ releases, we performed a systematic inspection of the exception handling code of each system.

During the manual code inspections we discovered a set of recurring program anomalies in the exception handling code of AspectJ programs. We organized these anomalies into a catalogue of bug patterns related to the exception

handling code. These bugs came mainly from three sources: aspects acting as handlers, aspects as exception signalers, and misuses of the `declare soft` construct.

Our findings indicate that the exception handling code of AO systems can indeed be fault prone. We have observed that some characteristics of AO compositions such as (i) *the ability to externally modify the basecode* (Krishnamurthi et al., 2004; Aldrich, 2005), (ii) some developers and approaches advocating an *oblivious development process* (Filman and Friedman, 2005), (iii) *the load-time weaving* (Colyer, 2004; Bodkin, 2005; Bodkin, 2006) available in some Aspect Oriented (AO) languages, and (iv) the *quantification* property (Filman and Friedman, 2005) strengthen problems that already exists in OO system development (e.g., *uncaught exceptions* and *unintended handler action*).

We have observed that AO compositions have the ability of affecting negatively the robustness of exception-aware software systems. Therefore, there is a need for both: improving the design of exception handling mechanisms in AO programming languages, and building verification tools and techniques tailored to improve the reliability of the error handling code in aspect-oriented programs.

To support the empirical study described above we developed SAFE (*Static Analysis for the Flow of Exceptions*) a static analysis tool that calculates the exception-flow of AspectJ programs. The exception-flow analysis algorithm implemented in SAFE traverses a program representation, based on the program call graph and exceptional control flow information, to discover: (i) every exception that may escape from an application or an advice method, and (ii) the *exception path* of each exception. The *Exception Path Miner,* one of the SAFE tool's components, parses the exception paths and classifies them according to the *Signaler-Handler* relationship. This information is useful for guiding manual inspections of the exception handling code, and discovering the elements responsible for signaling the exceptions that are not adequately handled inside the system (e.g., *uncaught* exceptions).

The lack of verification approaches to the exception handling code of AO applications motivated the third part of our work: the development of a verification approach for the exception handling code. The approach proposed here is supported by the SAFE tool and aims at assisting the developer when checking the reliability of the exception handling code of AO applications. This

approach provides brief guidelines for the developer on how such exceptions should be handled inside an AO application.

8.2. Contributions

The main contributions of this work are as follows:

- ***Exploratory Study.*** This work presents the first systematic analysis which aims at investigating how aspects affect the exception flows of programs (Chapter 3).
- ***Bug Patterns Catalogue for the Exception Handling Code of AspectJ Programs.*** One of the main outcomes of our exploratory study was a set of *bug-patterns* related to the exception handling code of AO programs which were characterized based on the data empirically collected (Chapter 4).
- ***Analysis of AO Compositions' Characteristics x Exception Handling Mechanisms.*** The goal of exception handling mechanisms is to make programs more reliable and robust. In this study we could observe that some properties of AOP may conflict with characteristics of exception handling mechanisms. We discuss, based on data empirically collected during the study, how the *quantification* and *obliviousness* properties pose specific pitfalls to the design of exception handling code. (Chapter 4, Section 4.2.2)
- ***Other AO Languages.*** To answer the question of *to which extent our findings could be applied for other systems implemented in other AOP languages*, we have investigated other AOP technologies (i.e., CaesarJ, JBoss AOP and Spring AOP) derived from Java language. We have observed that they followed similar join point model as the one used by AspectJ, and offered closely related *pointcut* designators. Such common characteristics among AO languages, therefore, allow aspects to add or modify the behavior on similar join points, potentially adding new exceptions. Consequently, although the exploratory study has focused on AspectJ systems, our findings (e.g., most of the bugs from the bug pattern catalogue) can be applied to systems developed in other AO languages. (Chapter 4, Section 4.2.1)

- ***Exception-Flow Analysis tool for AspectJ programs.*** This work presents an exception flow analysis tool for AspectJ programs (Chapter 5), which was initially developed to support the exploratory study presented here. It interprets the constructs added by AspectJ weaver (neglected by existing OO static analysis tools) and defines a set of heuristics to deal with AO concepts during the exception flow analysis (Chapter 5, Section 5.2).

- ***A Verification Approach for the Exception Handling Code of AspectJ systems.*** The lack of verification approaches for the exception handling code of AO systems, and the inherent difficulties associated with the testing of exception handling code, motivated the verification approach proposed in out work. The verification approach proposed here is based on the use of SAFE tool to statically check the reliability of the exception handling code of AO applications (Chapter 6).

- ***Explore the Collateral Effects of Aspect Library Reuse.*** Aspect libraries are a relatively new reuse artifact, and only seminal studies had been performed so far. In this work we discussed the potential faults associated with *library aspects* reuse in the presence of exceptions (Chapter 6, Section 6.1.1). Moreover, it provides a way to identify potential problems that may happen on different aspect reuse scenarios.

The contributions of this work allow for: (i) developers of robust aspect-oriented applications to make more informed decisions in the presence of evolving exception flows - added when developing new aspects or integrating new aspect libraries, (ii) designers of AOP languages and static analysis tools to consider pushing the boundaries of existing mechanisms to make AOP more robust and resilient to exceptional conditions. Moreover, the proposed approach is also useful to help developers when building their own reusable aspect libraries.

8.3. Future Work

There are several ways our work can be continued, as follows:

- ***Generalize the Study Results to AO Programs Built from Scratch.*** It was our goal in this first empirical study to have an initial understanding of the

extent AspectJ mechanisms increase/decrease the number of errors in the exception handling code when compared with the Java counterparts. Since, many AO systems nowadays are generated from an OO version in which crosscutting concerns are *refactored* to aspects, the results of our study can be directly applied to a large set of current systems. However, the rate of bugs in AO systems built from scratch might be different. Although, we believe that the results of this study (e.g. catalogue of bug patterns) are of relevance for AO software built from scratch, since our catalogue reports potential misuses of AOP mechanisms in general inherent to aspect intrusive characteristics, an interesting future work is to run the analysis on some AspectJ programs built from scratch.

- ***Reuse Exception-Flow Information.*** The exception-flow information calculated for library methods called by an application could be calculated only once and then reused. This will positively impact the performance of our inter-procedural analysis.

- ***Integrate Exception-Flow Information with the IDE.*** The exception flow information calculated by SAFE at compile-time could be integrated with a development environment such as Eclipse as proposed by (Sinha et al., 2004). Such integration would enable the developer to navigate through the exception interfaces of methods (and method-like constructs) during development, eventually helping him/her to remove reported defects. Such integration could therefore contribute to the reduction of the number of exception handling defects - one of the main causes of software crashes.

- ***Perform Extensive Validation for Our Approach***. A future work is to apply the verification approach proposed here to other implementation scenarios in different AO systems. Such extensive validation could include qualitative metrics comprising the developers' opinion while using the approach. Moreover, we could also investigate the utility of this approach in different software evolution scenarios (e.g., corrective, adaptive and perfective tasks).

- ***Adapt the Approach.*** The verification approach could be refined to tackle specific problems of aspect library development. For instance, the exception interface of library aspects could be defined as one of the aspect libraries

artifacts. As a crosscutting interface (XPI) (Sullivan et al., 2005) documents the points of a system that can be affected by aspects, the *Exceptional Interface of Aspect Library* (EXI) could document which exceptions (*checked* or *unchecked*) can be signaled by each library aspect. The SAFE tool could calculate the exception interface of aspects and automatically generate the EXI.

- ***Extend the SAFE Tool to Support other AO Languages .*** The current implementation of SAFE tool can only analyze the exception flow of Java and AspectJ programs. An interesting future work is to extend the SAFE tool in order to enable the analysis of other AO Java-based languages such as: CaesarJ, Spring AOP, JBoss AOP. One way of doing this is to integrate it with CAPE (*Common Aspect Proof Environment*) (Faitelson and Katz, 2008) an extensible platform for integrating verification and analysis tools of aspect-oriented programs. The CAPE proof environment enables a tool designed to analyze AO programs to execute in a variety of AO languages.
- ***Improve the SAFE Tool Performance.*** The current version of SAFE tool depends on the Soot framework to build the program call graph and compute the exceptions that may flow from a method-like construct. Soot is a general propose bytecode analysis framework that can be used for transformations and optimizations of Java bytecode. It performs a set of steps that are not of interest to our analysis. Therefore, a straightforward way to improve the performance of SAFE tool would be to use other call graph construction algorithms (Grove and Chambers, 2001; (Salcianu, 2001) or even develop a limited version of Soot to meet SAFE tool's requirements.

8.4. PhD Roadmap

This section briefly presents the roadmap followed during this PhD. It summarizes the issues investigated during this period and the publications directly related to them. Figure 29 depicts the research issues and related papers ordered in a timeline.

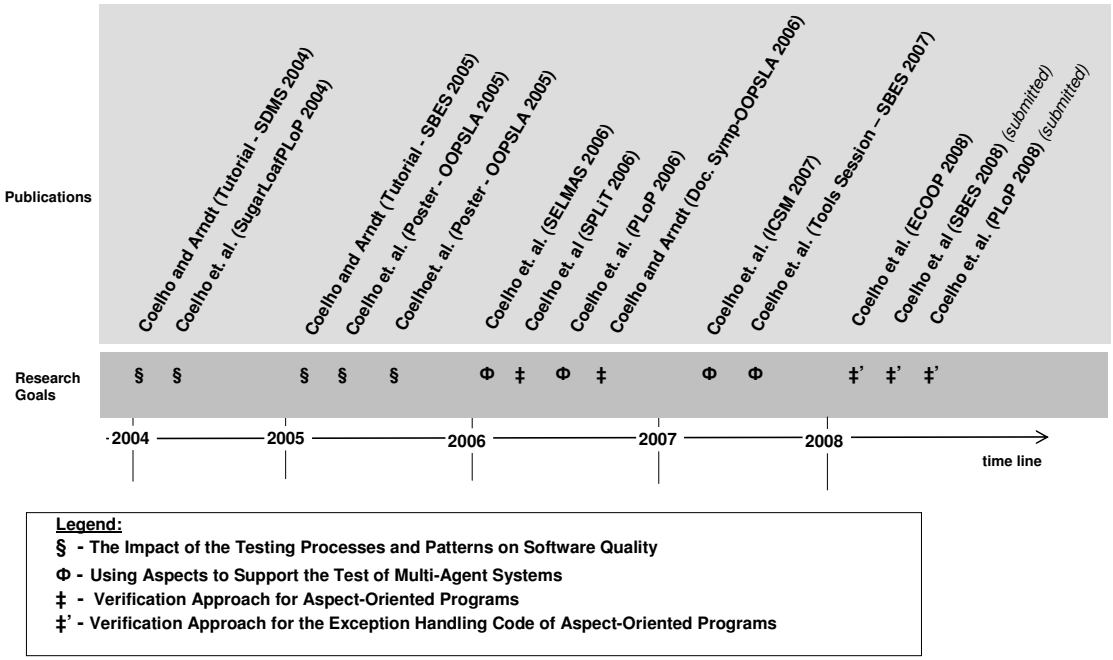

Figure 29. Research works organized in a timeline.

During my first and second years at PUC-Rio as a PhD candidate my research mainly focused on investigating of the impact of testing approaches on software quality. During these years I investigated *state of the art* testing practices (see [1] and [3] in Table 16). Based on research ideas and my previous experience as a Software Engineer in companies and P&D projects, we proposed testing approaches based on: (i) testing practices of agile methodologies such as Extreme Programming (see [4]); (ii) and the definition and use of Architectural Test Patterns (see [2] and [5] in Table 16).

In January 2006 during a three-month research visit at Waterloo University I started researching about the use of *Aspect technology* to support the testing process. During this year we defined a testing approach for asynchronous multi-agent systems based on the use of *Aspect technology* to *control* the test input and *observe* the test output of systems composed by multiple autonomous agents (see [6] and [7] in Table 16). We also developed a supporting tool called JAT for the definition and execution of the agent tests developed according this approach. During this research we observed that there was a lack of approaches to test aspects – there were few guidelines and tools available to help us to test the crosscutting features of JAT tool (developed in AspectJ language).

This motivated my research in the field of aspect-oriented software verification; in this same year we proposed a testing approach for *crosscutting features* (see [8] and [9] in Table 16). Since then, I have continued the research on aspect-oriented software verification and it became the main research theme of my PhD.

In 2007, during a seven-month research visit at Lancaster University, we conducted an empirical study that revealed the flaws in the exception handling code of aspect oriented programs (see [12] in Table 16). This motivated me to narrow my research to the definition of a verification approach and a supporting tool to the exception handling code of aspect-oriented system (see [13] and [14] in Table 16). During this same year we also evolved the testing tool for multi-agent systems based on the use of aspects (see [10] and [11] in Table 16). Therefore, we can observe that during this PhD one research question motivated others, which directly and indirectly contributed to the research work described here.

Selected Publications
[1] COELHO, R.; STAA, A. V. ; **Tutorial: Strategies to Achieve Correct Software**. IV Simpósio de Desenvolvimento e Manutenção de Software da Marinha (SDMS 2004), Rio de Janeiro, 2004.
[2] COELHO, R.; KULESZA, U. ; STAA, A. V. ; LUCENA, C. J. P. . **Layred Information System Test Pattern**. Fifth Latin American Conference on Pattern Languages of Programming (SugarLoafPLoP 2004), 2004.
[3] COELHO, R.; STAA, A. V. ; **Tutorial: Software Testing,** 19 Brazilian Symphosium on Software Engineering (SBES 2005), Uberlândia, 2005.
[4] COELHO, R.; BRASILEIRO, E. V. ; STAA, A. V. .**Not so eXtreme programming: agile practices for R&D projects**. In: Object- Oriented Programming, Systems, Languages and Applications (OOPSLA 2005), 2005. *(poster)*
[5] COELHO, R.; KULESZA, U. ; STAA, A. V. **Improving Architecture Testability with Patterns**. In: Object- Oriented Programming, Systems, Languages and Applications (OOPSLA 2005), 2005. *(poster)*
[6] COELHO, R., KULESZA, U., STAA, A. V., LUCENA, C. **Unit Testing in Multi-agent Systems using Mock Agents and Aspects.** International Workshop on Software Engineering for Large-scale Multi-Agent Systems (SELMAS2006), 2006.
[7] COELHO, R., DANTAS, A., KULESZA, U., STAA, A. V., CIRNE, W.,

LUCENA, C. **The Monitor Aspect Pattern**, Pattern Languages of Programming Design (PLoP 2006) in conjunction with OOPSLA 2006, 2006.
[8] COELHO, R., ALVES, A., KULESZA, U., COSTA, A., STAA, A. V., LUCENA, C., BORBA, P. **On Testing Crosscutting Features Using Extension Join Points**, 3rd Workshop on Product Line Testing (SPLiT 2006).
[9] COELHO, R., A., STAA. **Using Interfaces to Support the Testing of Crosscutting Features.** In: Doctoral Symposium of Object- Oriented Programming, Systems, Languages and Applications (OOPSLA 2006), 2006.
[10] COELHO, R.; CIRILO, E.; KULESZA, U.; STAA, A., RASHID, A.; LUCENA, C.; **JAT: A Test Automation Framework for Multi-Agent Systems** , International Conference on Software Maintenance (ICSM 2007), 2007.
[11] COELHO, R.; CIRILO, E.; KULESZA, U.; STAA, A. V., RASHID, A.; LUCENA, C.;**JAT Framework: Creating JUnit-Style Tests for Multi-Agent Systems**, Tools Session - Brazilian Symposium on Software Engineering (SBES 2007), 2007.
[12] COELHO, R.; RASHID, A.; GARCIA, A.; FERRARI, F.; CACHO, N.; KULESZA, U.;STAA, A. V.; LUCENA, C.; **Assessing the Impact of Aspects on Exception Flows: An Exploratory Study**, European Conference on Object Oriented Programming (ECOOP 2008), 2008.
[13] COELHO, R.; RASHID, A.; KULESZA, U.; STAA, A. V.; LUCENA, C.; **Unveiling and Taming the Liabilities of Aspect Libraries Reuse**, Brazilian Symposium on Software Engineering, (SBES 2008). (*submitted*)
[14] COELHO, R.; RASHID, A.; KULESZA, U.; STAA, A. V..; LUCENA, C.; **Exception Handling Bug Patterns in Aspect-Oriented Programs,** Pattern Languages of Programming Design (PLoP 2008) in conjunction with OOPSLA 2008, 2008. (*submitted*)

Table 15. List of selected publications.