1
Introduction

The separation of concerns has always been one of the main goals of software engineering. Recently, there is a growing consensus that some concerns may not be adequately represented by the traditional decomposition mechanisms (e.g., functions, classes, components) because they are not localized in a single module and are usually observed in many of them. Since then, new decomposition mechanisms have been proposed to address this limitation. In general, these mechanisms provide constructs to externally augment the behavior of a module (Aldrich, 2005). Such modifications include the behavior that could not be represented on traditional modules, and can happen statically (i.e., through the manipulation of the source code) or dynamically (i.e., the functionality is conditionally included depending on the program execution state).

One example of such modifications was introduced by the Common Lisp Object System (CLOS) (Bobrow et al., 1988). CLOS proposed a construct called advice, which defines an additional behavior that could be included *before* the execution of a method, or *after*, or even *before* and *after* at the same time. This new construct had the ability to change the program control flow - even preventing the original body of a method to be executed. This new composition mechanism opened a new realm of possibilities in software development.

The Aspect-Oriented Programming (AOP) (Kiczales, 1996; Kiczales et al., 1997) is the most popular form of such way of composition mechanism nowadays. The goal of AOP is to modularize concerns that crosscuts the primary decomposition of a software system through a new abstraction called aspect. The aspect uses advice constructs to include additional behavior on specific points in the code, called program join points (e.g., method call, method execution, and object initialization). Pointcuts are constructs, defined inside the aspects, that help identify these join points on code – they are represented by expressions that syntactically matches a specific set of join points. AspectJ (Kiczales et al., 2001a; Kiczales et al., 2001b) is the most used aspect-oriented programming language

nowadays. It adds the AO constructs to the Java programming language, allowing developers to structure a program using both classes and *aspects* to represent crosscutting concerns.

It has been empirically observed that AOP decompositions promote modularity (Garcia et al., 2005; Kiczales and Mezini, 2005) design stability (Greenwood et al., 2007), and that aspects abstractions can be used to modularize the conventional crosscutting concerns such as transaction management (Rashid and Chitchyan, 2003; Soares et al., 2006), distribution (Soares et al., 2006), and certain design patterns (Hannemann and Kiczales, 2002; Garcia et al., 2005) and exception handling (Filho et al., 2006; Filho et al., 2007) (in specific situations). Most of the existing research works and empirical studies that have investigated the positive and negative impacts of AOP in the modularization of crosscutting concerns have focused on the "normal" control flow of programs (Katz, 2006; Sanen, 2006; Rinard, 2004). Only few of them have mentioned the exceptions that may be thrown by aspects (Soares, 2004), and the pitfalls associated with specific AO constructs (e.g., exception softening (Colyer, 2004; Kienzle and Guerraoui, 2002; Rashid and Chitchyan, 2003)). However, the impact of aspects in the exceptional control flow is broader than AO specific constructs. It is intrinsically related to how the aspect *composition mechanism* works.

1.1. The Problem

Aspects allow a developer to externally augment the behavior of a method (Krishnamurthi et al., 2004; Aldrich, 2005). This way of composition works *in reverse*: the aspect declares which classes it should affect rather than vice-versa. This means that removing and adding aspects to a program usually[1] does not require editing the affected class definitions.

When an aspect adds a new functionality to a program, this additional functionality may bring new exceptions (i.e., abnormal computation states that arise as a consequence of faults on the application itself) such as: access of null references or division by zero, a noisy user input or faults on an underlying

[1] In some scenarios where AspectJ inter-type declarations are used, the developer may edit parts of the base code in order to include calls the the methods added by inter-type declarations.

middleware. Such exceptions will flow through the advised method call chain until they are handled. In Java and AspectJ programs if no handler is found for the exception, it remains *uncaught*, causing the software to crash in an unpredictable way - which is one of the main causes of failure in Java programs (Jo et al., 2004). Aspect developers may do their best to ensure that aspects do not create faults that impact the applications. However, unexpected behavior in aspect code such as referencing unanticipated null values or calling a library method that throws undocumented runtime exceptions is often present (Coelho et. al., 2008).

Moreover, the verification approaches for AO programs proposed so far do not take into account the exceptional scenarios (i.e., scenarios related to exceptions occurrences and handling). They focus mainly on testing the "normal" program behavior. The reason is twofold. Since the exception handling code is not the *primary concern* to be implemented, it does not receive much attention during the *verification phase*. Moreover, testing the exception handling code is inherently difficult, since it is tricky to provoke all exceptions during tests, and the large number of different exceptions that can happen in a system may lead to *test-case explosion* problems (Myers, 2004; Bruntink et al., 2006).

The exception handling code of AO programs is, therefore, one of the least well understood and tested parts of the system. As a result the exception handling code, which provide means to help developers build robust applications, may become itself a source of bugs and therefore threaten to system robustness.

1.2. Summary of Goals

Given that, the effects of aspect-oriented composition mechanisms on the exceptional control flow are not much understood, our first goal is to conduct and empirical study to discover these effects and their extent in AO programs.

Since the manual analysis of the exception-flow can easily become infeasible (Robillard and Murphy, 1999), before performing the empirical study we had to implement a tool that automatically calculates the *exception paths* (i.e., the path in a program call graph that links the signaler and the handler of an exception) in AO system. The tool should provide a better understanding of the flow of exceptions in AO applications, and also identify possible flaws in the

exception handling code such as *uncaught exceptions* (Miller and Tripathi, 1997; Jo et al., 2004) and exceptions caught by *subsumption* (Robillard and Murphy, 1999). The development of such a tool represents our second goal.

Finally, the lack of approaches to assure the reliability of the exception handling code of AO programs motivated our third goal: the definition of a verification approach for the exception handling code of AO programs, based on static analysis.

Therefore, the goal of this thesis is threefold:

(i) to perform an exploratory study in order to investigate the effects caused by aspects on the exception flow of programs,

(ii) to build a static analysis tool to support the reasoning about the flow of exceptions in AO programs; and finally

(iii) to propose a verification approach for the exception handling code of AO systems based on static analysis.

1.3. Thesis Structure

The remainder of this thesis is structured as follows.

- Chapter 2 reviews some essential concepts concerning aspect-oriented programming (AOP) and exception handling mechanisms, and gives a brief introduction about verification approaches for exception handling code. The works related to this thesis are not described in this chapter. Since this thesis comprises three complementary parts (i.e., an exploratory study; a static analysis tool, and a verification approach) we opted to describe the related works in Chapter 7 after the description of any one of the parts that composes this thesis.
- Chapter 3 describes the empirical study conducted in this work, whose goal is to evaluate the impact of AOP on exception flows of AspectJ programs.
- Chapter 4 first details the results of the empirical study, which includes a catalogue of bugs associated with the exception handling

code. It then provides further discussions and lessons learned concerning the empirical study.

- Chapter 5 presents the supporting ideas and the implementation details of SAFE, the exception flow analysis tool developed in this work to calculate the exception flow information of AspectJ programs.
- Chapter 6 firstly describes a verification approach for the exception handling code based on static analysis. Then it presents one case study of the proposed approach. And finally it discusses the approach's effectiveness based on data collected during the empirical study.
- Chapter 7 presents works we believe are directly related to our own, being distributed in six categories: (i) static analysis tools and techniques; (ii) empirical studies regarding the exception handling code; (iii) AO fault models and bug patterns; (iv) studies regarding aspects interactions; (v) verification approaches for AO software; (vi) collateral effects of aspect libraries reuse.
- Chapter 8 presents the PhD roadmap, offers concluding remarks, pointing to future research work.