7
Related Work

In this chapter we present works that are directly related to our own. We have separated them into 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.

7.1.
Static Analysis Tools and Techniques

Several static analysis tools and techniques have been proposed so far to address problems related to exception handling code in different programming languages. They can be classified in two main categories: (i) machine-directed approaches – techniques developed to be integrated in compilers (e.g. optimization of exception handling (Ogasawara et al., 2001)) or data-flow analysis tools in the presence of exceptions (Sinha and Harrold, 1998); and (ii) human-directed approaches – intended to help developers build robust programs (i.e., reasoning about exceptions (Sinha and Harrold., 2000; Robillard and Murphy, 2003) and detecting potential faults on exception handling code (Bruntink et al., 2006)). Machine-directed approaches are outside the scope of this thesis. Human-directed approaches have been proposed for three languages: ML (Fahndrich et al., 1998; Yi and Ryu, 2002), Ada (Schaefer and Bundy, 1993), and Java (Robillard and Murphy, 2003; Jo et al., 2004; Fu et al., 2005).

In ML, methods do not support the definition of *exception interfaces*, and exceptions are represented as singular values (Garcia and Rubira, 2001) - they are not represented in hierarchical structures. Uncaught exception analysis was first introduced in ML. Fahndrich et al (1998) developed EAT an exception analysis tool on top of BANE, a general framework for implementing constraint-based analysis. Yi and Ryu (2002) developed an exception analyzer, also based on the use of control-flow analysis and a set-constraints framework, but with a different purpose: to detect uncaught exceptions in ML programs. These works focused on

performance and cost tradeoffs involved in the analysis, and did not address how exception flow information could be used for finding faults on the exception handling code such as *unintended handlers.*

Schaefer and Bundy (1993) proposed a tool that calculates the list of exceptions that escape from each method of programs developed in Ada. This approach focused on finding potential faults on exception handling code (e.g., unused handlers); but since exceptions in Ada are not represented in hierarchical structures, problems related to *exception subsumption* were not considered.

Some static analysis solutions were proposed to support the reasoning about the flow of exceptions in Java programs. Robillard and Murphy (2003) developed a tool called Jex that analyses the flow of exceptions in Java Programs. Based on Java source code, this tool performs static analysis in order to find the propagation paths of checked and unchecked exception types. It discovers *uncaught exceptions* and *exception subsumptions* which may represent potential faults. They use the class hierarchy analysis (CHA) to construct the program call graph traversed during the analysis. CHA algorithm yields a less precise call graph when compared to the one used in our approach (as detailed in Section 5.3.2). Moreover, the Jex tool requires the source code of the program to be analyzed, some libraries used by the application may not available during analysis which also affects the tool precision.

Jo et al. (2004) applied the idea of Yi and Ru (2002) to Java programs, to detect too general method exception interfaces (i.e., list of checked exceptions associated with each method) and unnecessary handlers. This approach does not consider the *unchecked* exceptions, and does report the *exception path* for each exception thrown on a system. Fu and Ryder (2005) proposed a static analysis tool, built upon Soot framework for *bytecode* analysis and SPARK a call graph builder to analyze the exception flow on Java programs. This tool generates the *exception paths* to specific exceptions (e.g., IOException) thrown on the system, supports a white-box coverage testing of error recovery code (i.e., the code inside exception handlers) of server applications written in Java based on fault injection. The static analysis tool performs code instrumentation to: (i) guide the fault injection and (ii) record the recovery code exercised. Fu et al. (2007) extended the tool proposed in (Fu et al., 2005; Fu and Ryder, 2005) in order to compute

exception chains (i.e., a combination of semantically-related exception paths). The *exception chains* are capable of representing in one single path the propagation paths of one exception that is wraped in other exception and than rethrown - instead of representing just discrete segments (*exception paths*) of each exception. Such analysis enabled by a *Data-Reachability* analysis algorithm that statically analyses the handled body (catch clause) linking each exception with its root cause, after the exception paths are calculated.

Our approach leverages the previous proposals on exception flow analysis to the analysis of AspectJ programs, which none of them handles. The exception flow analysis solution most similar to ours is the one proposed by Fu et al. (2005). Although, both solutions are built on top of Soot framework for *bytecode* analysis and used SPARK framework to build the program call graph they differ: in *focus* and tool characteristics. The tool proposed by Fu et al. (2005) analyzes the program *bytecode*, however id does not interpret the specific statement added to the Java *bytecode* by the AspectJ weaver. In our tool we have defined a set of heuristics to interpret the code added to the Java *bytecode*. Moreover, while their work supports a white box testing approach (performing code instrumentation, and fault injection) our static analysis tool tries to statically find bugs in the exception handling code (detection of uncaught exceptions and exception *subsumption* and exception handling contract checking). We opt for detecting faults statically instead of through testing approaches since early detection and prevention of faults is less costly (Bruntink et al., 2006) and testing exceptions is inherently difficult (see Chapter 2, Section 2.3.1.1).

7.2. Empirical Studies regarding the Exception Handling Code

Cabral and Marques (2007) performed a quantitative study in which they examined the source code samples of 32 different OO applications, written in Java and .NET. The goal of their study was to identify how exceptions were handled inside each application. They examined the code inside every exception handler and observed that the actions taken by them were usually very simple (e.g., logging and present a message to the user). This work did not consider the exceptions *thrown* inside each application, how the exceptions flow within the

applications (i.e., *exception paths*), nor the number of *uncaught* exceptions, and exception *subsumptions*.

In the context of aspect-oriented software development, some the empirical studies have been conducted to investigate investigated the use of aspects to modularize the exception handling code (Lippert and Lopes, 2000; Filho et al. 2006; Filho et al. 2007).

Lippert and Lopes (2000) performed a seminal study to investigate the use of AO constructs to modularize the exception handling code a large OO framework called JWAM. The goal of this study was to assess the usefulness of aspects for separating exception handling code from the normal application code. The authors observed that when the application adopts a general error handling policy (i.e. does not depends on specific characteristics of application modules), the use of aspects to modularize *exception detection* and *handling* brings several benefits, such as: better code reuse and a consequent decrease in the number of LOC. In their study, they obtained a large reduction in the amount of exception handling code present in the application – from 11% of the total code in the OO version to 2.9% in the AO version.

Castor Filho et al (2006) performed a similar study whose goal was to understand the benefits and limitations of using aspects to modularize the exception handling code in realistic scenarios. In this study, the authors attempted to aspectize the exception handling code of four different applications (i.e., one AO application and three OO applications). This work revealed that the *aspectization* and reuse of exception handlers is not straightforward as advocated beforehand by (Lippert and Lopes, 2000). Instead, it depends on a set of factors, such as: the type of exceptions being handled, what the handler does, the amount of contextual information needed; what the method raising the exception returns; and what the throws clause actually specifies.

Castor Filho et al (2007) elaborated a catalog of best and worst practices related to the *aspectization* of exception handling. This catalog aims at helping developers to decide when they should or should not extract exception handling code to aspects. Such decision is based on the influence of the aspectization on coupling, cohesion, and separation of concerns metrics.

All the aforementioned studies (Lippert and Lopes, 2000; Filho et al. 2006; Filho et al. 2007) aimed at *aspectizing* exception handling constructs, they did not tackle the problems that may arise when exceptions flow from aspect advices. Moreover, even though they pointed out some of the limitations of AspectJ constructs for handling exceptions, they did not assess the error-proneness of AspectJ mechanisms to handle exceptions.

Our empirical study takes into account these issues, neglected by aforementioned studies: (i) the consequences of exceptions signaled by aspects, (ii) the error-proneness of AspectJ constructs for handling exceptions. Furthermore, our work aimed at providing a better understanding of the flow of exceptions in AO applications identifying possible flaws in the usage of aspects in the presence of exceptions (catalogue of *bug patterns*). As a consequence, our study helps programmers to verify the reliability of the exception handling code in AspectJ systems.

7.3. AO Fault Models and Bug Patterns

The new constructs available in aspect-oriented languages, represent also new sources of faults. For that reason works have been proposed aiming at characterizing the new kinds of faults that can happen in AO programs (Alexander et al., 2004; Ceccato et al., 2005; Baekken, 2006; Bækken and Alexander, 2006; Zhang and Zhao, 2007).

Alexander et al. (2004) proposed a candidate fault model which includes a set of fault types related to AspectJ features. Bækken (2006) presents a fine-grained fault model for pointcuts and advice in AspectJ programs based on the work presented by (Alexander et al., 2004). Alexander's fault model was later extended by Ceccato et al. (2005), who characterized faults related to "*incorrect changes in exceptional control flow*". In this work he mentions that the exceptions thrown or softened represent sources of potential faults in AO programs, but they do not detail the potential failures that can derive from them, such as: *uncaught exceptions* and *unintended handler actions* and *obsolete handlers*.

Regarding bug patterns in AO programs, Zhang and Zhao (2007) presented a set of general bug patterns for AspectJ programs. Bug patterns differ from the fault model presented on the previous works, since they represent "*recurring relationship between potential bugs and explicit errors in a program*".

None of these authors mentioned above (Alexander et al., 2004; Ceccato et al., 2005; Baekken, 2006; Bækken and Alexander, 2006; Zhang and Zhao, 2007) tackled the potential problems related to the exception handling code in AO programs (e.g., *unstable exception interfaces*). Moreover, none of these works conducted an observational study to provide evidences of the proposed *bug patterns,* or fault models. The set of bug patterns presented in our work are specifically related to exception handling code in AO software. They represent recurring faults found throughout a fine-grained analysis of a set of AO applications presented in Chapter 3. Furthermore, the above mentioned authors do not detail the consequences the exceptions thrown by aspects in the context of aspect library reuse or propose an approach to deal with them.

7.4. Aspect Interactions

Since aspects crosscuts other concerns, they often exert broad influences on these (e.g. by modifying their semantics, structure or behavior). These dependencies between the aspectual and non aspectual elements of a system may lead to either desirable or (more often) unwanted and unexpected interactions. Some works have been proposed in order to (i) investigate and categorize such dependencies and (ii) devise ways to deal them (Clifton and Leavens, 2002; Katz, 2006; Weston et al., 2007; Rinard, 2004).

Clifton and Leavens (2002) proposed a first categorization of aspects, to support the modular reasoning AO programs. They classified aspects as *Observers* - aspects that do not change any specification of the base module - and *Assistants* - aspects that affect in some way the specification of the base module. They also propose an extension to AO language on which the base modules should explicitly reference the aspects that affect their specifications. This work does not account for the interference between aspects.

Rinard (2004) developed a program analysis system that automatically classifies interactions between aspects and methods. This classification is based in two main factors: (i) the control flow elements that effect how and when the added behavior executes; (ii) the scope (set of fields) accessed by methods and advice. The major contribution of this work is the use of points-to analysis and scope analysis (Salcianu, 2001) to determine which fields aspects and methods modifies.

Katz (2006) provided a categorization of aspects based on classes of temporal properties (e.g., safety, liveness, or existence properties) that are guaranteed to hold for a system with any aspect of a specific category woven into it. According to this classification scheme aspects can be: (i) *spectative* - that only gather information about the system to which they are woven; (ii) *regulative* - that change the flow of control but do not change the computation done to existing fields (e.g., define which methods are activated in which conditions); and (iii) *invasive* - that do change values of existing fields, but still should not invalidate desirable properties. The categories are used as a basis to analyze the interaction between aspects and the base code.

Weston et al. (2007) present an approach for detecting aspect interference using incremental data-flow analysis. Based on the analysis of def-use pairs this approach reveals subtle interactions between aspects (e.g., and aspect modifies a variable that is passed as an argument to a method, which is advised by another aspect). The results of the interference analysis are presented in the form of aspect categories. It categorizes aspects in two broad classifications: (i) interference between aspects and the base code and (ii) interference between two aspects. This categorization is built on the previously proposed categorizations detailed before.

The works presented above focused on interactions between aspects and classes within normal control flow. Most of them aim at providing a program analysis algorithm that automatically recognizes these interaction patterns. However, the interaction analysis proposed by them does not account for the effects of exception occurrences and exception handling constructs. In our work we could observe that new kinds of interaction, between aspects and classes, emerged from the exceptional scenarios (e.g., one class catches one exception thrown by an aspect). Such *Signaler-Handler* relationships between the elements of an AO system, helps on the definition of the exception handling policy of an

AO system and also can be used as a coupling metric between these elements on exceptional scenarios.

7.5. Verification Approaches for AO Systems

The techniques and tools proposed so far to assure the quality of aspect-oriented code mainly focus on: (i) *test-input generation* (Xie and Zhao, 2006); and (ii) *definition of test criteria* (Zhao, 2003; Lemos et al., 2007); and (iii) *test selection* based on branch and interaction coverage, dataflow coverage (Xie et al., 2006), and mutation testing (Anbalagan and Xie, 2006).

Xie and Zhao (2006) proposed Aspectra a test generation framework that leverages an existing OO test-generation tool to generate integration tests for the woven classes. Aspectra generates the test input but does not define a test oracle. Therefore, manual effort is still needed to inspect the executions of the selected tests in order to verify weather the test failed or succeeded. This work was later extended by Xie et al (2006). They proposed Raspect, a framework for detecting redundant unit (i.e., advised methods, advice, and intertype methods) tests for AspectJ programs. This framework detects redundant tests that do not exercise the new behavior. Therefore it selects only non-redundant tests from the automatically generated test suites, thus allowing the developer to spend less time in inspecting this reduced set of tests.

Anbalagan and Xie (2006) proposed a pointcut mutation testing tool, whose goal is to inject faults into an existing program, and check whether the test suite is sensitive enough to detect the injected fault. The tool proposed by Anbalagan and Xie works in three stages. First it identifies join points that are matched by a pointcut expression, generates mutants (i.e., variations of the pointcut expression that resemble closely the original one) of this pointcut expression. Next, it identifies join points that are matched by these mutants. The mutants' matched join points are then compared with those of the original pointcut and these mutants are classified as different types of mutants for selection.

Zhao (2003) adapted the OO data-flow testing approach proposed by (Sinha and Harrold., 2000) for the test of AspectJ programs. When Zhao's testing approach was proposed, the AspectJ weaver combined aspects and classes by

inlining the pieces of advice in the affected join points - aspect and non-aspect code was mixed without explicit reference to the aspects. As a consequence, this testing approach considers clusters of aspects and classes as the units to be tested – the advice code can only be tested when combined to the code affected by it.

Lemos et al (2007) defined a family of control flow and data flow based testing criteria for the structural test of AspectJ programs. They propose the derivation of a control and data flow models for aspect-oriented programs based on the static analysis of the woven *bytecode*. Using this model, called aspect-oriented def-use graph (AODU), is used as the basis to define aspect-oriented testing criteria. The testing criteria proposed in this work were in the JaBUTi/AJ testing tool.

The techniques and tools presented above focus on the normal control flow of programs. They do not account for the exceptions signaled and handled inside the system, and consequently does not propose a way of assuring the reliability of the exception-handling code of AO systems. In our approach we opt for assuring the reliability of the exception handling code though an approach based on static analysis because it is difficult to simulate the exceptional conditions during tests, and the huge number of different exceptions that can happen in a system may lead to *test-case explosion* problem as mentioned before.

7.6. Collateral Effects of Aspect Libraries Reuse

As mentioned before aspect libraries are a relatively new artifact and best practices for their development and reuse still need to be explored in detail. Although, they enable the reuse of typical crosscutting concerns (e.g., monitoring, transaction management) they can also bring new challenges to AO software development. So far, initial work has been developed which investigate the problems related to library aspects reuse (McEachen and Alexander, 2005; Apel et al., 2006; Lopez-Herrejon et al., 2006).

These works discuss unanticipated aspect composition problems in the context of incremental software development, such as: an aspect may affect subsequent integrated elements even though they were implemented without being aware of them. Such unanticipated aspect compositions can lead to unpredictable effects and errors.

However, these works do not tackle the problems that may arise when exceptions flow from re-used. In our work we investigated the collateral effects of reusing aspects in the presence of exceptions. Moreover, although some problems related to aspect libraries reuse are similar to the ones associated with OO libraries reuse, we have shown that some characteristics of AO compositions aggravated the problems (see Chapter 6, Section 6.1.1).

7.7. Summary

In this chapter we presented the works we believed are directly related to the work presented here. Since our work can be divided in three main parts (i) the exploratory study, (ii) the exception-flow analysis tool for AspectJ programs, and (iii) the verification approach for exception handling code, the related work were presented in categories related to each one of them.

Next chapter concludes the work presented in this dissertation, summarizing the main contributions of this thesis and pointing directions for future work.