

On the Robustness Assessment of Aspect-Oriented Programs*

Roberta Coelho¹, Otávio Augusto Lazzarini Lemos², Fabiano Cutigi Ferrari²,
Paulo Cesar Masiero², Arndt von Staa³

¹Informatics and Applied Mathematics Department, Federal University of Rio Grande do Norte – UFRN, Natal, Brazil

²Computer Systems Department, University of São Paulo – USP, São Carlos, Brazil

³Informatics Department, Pontifical Catholic University of Rio de Janeiro – PUC-Rio, Rio de Janeiro, Brazil

roberta@dimap.ufrn.br, {oall,ferrari, masiero}@icmc.usp.br, arndt@inf.puc-rio.br

Abstract

Empirical studies have shown that Aspect Oriented Programming (AOP) promotes the modularity and design stability of systems in the presence of crosscutting concerns. However, as important as measuring the impact of AOP on the design stability is to measure its impact on system robustness. The effective adoption of a new technology might be impaired if it makes programs less resilient to internal and external unexpected behavior. In this paper we propose an approach that explores the synergy between static and dynamic analysis to assess the impact of AOP on system robustness with regard to exception handling (EH). This impact is quantified in terms of a set of exception mechanism related metrics which are collected (i) during static checks of EH contracts, and (ii) during the structural testing of EH code.

Keywords Aspect Oriented Programming, assessment, metrics, static analysis, structural testing.

1. Introduction

Empirical studies [7] have shown that AOP promotes modularity and design stability of systems in the presence of crosscutting concerns. The design stability encompasses the sustenance of system modularity properties and the non-observance of ripple effects in the presence of changes. In order to assess the various facets of design stability and modularity such works defined and collected a set of metrics such as, cohesion, change propagation [8], concern interaction analysis [8] and identification of architectural ripple effects [8].

However, as important as measuring the impact of a new technology on the system modularity and design stability is to measure its impact on the system robustness – the ability a system has to respond to faults. A new technology might render less useful for practical purposes if it makes the system less resilient to system or resource faults. Therefore, there is a need to define procedures and metrics to assess the impact of new modularization techniques, such as AOP, on software robustness. Such metrics should provide means of evaluating whether the system provides reasonable

response to system (or resource) problems in order gracefully degrade, rather than experience application crashes.

Exception handling (EH) is a widely used mechanism for building robust systems. It allows the system to detect errors and respond to them correspondingly through the execution of recovery code encapsulated into exception handlers. Its importance is indicated by the embedding of EH specific constructs (e.g., `try-catch` blocks) in mainstream programming languages such as Ada, Java, and C#.

Although the goals of exception handling mechanisms are to make programs more reliable and robust, the integration of exception handling mechanisms with new modularization techniques raises unique issues. As a consequence, the code dedicated to improve the robustness may bring the opposite effect: it can become itself a significant source of system failures.

Although AOP may be used to promote the modularity and reuse of EH code [5] – i.e., the recovery code can be encapsulated in pieces of advice - we could observe in previous empirical studies [16][4] that AOP may also increase the error-proneness of the EH code. Some negative observations of these studies were: (i) higher evidence of uncaught exceptions when aspect advices act as exception handlers, thereby leading to unpredictable system crashes; and (ii) a multitude of exception subsumptions [11], some of them leading to *unintended handlers*, i.e., exceptions that are thrown by aspects and unexpectedly caught by existing handlers in the base code.

The assessment of such exception-related metrics were based on the use of an exception flow analysis tool, called SAFE [4]- that statically analyzed the bytecode in order to discover the exception flows – and on manual inspections, during which *infeasible exception flows* were detected.

Unfortunately this static-based assessment neither reveals whether the error recovery code encapsulated into handlers is ever exercised - such code may be protected by an `if` clause which is always *false* – nor checks whether it takes the appropriate action. Such faults of the EH code cannot be detected during static analysis, since they depend on runtime conditions, e.g., specific input that may exercise the faulty paths. The white-box testing of error recovery code can aid in such task [3]. On the other hand, applying structural testing to the whole system without previous static analysis can be too expensive, since several test cases might be required, depending on the testing criteria adopted.

* The authors are financially supported by CNPq (under grant 477425/2008-9) and FAPESP (under grants 2008/10300-3 and 05/55403-6).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

```

public aspect VendingMachineEAspect {
    public pointcut vendExec():
        execution(public void VendingMachine.vend(..)
            throws ZeroValueException, IllegalAmountException);

    public pointcut vend(VendingMachine v, int sel) :
        vendExec() && this(v) && args(sel);

    declare soft : SelectionException : vendExec();

    void around(VendingMachine v, int sel) : // UNIT A
        vend(v, sel) {
        try {
0         proceed(v, sel);
9-16        } catch (SelectionException e) {
21            v.currAttempts--;
25            if (v.currAttempts <
                VendingMachine.MAX_ATTEMPTS)
29                System.out.println("Enter selection again");
            else {
40                if (e instanceof
40                    IllegalSelectionException)
48                    System.out.println("Illegal " +
48                        "Selection! Transaction aborted.");
                    else
59                    System.out.println("Invalid " +
59                        "Selection! Transaction aborted.");
67                try { v.returnCoins(); }
                catch (ZeroValueException e1) {
74                    e1.printStackTrace();
                }
            }
        }
        ...
    }

    public class VendingMachine {
        ...
        static final private int INSERT = 0, VEND = 1,
        RETURN = 2, CHECKVALUE = 3, TURNOFF = 4;
        static final int MAX_ATTEMPTS = 3;

        public void returnCoins() // UNIT D
            throws ZeroValueException {
0         if (currValue == 0)
7-17        throw new ZeroValueException();
15        System.out.println("Coins returned");
            currValue = 0;
            currAttempts = 0;
        }

        public void vend(int selection) // UNIT B
            throws ZeroValueException, IllegalAmountException {
0         if (currValue == 0)
7-14        throw new ZeroValueException();
15        d.dispense(currValue, selection);
27        int bal = d.value(selection);
            totValue += currValue - bal;
            currValue = bal;
            System.out.println("Item dispensed");
35        returnCoins();
        }
        ...
    }

    public class Dispenser {
        ...
        public void dispense (int currVal, int sel) // UNIT C
            throws SelectionException, IllegalAmountException {
0         SelectionException se = null;
0         IllegalAmountException iae = null;
0-10        if (sel < MIN_SEL || sel > MAX_SEL) {
16            se = new IllegalSelectionException();
        } else {
54-59        if (!available(sel)) {
62-96        se = new SelectionNotAvailableException();
        } else {
100         int val = value(sel);
105         if (currVal < val)
113             iae = new IllegalAmountException(val-currVal);
        }
126        if (se != null)
130            throw se;
132        if (iae != null)
137            throw iae;
140        }
    }
}

```

Figure 1. Partial AspectJ code example with Exception Handling.

In this context, the contributions of this paper are as follows: (i) we present a framework for assessing the robustness of AO programs with regard to exception handling (EH) (Section 3); this framework explores the synergy between static and dynamic analysis (i.e., white-box testing) (Section 3.3), including new metrics that were not used in the previous empirical studies [4][16]; (ii) in addition to the metrics suite, we define a corresponding measurement approach which was instantiated for AspectJ language (Section 3.4). In this paper we have discussed the usefulness of the measurement framework, by applying it in an extended version of the Vending Machine program [12]. The preliminary results of this case study together with the results of previous empirical studies (which used a subset of the metrics defined on this framework to access the robustness of the EH code of three medium sized AO programs [4]) have show that this framework can be used to predict the robustness of the AO code.

2. Exception Handling in AO Programs

In an AO program, an exception may be raised by a method or by an advice whenever an abnormal computation state is detected. The *exception signaler* is the element that detects the abnormal state and raises the exception. For instance, in the AspectJ code example shown in Figure 1, the *dispense* method of the *Dispenser* class detects abnormal conditions – illegal or unavailable selections, or an illegal amount – and raises instances of *SelectionException* or *IllegalAmountException* subtypes of

SelectionException; the *dispense* method is an exception signaler.

After a method or an advice signals an exception, the runtime system attempts to find the block of code that will be responsible for handling it (i.e., the *exception handler*). In AO programs, an exception handler can be defined either within a method or within an advice (hereafter also referred to as a *unit*). The *exception path* is a path in a program call graph that links the signaler and the handler of an exception.

When the exception handler is defined within an advice it has the ability to handle exceptions raised by any join point in the program – regarded that this join point is defined on the pointcut expression associated to it. This is possible because specific types of advice (e.g. around advice) wraps the advised join points and gets access to the exceptions that flow from it. The *Error Handling Aspect* [11] is the aspect that contains one or more advices responsible for handling exceptions raised by other join points in the base code or in other aspects. For instance, in the code presented in Figure 1, the around advice of the *VendingMachineEAspect* intercepts the *vend* method and catches *SelectionException* instances that can flow from the call to the *dispense* method made inside the *vend* method. Note that the handler is supposed to implement the following rule: if the user mistakenly selects an invalid or illegal selection for more than three times in a role – the value of *MAX_ATTEMPTS* – the transaction should be aborted (lines annotated with 48 and 59).

Table 1: Suite of Exception-Related Robustness Metrics

Attribute	Metric	Description
Exception Flow Size	Exception Paths (EP)	The number of exception paths found on a program. We call exception path a path in a program call graph that links the exception signaler of an exception instance (the unit that contains a throw statement) and the handler of such exception (the unit that contains the catch clause that handles it).
Exception Handling Coupling	Exceptions Signaled on Aspect-advice and Handled on Aspect-advice (AA)	The number of exception paths in which an exception instance is signaled by an Aspect (advice, or declare soft construct) and handled by an Aspect (advice, or declare soft construct).
	Exceptions Signaled by an Aspect and Handled by a Class (AC)	The number of exception paths in which an exception instance is signaled by an Aspect (advice, or declare soft construct) and handled by a Class (method)...
	Exceptions Signaled by a Class and Handled by an Aspect (CA)	The number of exception paths in which an exception instance is signaled by a Class (method) and handled an Aspect (advice, or declare soft construct)..
	Exceptions Signaled by a Class (ESC) and Handled by an Aspect (CC)	The number of exception paths in which an exception instance is signaled by a Class (method) and handled by a Class (method)..
Handler Type	Uncaught Exception (UE) - Aspect-signaled (AUE) - Class-signaled (CUE)	The number of exception paths in which the exception instance is not caught inside the system (UE). - When the signaler of such exception instance is an Aspect (AUE). - When the signaler of such exception instance is a Class (CUE).
	Specialized Handling (SpH) - Aspect-signaled (ASpH) - Class-signaled (CSpH)	The number of exception paths in which an exception instance is caught by a handler whose type is the same type of the exception type being caught. - When the signaler of such exception instance is an Aspect (ASpH) - When the signaler of such exception instance is a Class (CSpH)
	Exception Subsumption (ESu) - Aspect-signaled (AESu) - Class-signaled (CESu)	The number of exception paths in which an exception instance is caught by a handler whose type is a supertype of the exception type being caught. - When the signaler of such exception instance is an Aspect (ASpH) - When the signaler of such exception instance is a Class (CSpH)
Exception Handling Code Complexity	Cyclomatic Complexity of Signalers (CCS)	The cyclomatic complexity of all signalers present in the system.
	Cyclomatic Complexity of Handlers (CCH)	The cyclomatic complexity of all signalers present in the system.
Exception Handling Testing	Rate of successful Exception Handling Test Cases (ST)	The number of successful test cases over the total number of test cases. Note that the test set considered has to target the execution of exception handling constructs.
	Coverage of Exception Handling Constructs (CEH)	The percentage of covered EH-related test requirements by test cases that target exception handling constructs.
	Dynamic Robustness (DR)	Combines the coverage of EH code (CEH) with the rate of successful test cases (ST) for a given AO program and a given test set. ST is given a higher weight, since it tells more about the robustness of the system than the coverage considered by itself. DR is computed as: $CEH + (2 * ST)$.

3. A Robustness Measurement Framework

This section aims at defining a measurement framework for assessing robustness of AO and OO programs. It complements existing AO assessment metrics by explicitly dealing with the error proneness of EH code. The framework aims at supporting the software engineers to: (i) diagnose robustness problems caused by the exception handling concern, and (ii) compare design solutions (e.g., OO and AO) with respect to characteristics of the exception handling code.

This framework includes a set of metrics related to characteristics of exception paths, and metrics related to the error recovery code itself (encapsulated into handlers and signalers). Table 1 presents a summary of such metrics suite with a brief definition for each one.

3.1 Exception Path Related Metrics

In this framework each *exception path* is classified in three dimensions as detailed in Figure 2. Each axis represents one property of the exception path for which a set of values are permitted. The first axis specifies the *exception signaler*, which can be: a class method, an advice, or a declare soft construct. The second axis defines the exception handler associated with the exception path. The exception can be handled by a class method, by after and around advice [5] or by the declare soft AspectJ-specific construct. The third axis represents the

handler action. An exception occurrence can be caught in two basic ways: (i) it can be caught by a specialized handler - when the catch argument is the same type of the caught exception type; and (ii) it can be caught by subsumption - when the catch argument is a supertype of the exception being caught. When no handler is defined for an exception it remains uncaught in Java and AspectJ programs such exceptions will transparently propagate back to the program entry point, causing the Java virtual machine to terminate – the system crashes unpredictably. The metrics proposed on our assessment framework combines the exception path properties as detailed in Table 1.

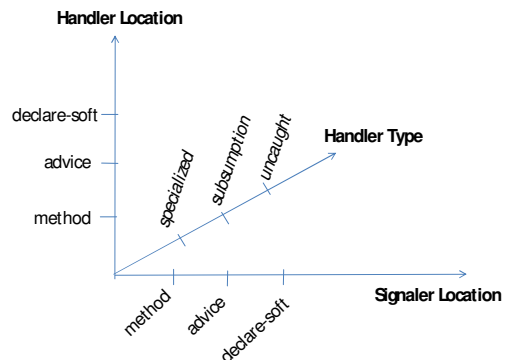


Figure 2. Three-dimension classification for exception paths.

3.2 Exception Handling Testing Metrics

We have also defined a set of metrics to be collected during white box testing. Such metrics aims at assessing the correctness of the logic of complex exception handlers and signalers (called *units* from now on). By *complex* we mean units that contain more than a single possible executable path (*i.e.* units that contain at least one conditional command – *if*, *for*, *while*, etc.). The dynamic assessment is done by means of the execution of test cases, which must exercise the logic of the EH structure of an AO program. We measure the dynamic robustness of an AO program in terms of three metrics: (i) the number of successful test cases that exercise exception handlers (ST), (ii) the coverage of the EH structure obtained by the test cases (CEH), and (iii) a combination of the rate of successful test cases with the attained coverage – called dynamic robustness (DR). ST and CEH are simpler metrics so we restrict our discussions on the combined DR metric.

Coverage has to be taken into account because it is possible to create several successful test cases that only exercise a small part of the EH structure of the program. On the other hand, coverage should not be considered by itself because it is possible to cover 100% of the EH code with failing test cases, which would indicate a lack of robustness in the program. Also, the number of successful test cases – and consequently the number of failing test cases – should be assigned a higher weight in the combined metric since they tell more about the robustness of a system. A low coverage might mean only that the available test set is poor, and thus exercises few parts of the EH code.

The Dynamic Robustness (DR) metric is then calculated as follows: $Co + (Su * 2)$, where Co is the coverage percentage of the EH code, and Su is the percentage of successful test cases over the total number of test cases of the available test set (by *successful* we mean test cases that execute without failing or resulting in error). For instance, if a test set covers 80% of the EH structure of an AO program and contains 2 successful test cases of a total of 4, DR will be evaluated to: $0.8 + (2/4 * 2) = 1.8$. It is obvious that failing test cases would not remain in the system, but while the program is still not fixed, DR provides a measure of robustness based on the current test set. Note that DR varies between 0 and 3, and a higher number means better robustness.

DR must also be computed under two conditions:

1. The test set must only contain non-redundant test cases, that is, every test case must enhance the coverage of the EH code of the program. This is done to prevent increasing the DR when multiple test cases succeed but exercise coincident parts of a handler;
2. The test set considered refers only to test cases that exercise handlers.

A low DR might in some cases indicate a poor test set, that exercises few parts of the EH code, and not necessarily low robustness of the AO program. This is reasonable because the test set should also be taken into account while measuring the robustness of a program: if important handlers of the program are not being exercised, their behavior is unknown and so is the system robustness.

3.3 The Assessment Approach

This section introduces an approach to assess exception handling code in OO and AO software, comprised by two major phases:

1. *static analysis phase* which contains a set of steps responsible for discovering and classifying the exception paths (see steps 1-3 in Figure 3).
2. *dynamic analysis phase* which is performed by means of structural testing, in which partial models of the system are built in order to represent the exception-dependent execution flows (see steps 4-6 in Figure 3).

During the first steps of our approach (steps 1-2) the exception paths are discovered and the exception path related metrics detailed in Table 1 are calculated. This phase relies on an extended version of the program call graph (EHPCG) which contains additional information in each node. Such information comprises the statements where exceptions may be thrown and handled (by an enclosing try-catch block). Based on the EHPCG, every statement that may throw an exception can be identified, and for each of them, it traverses the program call graph backwards looking for handlers capable of catching it. If no handler is found, the algorithm reaches the program entrance point and the exception is classified as an uncaught exception.

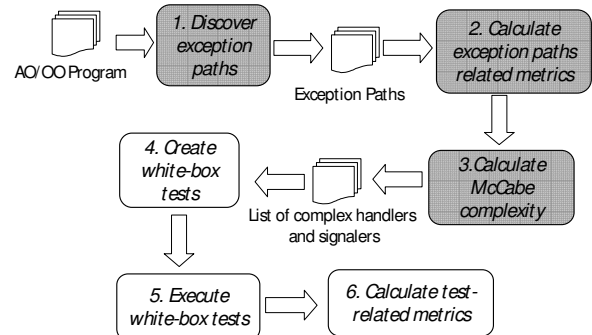


Figure 3. The Assessment Approach.

The second phase of our approach (steps 4-6) relies on control and data flow underlying models aiming at improving the reliability of the exception-dependent code. In short, it consists in building a model that includes parts of an AO system where exceptions flow through. This partial representation of the system, called Exception-Aware Control Flow Graph (EACFG) consists of an inter-procedural control flow graph constructed for the whole system that focuses on the exception-related logic.

The EACFG graph is constructed in the following way: initially a graph with all dependencies among pieces of advices and methods (the units) is constructed. In this graph, a node represents a unit and an edge represents a method call or the flow of execution to an advice. Then, an analysis is performed to detect which units throw or handle exceptions in the program. The nodes that represent these units are expanded with the actual internal control flow involved. However, if the control flow of an exception-related unit is simple, that is, it contains only a single possible path, it is not expanded. We do this because signalers and handlers with simple control flow can be

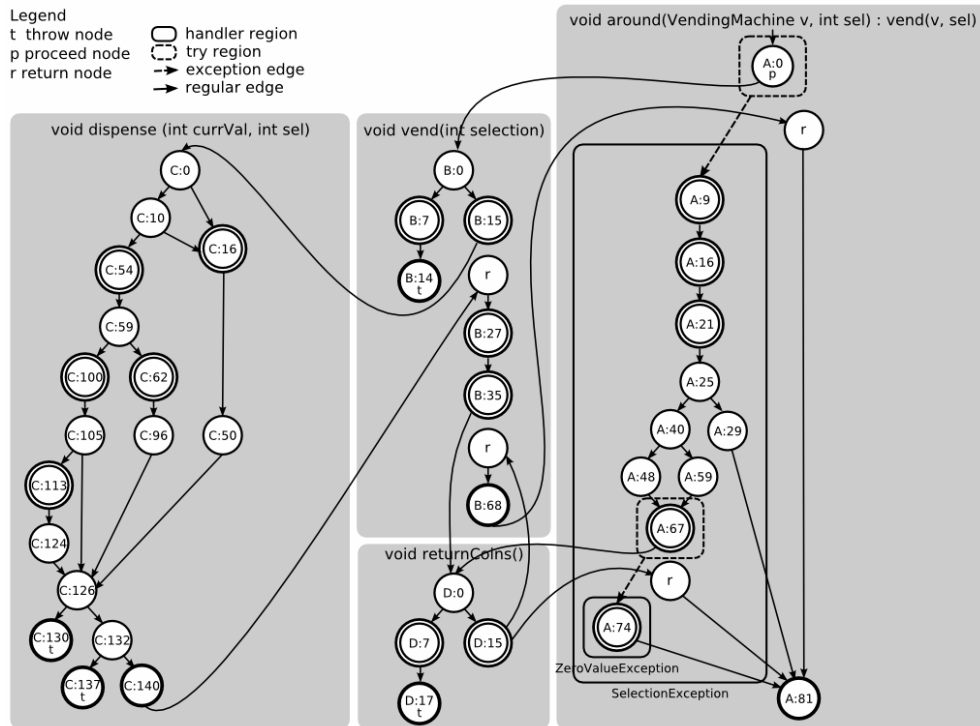


Figure 4. Example of an EACFG.

covered with a single test case, since there is only one possible path to be executed. The intermediary units and the ones that include signalers or handlers with simple logic appear in the graph as a single node, as they were represented originally.

The EACFG allows for a clearer view of the exception-dependent parts of the system when compared to a full inter-procedural control flow representation. Once an EACFG is defined, it can be used as a basis for the definition of test EH-based testing criteria.

Figure 4 present an EACFG generated for the partial AspectJ code presented in Figure 1. Nodes are labeled with a sequential letter for each unit (see Figure 1), followed by a semicolon and a number. The number corresponds to the bytecode offset of the first instruction represented by that node. We use such representation because our implementation of the testing part of the framework is based on Java bytecode. Nevertheless, the graph could also be generated from the source code.

3.4 Application Example

Consider the AspectJ code example presented in Figure 1. By applying our framework, the first step – the static analysis – would detect the unhandled `ZeroValueException` instance that might be thrown by the `returnCoins` method, when it is called from the `vend` method. Note that `vend` re-throws the exception, as well as the other operations that use it (not shown in the figure).

For the second step of our approach, the static analysis would point the `SelectionException` handler contained in the `around` advice as a complex handler that should be tested to enhance DR. At first, the static analysis would not detect any problems with the `SelectionException` instances that can be thrown by the `dispense` method,

because they are handled by the `SelectionException` handler present in the `around` advice. However, while trying to cover nodes `A:40`, `A:48`, `A:59`, `A:67` of the corresponding graph (Figure 4), that is, the case where the user fails more than three times by selecting invalid or unavailable selections, the tester reveals a fault: these nodes cannot be covered by any inputs, they are unfeasible. This fault is caused by the line annotated with 21 in Figure 1, `currAttempts` should be increased every time an exception is caught, and not decreased. The JUnit test case presented in Figure 5 reveals the fault (the `operate` method issues a sequence of commands to the `VendingMachine`). For space reasons we do not make a full analysis of the other metrics in this paper.

```
public void testAbortTransactionIllegalSel() throws
    ZeroValueException, IllegalAmountException {
    Integer[] ops = {0, 10, 1, 350, 1, 350, 1, 350};
    vm.operate(ops);
    assertTrue(allOutput.toString().contains(
        "Illegal Selection! Transaction aborted."));
}
```

Figure 5. Fault-revealing test case.

3.5 Implementation of the Assessment Framework for AspectJ Programs

We have been working on the implementation of this assessment framework for AspectJ programs. We have implemented the SAFE tool for the static exception flow analysis of AspectJ programs; this tool, based on Soot framework, creates the EHPFG and calculates the exception paths [4]. We are currently extending this tool to automatically calculate the exception-path related metrics that should be collected on steps 2 (see Figure 3). To support steps 3 to 6 we are currently extending JaBUTi/AJ tool [3]. JaBUTi/AJ (Java Bytecode Understanding and

Testing - for AspectJ) automates the creation of control- and data-flow graphs of AO programs and a set of associated testing criteria.

4. Related Work

Assessment of Exception Handling Code. In a previous work [4] we assessed the error proneness of the exception handling code of AO programs based on the number of uncaught exceptions, specialized handlers and exception subsumptions. In the present work we focus on the definition of a metric suite that details such metrics, and includes additional statically-collected and testing-based metrics. In [19], Cacho et al. aimed at measuring the number of uncaught exceptions in an exploratory study whose main goal was to compare three different techniques for modularizing EH behavior. Taveira et al. [20] also evaluated three different approaches for modularising EH behavior. Differently from Cacho et al., however, they applied a set of size and separation of concerns (SoC) metrics focusing on measuring the reuse of EH-specific code. Their size metrics are statically collected and include, for example, the number of EH-related elements (`try-catch-finally` blocks), the number of LOC implementing the EH concern, and the number of blocks of code responsible for handling exceptions. Our goal differs from the aforementioned authors' goals as we aim at assessing the robustness OO and AO applications with respect to EH, by means of static and dynamic analyses. Moreover, our static metrics focus on the flow of exceptions through the system and on the internal complexity of exception signalers and handlers.

Test-related Metric Suite. Nagappan [2] proposes a software testing metric suite to be used as an *early warning* indicators of program reliability. Such suite is based on the fact that measures derived from the assessment of system behavior (e.g., the number of defects found in test) have been shown to be useful as early indicators of externally-visible product quality [1]. The Nagappan's work does not consider the exceptional control flow during tests. As a consequence, the author did not take into account whether or not EH criteria were addressed by a set of the test cases. In our work we focus on exception-related testing metrics neglected by this aforementioned work.

Checking the Reliability through Testing. Sinha and Harrold [12, 13] define a set of structural-based test criteria, which requires the execution of the exception-related structure of a program, such as paths that cover a definition of an exception object and its subsequent use inside a catch block. The use of such criteria while performing the white-box testing the EH code of a medium-sized program may become prohibitive due to the huge number of test-cases that should be implemented. In our approach we do not face such test case explosion problem since we only consider pieces of advice that contains EH. Thus, instead of creating a complete interprocedural graph to represent the whole system, we proposed a partial representation that only considers the EH code encapsulated on signalers and handlers.

Checking the Reliability of EH Code through Static Analysis. Static analysis tools have been proposed to discover the exception paths of OO programs [6], [16] and check whether the recovery actions are executed in all exceptional scenarios [9], however none of such *tools* are

able of analyzing AspectJ programs nor are able of calculating the exception path related metrics presented in Table I.

5. Concluding Remarks

This paper presented a measurement framework consisting of the following: (i) a suite of metrics related to the EH code, collected statically and dynamically (through a white box testing technique), and (ii) a proposed approach for collecting such metrics. This paper also discusses how such approach can be implemented for AspectJ language, through the extensions of existing static analysis and testing tools, SAFE [4] and Jabuti/AJ [3] respectively. We are currently working on the tools extensions and conducting other case studies in order to refine the measurement framework presented here.

References

- [1] V. R. Basili, L. C. Briand, W. L. Melo. 1996. A Validation of Object-Oriented Design Metrics as Quality Indicators, IEEE Transactions on Soft. Engineering 22(10), 751-761.
- [2] N. Nagappan. 2004. Towards a Software Testing and Reliability Early Warning Metric Suite, In: ICSE 2004, 60-62.
- [3] O. A. L. Lemos, I. G. Franchin, and P. C. Masiero. 2009. Integration Testing of Object-Oriented and Aspect-Oriented Programs: A Structural Pairwise Approach for Java. Science of Computer Programming 74(10), 862-882.
- [4] R. Coelho, et al. 2008. Assessing the Impact of Aspects on Exception Flows: An Exploratory Study. In: ECOOP 2008, 207-234.
- [5] F. Castor, A. Garcia, and C. Rubira. 2007. Extracting Error Handling to Aspects: A Cookbook. In: ICSM 2007, 134-143.
- [6] C. Fu and B. G. Ryder. 2007. Exception-chain Analysis: Revealing Exception Handling Architecture in Java Server Applications. In: ICSE 2007, 230-239.
- [7] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa. 2005. Modularizing Design Patterns with Aspects: A Quantitative Study. In AOSD 2005, 3-14.
- [8] P. Greenwood, et al. 2007. On the Impact of Aspectual Decompositions on Design Stability: An Empirical Study. In: ECOOP 2007, 176-200.
- [9] S. Thummalapenta, and T. Xie. 2009. Mining Exception-Handling Rules as Sequence Association Rules. In: ICSE 2009, 496-506.
- [10] G. Kiczales, et al. 1997. Aspect-Oriented Programming. In: ECOOP 1997 (LNCS, vol. 1241), 220-242.
- [11] M. P. Robillard, and G. C. Murphy. 2003. Static Analysis to Support the Evolution of Exception Structure in Object-Oriented Systems. ACM Transactions on Software Engineering and Methodology, 12(2), 191-221.
- [12] S. Sinha, and M. J. Harrold. 1998. Analysis of Programs with Exception-Handling Constructs. In: ICSM 1998, 348-357.
- [13] S. Sinha and M. J. Harrold. 2000. Analysis and Testing of Programs with Exception Handling Constructs. IEEE Transactions on Software Engineering 26(9), 849-871.
- [14] S. Soares, P. Borba, and E. Laureano. 2006. Distribution and Persistence as Aspects. Softw. Pract. Exper. 36(7), 711-759.
- [15] J. Viegas, and J. Voas. 2000. Can Aspect-Oriented Programming Lead to More Reliable Software? IEEE Software, 17(6), 19-21.
- [16] R. Coelho A., U. Kulesza, A. Staa, A. Rashid, and C. Lucena, Unveiling and Taming Liabilities of Aspects in the Presence of Exceptions: A Static Analysis Based Approach, Journal of Systems and Software, 2009. (to appear)
- [17] R. Miller e Triphati Issues with Exception Handling in Object-Oriented Systems. ECOOP'97, 1997, p.85-103.
- [18] C. Fu, A. Milanova, B. Ryder., and D. Wonnacott. 2005. Robustness Testing of Java Server Applications. IEEE Transactions on Software Engineering, 31(4), 292-311.
- [19] N. Cacho, F. Neto, A. Garcia, F. Castor. 2009. Exception Flows made Explicit: An Exploratory Study. In: Brazilian Symp. on Softw. Engineering (SBES).
- [20] J. C. Taveira, et al. 2009. Assessing Intra-Application Exception Handling Reuse with Aspects. In: Brazilian Symp. on Softw. Engineering (SBES)