How Exception Handling Constructions are Tested: An Initial Investigation with Open Source Software

Auri Marcelo Rizzo Vincenzi
Instituto de Informática
Universidade Federal de Goiás, UFG
Goiânia-GO, Brazil
e-mail: auri@inf.ufg.br

João Carlos da Silva
Instituto de Informática
Universidade Federal de Goiás, UFG
Goiânia-GO, Brazil
e-mail: jcs@inf.ufg.br

Plínio de Sá Leitão-Júnior
Instituto de Informática
Universidade Federal de Goiás, UFG
Goiânia-GO, Brazil
e-mail: plinio@inf.ufg.br

José Carlos Maldonado
Inst. de Ciências Matemáticas e de Computação
Universidade de São Paulo, USP
São Carlos-SP, Brazil
e-mail: jcmaldon@icmc.usp.br

Márcio Eduardo Delamaro
Inst. de Ciências Matemáticas e de Computação
Universidade de São Paulo, USP
São Carlos-SP, Brazil
e-mail: delamaro@icmc.usp.br

Marcos Lordello Chaim
Escola de Artes, Ciências e Humanidades
Universidade de São Paulo, USP
São Paulo-SP, Brazil
e-mail: chaim@usp.br

***Abstract*—Software testing is one of the most important activities in software development to deliver quality to the final product. Aiming at high efficacy, high quality and a low-cost testing strategy, several testing techniques and criteria have been proposed in the last decades. In particular, structural testing techniques are among the most popular. The authors have extended traditional structural testing in order to meet this requirement, allowing its application to a software with exception handling structures to assess the coverage measurement of such structures. In this paper, we present control- and data-flow criteria to exercise such structures and then evaluate four well-known open source software projects according to these criteria. The results show that test cases for those software achieved low coverage of exception handling code and normal execution code as well. The work also shows that using test criteria which discriminate between exceptional and normal testing requirements might be useful to produce a better degree of information about the test set evaluated.**

Keywords-software engineering; testing criteria; structural testing; code coverage; testing tools

I. Introduction

The exception handling mechanism available in a variety of languages brings improvements on how to deal with error handling or special conditions to product implementation. Instead of using the traditional return value for error indication, exceptions provide a more sophisticated approach for error handling. Despite its benefits, the use of exceptions brings additional challenges to system verification and validation.

By complementing other verification and validation techniques, like technical revision and formal methods, software testing enhances productivity and provides evidence of the reliability and the quality of the product. In addition, testing artifacts can be valuable information to other software engineering tasks, like debugging and maintenance.

Structural testing determines testing requirements from program source code. In general, structural testing criteria use a program representation known as def-use graph that abstracts the flow of control and variable usage of the program under testing. This paper describes a set of structural testing criteria for programming languages with exception handling mechanism. The underline control- and data-flow model is defined to represent such criteria and a tool which supports the model and implements the testing criteria instantiated for Java is described.

A set of Open Source Software (OSS) projects was evaluated in a large international project, aiming at encompass metric definition, measurement practices, data analysis, test suite definition, performance benchmarking, and indicator computation [1]. We applied structural testing to such projects to assess the quality of the available test sets. Some of these projects are employed in this paper to illustrate how OSS have been using exception handling constructions and how well their test sets exercise such structures.

The paper is organized as follows. In Section III, the exception handling mechanisms of Java language are described; Section IV presents the set of control- and data-flow based criteria we have extended to deal with exception handling constructions. In Section V, we present the data collected from four OSS projects, drawing a picture about the usage of exception handling constructions in those projects and how their OSS communities develop test cases for covering such

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9

pieces of code. In Section VI, we offer our conclusions and future work.

II. Related Work

Aberdour [2] compares close- and open-source software quality assurance and quality control, enumerating eleven differences. In the context of our work, four of them have a great impact since, according to Aberdour [2] in the OSS software development 1) the development methodology often is not defined or documented; 2) the testing and quality assurance methodology is unstructured and informal; 3) the defect discovery occurs from black-box testing late in the development process; and 4) the empirical evidences regarding quality are not collected. In one of the proposed guidelines to improve OSS development process, Aberdour [2] mentioned that the user based system testing should be complement with formal testing techniques and regression testing automation.

Considering specifically the testing process on OSS community, Zhao and Elbaum [3] conducted a survey with 200 OSS and found that instead of focusing on high quality milestone releases, the "release early, release often" process, traditionally adopted by the OSS community, results in a continual improvement by a large number of developers contributing iterations, enhancements, and corrections. With respect to the way OSS community test their software, Zhao and Elbaum [3] discovered that: 1) testing effort is concentrated on system testing; 2) fewer than 20% of OSS developers use test plans; 3) only 40% of projects use testing tools, but this percentage increases in case of Java, which has several available tools; 4) less than 50% of OSS use coverage concepts or tools to improve test quality.

A more recent study from Khanjani and Sulaiman [4] corroborates the ones above recognizing that despite the fact the open source development has seen remarkable success in recent years, there are a number of product quality issues and challenges facing the OSS development model. Considering exclusively the testing activities, they highlight the lack of knowledge of participants to understand the OSS system architecture and to create additional test cases for it.

Since we are interested to measure the coverage of exception handling code, we evaluated a few papers that discuss the analysis and testing of programs with exception handling structures. For instance, the works of Chatterjee et al. [5] and Choi et al. [6] present models to compute control- and data-flow information for dealing with exceptions, but no tool which implements the proposed models is available.

Sinha and Harrold [7] developed a family of criteria to deal with exception handling construction instantiated for the Java language. The testing criteria definition use a control- and a data-flow model known as interprocedural control-flow graph (ICFG) [8], [9], which is used for the identification of testing requirements. Java exceptions, as presented next, may be synchronized (explicitly raised by a `throw` statement) or unsynchronized (that can be raised at any time implicitly). The main limitation of ICFG is that, unlike our model, it does not represent unsynchronized exceptions. On the other hand, by considering only the synchronous exception it is possible to verify the type of exception to be raised when connecting nodes, so that no false edges are generated. To collect the data for the experiment, the authors used a tool named JABA, which is an acronym for Java Architecture for Bytecode Analysis. JABA provides language-dependent analysis for Java programs and is part of the Aristotle Analysis System [10], but JABA only performed the static analysis and, as soon as we know, there is no tool which implements such criteria.

III. Exception handling: features and requirements in Java

According to Perry *et al.* [11], a pervasive exception handling is required by almost anything that has an algorithmic flow, such as a design process, a workflow or a computer program. Exceptions are used not only as an implication of error, but also as an indication of deviations from the normal conditions established by the system. The main task of an exception handling mechanism in the context of programming languages is to overcome the problems posed by using the usual "return values from a function" as an indication of unexpected conditions. The use of exceptions to indicate error conditions ease the propagation of the erroneous state and also the implementation of the fault tolerance mechanism.

Programming languages like Java, C++ and Ada have similar exception handling mechanisms. In the case of Java, exceptions are represented by objects. We focus on Java for some reasons: 1) it is one of the largest used programming languages in this last decade [12]; 2) there are several open-source Java software with unit testing available; and 3) our previous effort on developing testing tools for Java [13].

Figure 1 shows part of the exception handling class hierarchy of the Java language. All those classes are part of the `java.lang` package. As Figure 1 shows, the `Throwable` class, an immediate subclass of Object, is the root class of the entire exception hierarchy. It has two direct subclasses: `Exception` and `Error`.

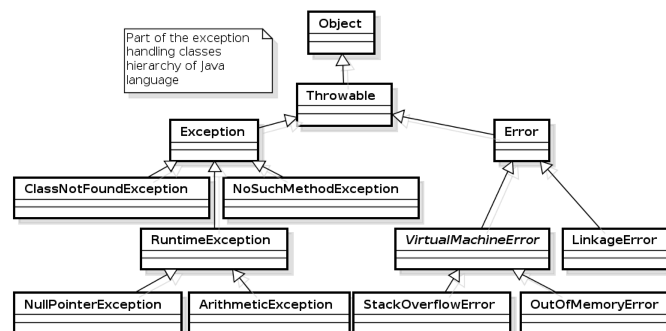

Figure 1. Part of the exception handling class hierarchy of Java [14].

Subclasses of `Exception` represent exceptional conditions that a normal Java program may handle and, except for `RuntimeException` and its subclasses, all the other subclasses of `Exception` are called "checked exceptions",

i.e., exceptions that must be handled since they are verified at compilation time. `RuntimeException` and its subclasses, also known as “unchecked exceptions”, represent runtime conditions that may generally occur in any Java method, but the method is not required to inform that it can raise runtime exceptions. Although they can be handled, unchecked exceptions are not identified at compilation time. On the other hand, all other standard exceptions a method can throw must be informed by means of a `throws` clause. A Java program should try to handle all standard exceptions, since they represent abnormal conditions that should be anticipated and caught to prevent program termination.

In addition to checked and unchecked exceptions, there are errors that can never be raised or handled since they are used to show serious problems with the Java virtual machine, the class loader or any other error which will abort program execution.

All checked exceptions may have exception handling code associated with them. This is done in Java by using a `try-catch-finally` construct. There are three possible valid combinations of these statements: `try-catch`, `try-catch-finally`, and `try-finally`. The `try` statement is composed by a `try` block. The `catch` block is composed by one or more `catch` clauses, responsible for specifying the exception handlers. The `catch` clause formal parameter determines the kind of exception it handles and the variable which will be assigned with the exception instance. The `finally` block, when present, is always executed, even in the presence of control-flow transfer statements like break, continue, and return in the body of the `try` block [14]. A feature of Java’s exception handling mechanism is its non-resumable model, which means that once an exception is raised, the control flow returns to the first statement after the `try` statement responsible for handling such an exception.

In terms of testing, the exception handling mechanism affects the normal control-flow execution. Moreover, the set of instructions that may produce exceptions also has to be considered in the creation of basic CFG blocks. The set of instructions responsible for raising synchronous checked exceptions may be found elsewhere [15].

IV. STRUCTURAL TESTING FOR EXCEPTION HANDLING

In this section, we present our approach to the structural testing of programs with exception handling constructs. It is part of a general framework that permits the application of control- and data-flow criteria to object oriented programs, in particular those developed using the Java language.

As part of this framework, a control- and a data-flow model were developed to accommodate our needs. The model is based on the analysis of bytecode programs instead of source code. This approach offers some advantages, it is language-independent and reflects the actual structure of a program under testing. The next subsections summarize our approach and the way it affects the testing of units with exception handling structures.

A. Control- and data-flow models

A common representation of the program under testing, known as Control-Flow Graph (CFG), is generally used to abstract the internal control flow of the tested unit. A program P can be decomposed in a set of disjoint blocks of statements so that the execution of the first statement inside a given block leads to the execution of all other statements in that block in the order they appear in. All statements in a block, except possibly the first, have a single predecessor. All statements in a block, except possibly the last, have exactly one successor. This means that there is no external control flow from/to statements in the middle of the block. In a CFG, such basic blocks are represented as vertex and the possible execution flow from one block to another is represented as directed edges. A CFG has a single entry node that represents the block which contains the entry instruction of the unit. An exit node has no outgoing edge.

A Def-use graph ($\mathcal{DUG}$) is an extension of the Control-Flow Graph including sets of variables defined and used on each CFG nodes [16]. Therefore, the $\mathcal{DUG}$ contains information about the data flow of the program under testing, characterizing associations between statements in which a definition occurs and statements in which a use is present.

It is out of the scope of this paper to discuss the complete OO testing approach and all the models and algorithms used to analyze the programs. It may be seen in [17].

Two points should be highlighted in the analysis of control-flow characteristics of a Java bytecode program:

- the use of intra-method subroutine calls. JVM has instructions `jsr`, `jsr_w` and `ret` that allow a piece of the method code to be “called” from several points in the same method. This is mostly used to implement the `finally` block of Java.
- exception handlers. Each piece of code inserted in a `catch` block of a Java program is an exception handler (EH). The execution of such a code is not performed by ordinary control-flow, but by the throwing of an exception. In the bytecode code the exception handler is not activated by ordinary instructions either. Each method has a table that describes where the handlers are located in the code and which piece of code they apply to. The flow of execution that is activated by an exception is represented in our $\mathcal{DUG}$ by a different type of edge, called an “exception edge”.

To deal with Java’s exception-handling mechanism, the underlying representation model, i.e., the $\mathcal{DUG}$, should reflect the control-flow during normal program execution and also during the occurrence of exceptions. To represent regular and exception control-flows, we use two kinds of edges: *regular edges* represent the regular control-flow, i.e., defined by the language statements; and *exception edges* represent the control-flow when an exception is raised. With such distinction, testing criteria can be defined to assess test coverage on normal execution flow and on exceptional execution flow.

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9

B. Testing criteria

The basis to define testing criteria for exception handling structures is the concept of *exception-free path*:

> An *exception-free path* is a path $\pi \mid \forall (n_i, n_j) \in \pi \Rightarrow (n_i, n_j)$ that is reachable through a path which does not contain any exception edge.

A path that includes a node n, which may only be reached through a path that contains an exception edge is an *exception-dependent* path.

To address explicitly the coverage of exception handlers code, two non-overlapping testing criteria were defined, so that the tester may concentrate on different aspects of a program at a time. Given the test set $T = \{t_1, t_2, ..., t_r\}$ and the corresponding set of paths $\Pi = \{\pi_1, \pi_2, ..., \pi_r\}$ executed by the elements of T, we define:

- *all-nodes-exception-independent* (All-Nodes$_{ei}$): Π satisfies the all-nodes-exception-independent criterion if every node $n \in N_{ei}$ is included in Π. In other words, this criterion requires that every node of the $\mathcal{DUG}$ graph, reachable through an exception-free path, is executed at least once.
- *all-nodes-exception-dependent* (All-Nodes$_{ed}$): Π satisfies the all-nodes-exception-dependent criterion if every node $n \in N_{ed}$ is included in Π. In other words, this criterion requires that every node of the $\mathcal{DUG}$ graph, not reachable through an exception-free path, is executed at least once.

Considering edges as testing requirements, we have:

- *all-edges-exception-independent* (All-Edges$_{ei}$): Π satisfies the all-edges-exception-independent criterion if every edge $e \in E_{ei}$ is included in Π. In other words, this criterion requires that every edge of the $\mathcal{DUG}$ graph that is reachable through an exception-free path is executed at least once.
- *all-edges-exception-dependent* (All-Edges$_{ed}$): Π satisfies the all-edges-exception-dependent criterion if every edge $e \in E_{ed}$ is included in Π. In other words, this criterion requires that every edge of the $\mathcal{DUG}$ graph not reachable through an exception-free path is executed at least once.

As with the all-nodes and all-edges criteria, we split the all-uses criterion [16], so that two sets of non-overlapping testing requirements are obtained. We named such criteria all-uses-exception-independent and all-uses-exception-dependent, respectively.

- *all-uses-exception-independent* (All-Uses$_{ei}$): Π satisfies the all-uses-exception-independent criterion if for every node $i \in N$ and for every variable $x \in def(i)$, Π includes a def-clear, exception-free path w.r.t. x from node i to every use of x. In other words, this criterion requires that every exception-independent def-c-use association (i, j, x) and every exception-independent def-p-use association $(i, (j, k), x)$ is exercised at least once for any given test case.
- *all-uses-exception-dependent* (All-Uses$_{ed}$): Π satisfies the all-uses-exception-dependent criterion if for every node $i \in N$ and for every variable $x \in def(i)$, Π includes a def-clear, exception-dependent path w.r.t. x from node i to every use of x. In other words, this criterion requires that every exception-dependent def-c-use association (i, j, x) and every exception-dependent def-p-use association $(i, (j, k), x)$ is exercised at least once for any given test case.

The use of testing criteria which consider exception codes when defining testing requirements can improve the testing activity by offering hints to the tester on how the code is organized, in terms of a "normal" or "abnormal" flow. Our test criteria may help in at least three situations:

- it is well known that much of the exception handling code is hard to test and it is left untested intentionally. With the indication of exception-dependent and exception-independent requirements, the tester may consider only the latter, with no need to analyze the feasibility of each, according to his/her goals;
- on the other hand, if the application requires the execution of an exception-dependent code, the use of our criteria can guide the tester indicating which requirements need an abnormal situation to be covered and suggesting a possible incremental testing strategy;
- exception dependent testing requirements can be used as a static code metric. For example, comparing the number of exception independent testing requirements against the number of exception independent requirements may give an indication of the cost of testing both normal and abnormal flow and, in some extent, of the complexity of these parts of the program. Other metrics like lines of code or cyclomatic complexity could also be used in this way if one considers these two types of code.

C. Automation aspects

To support the application of the structural testing criteria presented in the previous sections, we are working on the development of an Open Source testing tool called JaBUTi. We have worked on this tool since 2003 [13], improving its functionalities and extending its application to a variety of software products. Currently, besides testing Java programs at unit level, the tool may also be applied for unit testing of Aspect-Oriented programs [18], Java components [19], Java micro-edition, and mobile programs [20], among others. In addition, the tool can be easily employed to work with any language which generates bytecode as a result of the compilation process.

The steps for executing JaBUTi are depicted in Figure 2. The first step is the creation of a test session, which shows the classes that compose the program under testing and those we want to instrument for the collection of the execution trace. The second step is the generation of testing requirements by using the eight testing criteria. Then, it is necessary to instrument the selected classes. After instrumentation, the program under testing may be executed with one or more test cases and the coverage information is recorded. After test set execution, the covered requirements are identified and the

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9

current status of the test session is updated and visualized on testing reports with different levels of detail. With the reports the tester may decide whether to continue or stop the testing activity based on his/her previously defined stopping criterion.

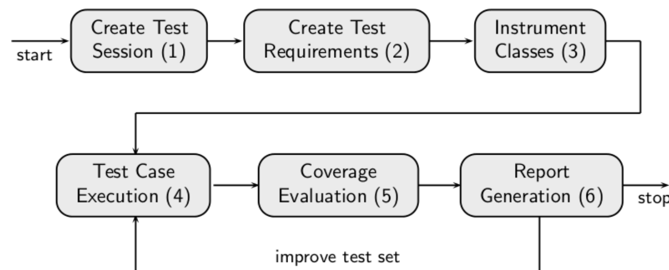

Figure 2. Steps of a test session execution.

We have successfully used JaBUTi on several projects and the tool has been released as an open source software to be used in the context of the QualiPSo project. The interested reader may consult [17], for further information.

V. Experimental application

In this section, we present the results obtained from the application of the exception-dependent criteria to a set of OSS. This initiative is part of our objectives in an attempt to identify the usual behavior of the OSS community while developing test sets for OSS.

Our first task was to make a static evaluation of some open source projects, namely HSQLDB, JUnit, JMeter, PMD, Weka, ServiceMix, Talend Open Studio, SpagoBI, Cimero, Jboss Application Server, Mondrian, Pentaho, and Spago. We have concluded that all of them have test sets associated with them and, as they are integrated with automated tools (Ant or Maven), it can be assumed that they are often run. However, despite this testing culture, the testing techniques applied by the OSS development community could not be identified with accuracy. Considering the current state of testing carried out by OSS communities, it can be observed that:

- in general, the only testing criterion applied is functional. There is no clear evidence of structural (control, data-flow) or fault based testing;
- there is no clear distinction between unit, integration, and system testing. Although there are test suites integrated into the build process (most projects use Ant or Maven to manage software compilation and packaging), there are no clearly defined test plans and strategies after the execution of the test suite. For example, how to proceed when failed test cases are found (e.g., if more than 10% of the test cases failed, the developers must be notified and the software package cannot be released).

A question which regards these test suites is: "Are *ad hoc* test suites sufficient to assign trustworthiness to OSS?" To answer this question we use an approach which comprises structural testing criteria for test set evaluation.

Formal standards like DO-178B [21] and ANSI/IEEE 1008-1987 [22] demand 100% statement and branch coverage for safety critical systems. Regardless of the level of coverage obtained, the importance of coverage testing does not lie on identifying which parts of the product were exercised during test set execution, but on identifying the ones which have not yet been executed.

Cornett [23] discusses the minimum acceptable code coverage and argues that a coverage level between 70-80% is a reasonable goal for system testing for the majority of software products. Moreover, Cornett [23] also defends that unit, integration, and system testing levels demand a decreasing coverage level since, in general, it is easier to achieve a higher coverage of a single unit than that of an entire system. An important point that has not been mentioned is how exception handling structures affect coverage level. It is not clear whether the 70-80% mentioned by Cornett considers normal and exception handling codes or only normal code. By using the testing criteria presented in Section IV, a more precise assessment of code coverage may be obtained.

As an initial investigation, we analyzed four traditional OSS: HSQLDB (version 1.9 Alpha 2), JMeter (version 2.3.2), JUnit (version 4.6), and PMD (version 5.0). The evaluation is performed via a testing tool that implements all the mentioned criteria, but we concentrate the analysis on the exception dependent ones. In this way, the restriction imposed by the selection of a OSS is the need that its unit test set run successfully, enabling the coverage information to be collected.

The OSS are implemented in Java and correspond to the last release available at the time the data was collected. We concentrate our effort on evaluating the impact of exception handling in these projects and how test sets were developed in order to cover exception code.

Our first evaluation consisted in identifying the size of the projects and the number of methods which employ exception handling constructions. The smallest OSS analyzed (JUnit) has 2,614 lines of code (LOC), and the biggest (HSQLDB) has 63,592 LOC. On average, at method level, the use of exception handlers construction is present on 8% of the total number of methods, percentage close to the average obtained by Sinha and Harrold [9] for a different set of programs. After performing the static analysis, we started the dynamic analysis.

We created an instrumented version of the programs under testing and executed the available test set against those versions, so that dynamic trace information could be collected and confronted with the structural testing criteria. Tables I, II, and III show the data obtained.

Tables I and II show the coverage after the execution of all available test sets developed by the OSS community for each program, considering the exception-independent and the exception-dependent testing criteria, respectively. For instance, the JMeter test set was the one which determined the highest coverage with respect to all testing criteria. For All-Nodes$_{ei}$, the test set covered 7,845 out of 20,462 required elements, 38.34% of coverage. As for the other testing criteria with higher complexity, the coverage percentage of the required elements were 28.27%, 26.55%, and 25.75%. In general, a level of coverage below 40% for these programs is very low

TABLE I. REQUIREMENT COVERAGE: EXCEPTION-INDEPENDENT CRITERIA

OSS	Criterion			
	All-Nodes$_{ei}$	**All-Edges$_{ei}$**	**All-Uses$_{ei}$**	**All-Pot-Uses$_{ei}$**
	Coverage (%)	**Coverage (%)**	**Coverage (%)**	**Coverage (%)**
HSQLDB	8,029 / 40,703 (19.73%)	7,476 / 45,098 (16.58%)	19,720 / 126,246 (15.62%)	67,847 / 458,843 (14.79%)
JMeter	7,845 / 20,462 (38.34%)	5,461 / 19,317 (28.27%)	10,935 / 41,180 (26.55%)	33,615 / 130,547(25.75%)
JUnit	608 / 1,951 (31.16%)	380 / 1,436 (26.46%)	631 / 2,624 (24.05%)	1,475 / 6,243 (23.63%)
PMD	7,938 / 21,184 (37.47%)	6,858 / 23,249 (29.50%)	13,331 / 57,552 (23.16%)	38,404 / 252,261 (15.22%)

TABLE II. REQUIREMENT COVERAGE: EXCEPTION-DEPENDENT CRITERIA

OSS	Criterion			
	All-Nodes$_{ed}$	**All-Edges$_{ed}$**	**All-Uses$_{ed}$**	**All-Pot-Uses$_{ed}$**
	Coverage (%)	**Coverage (%)**	**Coverage (%)**	**Coverage (%)**
HSQLDB	141 / 1,942 (7.26%)	49 / 6,513 (0.75%)	256 / 2,750 (9.31%)	3,591 / 38,032 (9,44%)
JMeter	51 / 1,541 (3.31%)	39 / 4,863 (0.80%)	52 / 2,093 (2.48%)	276 / 15,301 (1.80%)
JUnit	12 / 156 (7.69%)	9 / 184 (4.89%)	13 / 183 (7.10%)	29 / 632 (4.59%)
PMD	325 / 2,039 (15.94%)	121 / 3,814 (3.17%)	388 / 3590 (10.81%)	1689 / 20285 (8.33%)

TABLE III. EXCEPTION HANDLERS DATA AT METHOD LEVEL: ALL-NODES$_{ed}$ CRITERION

OSS	Number of methods	Number of requirements	Average	Number of methods with no coverage	Total coverage
HSQLDB	683	1,942	2.84%	669 (97.95%)	7.26%
JMeter	625	1,541	2.47%	595 (95.20)%	3.31%
JUnit	63	156	2.48%	57 (90.48%)	7.69%
PMD	374	2,039	5.45%	299 (79.95%)	15.94%

and demonstrates that much of the code is only executed by the users and that their test cases are probably not integrated in the official test set. In the case of HSQLDB, the percentage of coverage of the All-Nodes$_{ei}$ criterion is 19.73%, which means that more than 80% of source code is not executed by any official test case in the test set.

Table II shows the coverage obtained with respect to the exception-dependent criteria, i.e., those criteria which demand an exception to be raised for covering the testing requirements. Considering the most basic structural testing criterion (All-Nodes$_{ed}$), the highest coverage was determined by the test set of the PMD project, which executed 325 out of 2,039 testing requirements (15.94%). This is a clearly very low coverage and additional test sets should be developed at least to confirm that most of the exception handling construction in the program could be executed at least once.

When comparing such a coverage against the exception-independent criteria (Table I), one can see that even for the All-Nodes$_{ed}$ criterion, the level of coverage for all programs ranges from 19.73% for HSQLDB and 38.34% for JMeter. This implies that the provided test set for such programs has a very low coverage in terms of structural testing criteria, even for the criteria not related with exception handling.

In Table III, we present more detailed information about the total number of methods with exception handlers, the total number of testing requirements generated by the All-Nodes$_{ed}$ criterion, the average number of requirements per method, the number of methods which do not have exception handler construction executed by any test case, and the total coverage obtained for such a criterion. As Table III shows, there is a high percentage of methods with zero coverage against any exception-dependent criterion. For three programs, more than 90% of their methods have no test case to execute their exception handling constructions. The best program is PMD, for which the current test set is able to exercise 75 (20.05%) out of 374 methods with exception handlers, but still 79.95% of the methods are not executed by any test case.

Another point that might be inferred from Table III is that the exception handlers have normally few nodes, i.e., they are less complex in terms of logical structure. In fact, by analyzing such products, it is possible to observe that the majority of exception handlers have empty `catch` blocks, just avoiding the exception propagation but with no corrective action associated with it. The most complex exception handlers are found in PMD, which has on average 5.45 requirements per method, followed by HSQLDB with 2.84 requirements per method, considering the All-Nodes$_{ed}$ criterion.

These numbers show that all the analyzed projects reveal a low level of code coverage for code unrelated to exception handling structures. This is disturbing because it reveals the lack of concern from OSS communities on constructing a reference test set for their products. The tests are in fact performed *ad hoc* by the user and test cases are not incorporated in the official test set.

For exception handling criteria the situation is even worse. Although the complexity of exception handlers is not high – as shown by the number of testing requirements – the coverage of such testing requirements is very low. Many of the methods with this kind of code are not even executed once. In addition,

Copyright (c) IARIA, 2013. ISBN: 978-1-61208-304-9

there is no indication of test cases specifically designed to address exception handling.

In this scenario, the testing criteria presented in this paper may be of great help for developers, as they guide the tester through the process of selecting test cases that are or are not related to exception handling. Even if the adopted policy is not to execute exception handlers because they may be difficult to reach, our approach reveals which requirements could be neglected and which should be covered.

VI. Conclusions and future work

To support the control- and data-flow model and the defined testing criteria, we implemented a tool and presented experimental data collected from a set of four OSS. The experiment intended to assess the adequacy of pre-existent test sets against the set of exception-dependent structural testing criteria.

Our observations reveal that, for all the evaluated projects, the coverage of exception handling constructions was considerable low. For instance, the maximum coverage of the All-Nodes$_{ed}$ criterion was below 16%, which shows that, in general, there is no concern for the development of test cases to exercise exceptional conditions in the projects. Moreover, many exception constructions have empty `catch` blocks, which reveals that the exception handler, though present, is used only to avoid the spread of the exception, not to recover from an erroneous condition.

Even when evaluating the quality of the pre-existent test sets against the exception-independent criteria, the maximum coverage for the All-Nodes$_{ei}$ criterion was below 39%, which is generally regarded as a low level of coverage and an indicator that the test set should be improved. New versions of the analyzed software products may include additional test cases to improve the coverage with respect to the proposed testing criteria. This is an issue to be investigated; however, what this initial investigation indicates is that the open-source community should pursue more thorough test suites, especially addressing exception related code.

In future, we will continue to evaluate other OSS projects. Our aim is to finalize the evaluations of the previously developed test sets, to improve some of them based on the coverage criteria, to identify the contribution of the new added test cases in terms of their fault detection capability – considering the recorded faults in the bug tracker systems of these projects – and, finally, to define an incremental approach for testing OSS so that a minimal trustworthiness might be determined.

Acknowledgment

The authors would like to thank the Instituto de Informática – INF/UFG, Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – CAPES – Brasil, and Fundação de Amparo à Pesquisa do Estado de Goiás – FAPEG – Brasil, which support this work.

References

[1] QualiPSo, "Qualipso project (quality platform for open source software)," Project Homepage – Europe Comission – Grant Number IST-FP6-IP-034763, 2007, [retrieved: Oct., 2013]. [Online]. Available: http://www.qualipso.org/

[2] M. Aberdour, "Achieving quality in open source software," *IEEE Software*, vol. 24, no. 1, pp. 58–64, 2007.

[3] L. Zhao and S. Elbaum, "A survey on quality related activities in open source," *SIGSOFT Softw. Eng. Notes*, vol. 25, no. 3, pp. 54–57, May 2000, [retrieved: Oct., 2013]. [Online]. Available: http://doi.acm.org/10.1145/505863.505878

[4] A. Khanjani and R. Sulaiman, "The process of quality assurance under open source software development," in *Computers Informatics (ISCI), 2011 IEEE Symposium on*, 2011, pp. 548–552.

[5] R. K. Chatterjee, B. G. Ryder, and W. A. Landi, "Complexity of concrete type-inference in the presence of exceptions," in *Lecture Notes in Computer Science*, vol. 1381. Springer, Apr. 1998, pp. 57–74.

[6] J.-D. Choi, D. Grove, M. Hind, and V. Sarkar, "Efficient and precise modeling of exceptions for the analysis of java programs," *SIGSOFT Software Engeneering Notes*, vol. 24, no. 5, pp. 21–31, 1999.

[7] S. Sinha and M. J. Harrold, "Criteria for testing exception-handling constructs in Java programs," in *International Conference on Software Maintenance*. Oxford, England: IEEE Computer Society Press, Aug. 1999, pp. 265–274.

[8] ——, "Analysis of programs with exception-handling constructs," in *ICSM'98 – International Conference on Software Maintenance*, Bethesda, MD, Nov. 1998, pp. 348–357.

[9] ——, "Analysis and testing of programs with exception-handling constructs," *IEEE Transactions on Software Engineering*, vol. 26, no. 9, pp. 849–871, Sep. 2000.

[10] M. J. Harrold, L. Larsen, J. Lloyd, D. Nedved, M. Page, G. Rothermel, M. Singh, and M. Smith, "Aristotle: a system for development of program analysis based tools," in *ACM-SE 33: Proceedings of the 33rd annual on Southeast regional conference*. New York, NY, USA: ACM, 1995, pp. 110–119.

[11] D. E. Perry, A. Romanovsky, and A. Tripathi, "Current trends in exception handling," *ieeese*, vol. 26, no. 10, pp. 921–922, Oct. 2000.

[12] TIOBE Software BV, "TIOBE Index," Web site, Sep. 2013, [retrieved: Oct., 2013]. [Online]. Available: http://www.tiobe.com/

[13] A. M. R. Vincenzi, W. E. Wong, M. E. Delamaro, and J. C. Maldonado, "JaBUTi: A coverage analysis tool for Java programs," in *XVII SBES – Brazilian Symposium on Software Engineering*. Manaus, AM, Brazil: Brazilian Computer Society (SBC), Oct. 2003, pp. 79–84.

[14] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification*, 3rd ed. Addison Wesley, Jun. 2005.

[15] T. Lindholm and F. Yellin, *The Java Virtual Machine Specification*, 2nd ed. Addison-Wesley, 1999.

[16] S. Rapps and E. J. Weyuker, "Selecting software test data using data flow information," *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 367–375, Apr. 1985.

[17] A. M. R. Vincenzi, M. E. Delamaro, W. E. Wong, and J. C. Maldonado, "Establishing structural testing criteria for Java bytecode," *Software Practice and Experience*, vol. 36, no. 14, pp. 1513–1541, Nov. 2006.

[18] 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," *The Journal of Systems and Software*, vol. 80, no. 6, pp. 862–882, Jun. 2007.

[19] A. M. R. Vincenzi, J. C. Maldonado, W. E. Wong, and M. E. Delamaro, "Coverage testing of Java programs and components," *Journal of Science of Computer Programming*, vol. 56, no. 1-2, pp. 211–230, Apr. 2005.

[20] M. E. Delamaro and A. M. R. Vincenzi, "Structural testing of mobile agents," in *III International Workshop on Scientific Engineering of Java Distributed Applications (FIDJI'2003)*, ser. Lecture Notes on Computer Science, E. A. Nicolas Guelfi and G. Reggio, Eds. Springer, Nov. 2003, pp. 73–85.

[21] RTCA/EUROCAE, "Software considerations in airborne systems and equipment certification," Radio Technical Commission for Aeronautics – RTCA & European Organization for Civil Aviation Equipment – EUROCAE, Washington, D.C., EUA, Relatóro Técnico DO-178B/ED12B, Dec. 1992.

[22] IEEE, "IEEE standard for software unit testing," IEEE Computer Society Press, Standard ANSI/IEEE Std 1008-1987, 1987.

[23] S. Cornett, "Minimum acceptable code coverage," On-line article, 2007, [retrieved: Oct., 2013]. [Online]. Available: http://www.bullseye.com/minimum.html

Copyright (c) IARIA, 2013.