Characterizing, Verifying and Improving Software Resilience with Exception Contracts and Test Suites

Benoit Cornu, Lionel Seinturier, Martin Monperrus

Characterizing, Verifying and Improving Software Resilience with Exception Contracts and Test Suites

Benoit Cornu, Lionel Seinturier, and Martin Monperrus
INRIA - University of Lille

1. INTRODUCTION

At the Fukushima nuclear power plant, the anticipated maximum height of waves in a tsunami was 5.6m[1]. However, on March 11, 2011, the highest waves struck at 15m. Similarly, in software development, there are the errors anticipated at specification and design time, those encountered at development and testing time, and those that happen in production mode yet never anticipated, as Fukushima's tsunami.

In this presentation, we aim at reasoning on the ability of software to correctly handle unanticipated exceptions. We call this ability "software resilience". It is complementary to the concepts of robustness and fault tolerance [6]. Software robustness emphasizes that the system under study resists to incorrect input data (whether malicious or buggy). Fault tolerance can have a wide acceptation [2], but is mostly associated with hardware faults. "Software resilience" conveys the notion of risks from unanticipated errors (whether environmental or internal) at the software level.

2. APPROACH

We focus on the resilience against exceptions. Exceptions are programming language constructs for handling errors [5]. Exceptions are widely used in practice [4]. In our work, the resilience against exceptions is the ability to correctly handle exceptions that were never foreseen at specification time neither encountered during development. Our motivation is to help developers to understand and improve the resilience of their applications.

This sets a three-point research agenda: (RQ#1) What does it mean to specify anticipated exceptions? (RQ#2) How can one characterize and measure resilience against unanticipated exceptions? (RQ#3) How can one put this knowledge in action to improve the resilience?

> Our approach helps the developers to be aware of what part of their code is resilient, and to automatically recommend modifications of catch blocks to improve the resilience of applications.

2.1 What does it mean to specify anticipated exceptions?

A test suite is a collection of test cases, each of which contains a set of assertions [3]. The assertions specify what the software is meant to do. Hence, we consider that a test suite is a specification since they are available in many existing programs and are pragmatic approximations of idealized specifications[8].

For instance, "assert(3, division(15,5))" specifies that the result of the division of 15 by 5 should be 3. But when software is in the wild, it may be used with incorrect input or encounter internal errors [9]. For instance, what if one calls "division(15,0)"? Consequently, a test suite may also encode what a software package does besides standard usage. For instance, one may specify that "division(15,0)" should throw an exception "Division by zero not possible". We will present a characterization and empirical study of how exception-handling is specified in test suites.

The classical way of analyzing the execution of test suites is to separate passing "green test cases" and failing "red test cases" [1]. This distinction does not consider the specification of exception handling. Beyond green and red test cases, we characterize the test cases in three categories: the pink, blue and white test cases. Those three new types of test cases are a partition of green test cases.

Pink Test Cases: Specification of Nominal Usage.

The "pink test cases" are those test cases where no exceptions at all are thrown or caught. The pink test cases specify the nominal usage of the software under test (SUT), i.e. the functioning of the SUT according to plan under standard input and environment. Note that a pink test case can still execute a try-block (but never a catch block by definition).

Blue Test Cases: Specification of State Incorrectness Detection.

The "blue test cases" are those test cases which assert the presence of exception under incorrect input (such as for instance "division(15,0)"). The number of blue test cases B estimates the amount of specification of the state correctness detection (by amount of specification, we mean the number of specified failure scenarii). B is obtained by intercepting all bubbling exceptions, i.e. exceptions that quit the application code and arrive in the test case code.

White Test Cases: Specification of Resilience.

The "**white test cases**" are those test cases that do not expect an exception (they are standard green functional test cases) but use throw and catch at least once in application code. Contrary to blue tests, they are not expecting thrown exceptions but they use them only internally.

In our terminology, white test cases specify the "resilience" of the system under test. In our context, resilience means being able to recover from exceptions. This is achieved by 1) simulating the occurrence of an exception, 2) asserting that the exception is caught in application code and the system is in a correct state afterwards. If a test case remains green after the execution of a catch block in the application under test, it means that the recovery code in the catch block has

[1] those colors refers to the graphical display of Junit, where passing tests are *green* and failing tests are *red*

successfully repaired the state of the program.

The number and proportion of white test cases gives a further indication of the specification of the resilience.

2.2 How to characterize and measure resilience against unanticipated exceptions?

We now present two novel contracts for exception-handling programming constructs. We use the term "contract" in its generic acceptation: a property of a piece of code that contributes to reuse, maintainability, correctness or another quality attribute. For instance, the "hashCode/equals" contract[2] is a property on a pair of methods. This definition is broader in scope than that in Meyer's "contracts" [7] which refer to preconditions, postconditions and invariants contracts.

We focus on contracts on the programming language construct try/catch, which we refer to as "try-catch". A try-catch is composed of one try block and one catch block.

Source Independence Contract.

When a harmful exception occurs during testing or production, a developer has two possibilities. One way is to avoid the exception to be thrown by fixing its root cause (e.g. by inserting a not null check to avoid a null pointer exception). The other way is to write a try block surrounding the code that throws the exception. The catch block ending the try block defines the recovery mechanism to be applied when this exception occurs. The catch block recovers the particular encountered exception. By construction, the same recovery would be applied if another exception of the same type occurs within the scope of the try block.

This motivates the source-independence contract: the normal recovery behavior of the catch block must work for the foreseen exceptions; but beyond that, it should also work for exceptions that have not been encountered but may arise in a near future.

We define a novel exception contract, that we called "source-independence" as follows:

Definition A try-catch is source-independent if the catch block proceeds equivalently, whatever the source of the caught exception is in the try block.

Pure Resilience Contract.

In general, when an error occurs, it is more desirable to recover from this error than to stop or crash. A good recovery consists in returning the expected result despite the error and in continuing the program execution.

One way to obtain the expected result under error is to be able to do the same task in a way that, for the same input, does not lead to an error but to the expected result. Such an alternative is sometimes called "plan B". In terms of exception, recovering from an exception with a plan B means that the corresponding catch contains the code of this plan B. The "plan B" performed by the catch is an alternative to the "plan A" which is implemented in the try block. Hence, the contract of the try-catch block (and not only the catch or only the try) is to correctly perform a task T under consideration whether or not an exception occurs. We refer to this contract as the "pure resilience" contract.

A "pure resilience" contract applies to try-catch blocks. We define it as follows:

Definition A try-catch is purely resilient if the system state is equivalent at the end of the try-catch execution whether or not an exception occurs in the try block.

3. EMPIRICAL EVALUATION

We perform an empirical evaluation on 9 well-tested open-source software applications. We made an evaluation on the 9 corresponding test suites, with 78% line coverage in average. The line coverage of the test suites under study is a median of 81%, a minimum of 50% and a maximum of 94%.

All the experiments are based on source-code transformation, using Spoon[3].

For the test colors, we analyzes 9679 test cases. It shows that between 5 and 19% of test cases expect exceptions (blue tests) and that between 4 and 26% of test cases uses exceptions without bubbling ones (white tests).

For the catch-contrats,we analyzes the 241 executed catch blocks, shows that 101 of them expose resilience properties (source-independence or pure-resilience).

4. CONCLUSION

To sum up, our contributions are:

- A characterization of specification of software resilience in test suites,
- A definition and formalization of two contracts on try-catch blocks,
- A source code transformation to improve resilience against exceptions.

5. REFERENCES

[1] CNSC Fukushima Task Force Report. Technical Report INFO-0824, Canadian Nuclear Safety Commission, 2011.

[2] A. Avizienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[3] B. Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[4] B. Cabral and P. Marques. Exception handling: A field study in java and. net. In *ECOOP 2007–Object-Oriented Programming*, pages 151–175. Springer, 2007.

[5] J. Gosling, B. Joy, G. Steele, and G. Bracha. *Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

[6] J.-C. Laprie. From dependability to resilience. In *Proceedings of DSN 2008*, 2018.

[7] B. Meyer. Applying design by contract. *Computer*, 25(10):40–51, 1992.

[8] M. Staats, M. W. Whalen, and M. P. E. Heimdahl. Programs, tests, and oracles: the foundations of testing revisited. In *2011 International Conference on Software Engineering (ICSE)*, pages 391–400. IEEE, 2011.

[9] P. Zhang and S. Elbaum. Amplifying tests to validate exception handling code. In *Proceedings of the 2012 International Conference on Software Engineering*, pages 595–605. IEEE Press, 2012.

[2] http://docs.oracle.com/javase/7/docs/api/java/lang/Object.html#hashCode()

[3] http://spoon.gforge.inria.fr/