Information and Software Technology 57 (2015) 66–76

Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Exception handling analysis and transformation using fault injection: Study of resilience against unanticipated exceptions

Benoit Cornu *, Lionel Seinturier [1], Martin Monperrus [2]

University of Lille & Inria, Laboratoire d'Informatique Fondamentale de Lille (LIFL), Campus Scientifique, 59655 Villeneuve d'Ascq Cedex, France

ARTICLE INFO

Article history:
Received 14 March 2014
Received in revised form 14 August 2014
Accepted 15 August 2014
Available online 16 September 2014

Keywords:
Dynamic verification
Contract
Exception handling
Fault injection

ABSTRACT

Context: In software, there are the error cases that are anticipated at specification and design time, those encountered at development and testing time, and those that were never anticipated before happening in production. Is it possible to learn from the anticipated errors during design to analyze and improve the resilience against the unanticipated ones in production?
Objective: In this paper, we aim at analyzing and improving how software handles unanticipated exceptions. The first objective is to set up contracts about exception handling and a way to assess them automatically. The second one is to improve the resilience capabilities of software by transforming the source code.
Method: We devise an algorithm, called short-circuit testing, which injects exceptions during test suite execution so as to simulate unanticipated errors. It is a kind of fault-injection techniques dedicated to exception-handling. This algorithm collects data that is used for verifying two formal contracts that capture two resilience properties w.r.t. exceptions: the source-independence and pure-resilience contracts. Then we propose a code modification technique, called "catch-stretching" which allows error-recovery code (of the form of catch blocks) to be more resilient.
Results: Our evaluation is performed on 9 open-source software applications and consists in analyzing 241 catch blocks executed during test suite execution. Our results show that 101/214 of them (47%) expose resilience properties as defined by our exception contracts and that 84/214 of them (39%) can be transformed to be more resilient.
Conclusion: Our work shows that it is possible to reason on software resilience by injecting exceptions during test suite execution. The collected information allows us to apply one source code transformation that improves the resilience against unanticipated exceptions. This works best if the test suite exercises the exceptional programming language constructs in many different scenarios.

1. Introduction

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

Resilience is *"the persistence of service delivery that can justifiably be trusted, when facing changes"* [14]. *"Changes may refer to unexpected failures, attacks or accidents (e.g., disasters)"* [25]. In this paper, we aim at reasoning on the ability of software to correctly handle unanticipated errors.

We focus on the resilience against exceptions [11]. Exceptions are programming language constructs for handling errors. Exceptions are implemented in most mainstream programming languages [12] and widely used in practice [6]. In large and complex software, it is impossible to predict all error cases that will happen in the field (real-world environments are too unpredictable and usages too diverse). In this paper, the resilience against exceptions is the ability to correctly handle exceptions that were never foreseen at specification time neither encountered during development or testing. This is our deep motivation: helping developers to understand and improve the resilience of their applications against unanticipated exceptions.

The key difficulty behind this research agenda is the notion of "unanticipated": how to reason on what one does not know or on what one has never seen? To answer this question, we start

* Corresponding author. Tel.: +33 3 59 35 87 62.
E-mail addresses: benoit.cornu@inria.fr (B. Cornu), lionel.seinturier@univ-lille1.fr (L. Seinturier), martin.monperrus@univ-lille1.fr (M. Monperrus).
[1] Tel.: +33 3 59 35 87 76.
[2] Tel.: +33 3 59 35 87 61.

http://dx.doi.org/10.1016/j.infsof.2014.08.004
0950-5849/

by proposing one definition of "anticipated exception". First, we consider well-tested software (i.e. those with a good automated test suite). Second, we define an "anticipated exception" as an exception that is triggered during the test suite execution. To this extent, those exceptions and the associated behavior (error detection, error-recovery) are specified by the test suite (which is a pragmatic approximation of an idealized specification [24]).

Then, we simulate "unanticipated exceptions" by injecting exceptions at appropriate places during test suite execution. This fault injection technique, called "short-circuit testing", consists of throwing exceptions at the beginning of try-blocks, simulating the worst error case when the complete try-block is skipped due to the occurrence of a severe error. Despite the injected exceptions, a large number of test cases still passes. When this happens, it means that the software under study is able to resist to certain unanticipated exceptions. It can be said "resilient" according to our definition of "Resilience".

The art of injecting exceptions during test suite execution consist of (1) selecting the right places to inject exceptions (2) choosing the right point in time for injection and (3) throwing the appropriate kind of exceptions. With an intuitive, then formal reasoning on the nature of resilience and exceptions, we tackle those three challenges and define two contracts on the programming language construct "try-catch" that capture two facets of software resilience against unanticipated exceptions. The satisfaction or violation of those contracts is assessed using the execution data collected during short-circuit testing.

Finally, we use the knowledge on resilience obtained with short-circuit testing to replace the caught type of a catch block by one of its super-type. This source code transformation, called "catch stretching" is considered correct if the test suite continues to pass. By enabling catch blocks to correctly handle more types of exception (w.r.t. the specification), the code is more capable of handling unanticipated exceptions.

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

Our technique is novel. There are techniques to provide information about the test suite with respect to exceptions or to improve the test suite [5,8,10,26]. Our contribution is on analyzing and improving the applicative code itself (the test suite is just a means). Other papers make static analyses of exception handling [22,23]. Our contribution is a dynamic technique which uses a new kind of fault injection.

We evaluate our approach by analyzing the resilience of 9 well-tested open-source applications written in Java. In this dataset, we analyze the resilience capabilities of 241 try-catch blocks and show that 92 of them satisfy at least one resilience contract and 24 try-catch blocks violate a resilience property.

To sum up, our contributions are:

- A definition and formalization of two contracts on try-catch blocks.
- An algorithm and four predicates to verify whether a try-catch satisfies those contracts.
- A source code transformation to improve the resilience against exceptions.
- An empirical evaluation on 9 open sources applications with one test suite each showing that there exists resilient try-catch blocks in practice.

2. Background

In our work, we use the distinction of Avizienis et al. [2] between faults, errors and failures. However, we also consider the common usage, which consists of fault-injection and error-handling whereas it might sometimes be more appropriate to say error-injection or fault-handling. In our paper, for sake of understandability, we prefer the common usage and use well-known expressions such as fault-injection or fault model.

2.1. Background on exceptions

Exceptions are programming language constructs for handling errors [11]. Exceptions can be thrown and caught. When one throws an exception, this means that something has gone wrong and this cancels the nominal behavior of the application: the program will not follow the normal control-flow and will not execute the next instructions. Instead, the program will "jump" to the nearest matching catch block. In the worst case, there is no matching catch block in the stack and the exception reaches the main entry point of the program and consequently, stops its execution (i.e. crashes the program). When an exception is thrown then caught, it is equivalent to a direct jump from the throw location to the catch location: in the execution state, only the call stack has changed, but not the heap.[3] For a practical presentation of exceptions in mainstream programming languages, we refer to any introductory textbook, e.g. [20]. Avizienis et al. [2] do not mention exceptions. We consider exceptions as errors and we use the term error in the paper as much as possible.

2.2. Definition of resilience

We embrace the definition of "software resilience" by Laprie as interpreted by Trivedi et al.:

Definition. Resilience is *"the persistence of service delivery that can justifiably be trusted, when facing changes"* [14]. *"Changes may refer to unexpected failures, attacks or accidents (e.g., disasters)"* [25].

Along with Trivedi et al., we interpret the idea of "unexpected" events with the notion of "design envelope" [25], a known term in safety critical system design. The design envelope defines all the anticipated states of a software system. It defines the boundary between anticipated and unanticipated runtime states. The design envelope contains both correct states and incorrect states, the latter resulting from the anticipation of misusages and attacks. According to that, "resilience deals with conditions that are outside the design envelope" [25]. Along this line, we consider that the main difference between software resilience and software robustness is that software robustness deals with anticipated kinds of errors (i.e. inside the "design envelope").

In this paper, we focus on the resilience in the context of software that uses exceptions. We interpret and refine this general definition in the context of mainstream exception handling.

Definition. Resilience against exceptions is the software system's ability to reenter a correct state when an unanticipated exception occurs.

2.3. Specifications and test suites

A test suite is a collection of test cases where each test case contains a set of assertions [4]. The assertions specify what the

[3] The heap may change if the programming language contains a finalization mechanism (e.g. in Module-2+ [19]).