Exception Handling Analysis and Transformation Using Fault Injection: Study of Resilience Against Unanticipated Exceptions

Benoit Cornu, Lionel Seinturier, Martin Monperrus

Exception Handling Analysis and Transformation Using Fault Injection: Study of Resilience Against Unanticipated Exceptions

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

December 18, 2014

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 5.6m [1]. On March 11, 2011, the highest waves struck at 15m. 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 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, Laprie and Randell [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[1]. 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].

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

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 software is meant to do (i.e. it defines the design envelope). Hence, in the rest of this paper, we consider the test suites as specifications[2]. For instance, "assert(3, division(15,5))" specifies that the result of the division of 15 by 5 should be 3.

A test suite may also encode what a software package does outside standard usage (error defined in the design envelope). For instance, one may specify that "division(15,0)" should throw an exception "Division by zero not possible". Hence, the exceptions that are thrown during test suite execution are the anticipated errors. If an exception is triggered by the test and caught later on in the application code, the assertions specify that the exception-handling code has worked as expected.

Our definition of resilience relates to exceptions that are not specified, that are outside the design envelope. We consider test suites as approximation of the design envelope. Consequently, in this paper, the considered resilience consists in handling unanticipated exceptions where we define unanticipated exceptions as exceptions that are not triggered during test suite execution.

[2]Conversely, when we use the term "specification", we refer to the test suite (even if they are an approximation of an idealized specification [24])

3 Automatic Analysis and Increase of Software Resilience

We define in Section 3.1 two exception contracts applicable to try-catch blocks. We then describe an algorithm (see Section 3.2) and formal predicates (see Section 3.3) to verify those contracts according to a test suite. Finally we present the concept of catch stretching, a technique to improve the resilience of software applications against exceptions (see Section 3.4). The insight behind our approach is that we can use test suites as an oracle for the resilience capabilities against unanticipated errors.

3.1 Definition of Two Contracts for Exception Handling

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[3] is a property on a pair of methods. Our definition is broader in scope than Meyer's "contracts" [17] which refer to preconditions, postconditions and invariants contracts.

We focus on contracts on the programming language construct try and catch blocks, which we refer to as "try-catch". A try-catch is composed of one try block and one catch block. Note that a try with multiple catch blocks is considered as a set of pairs consisting of the try block and one of its catch blocks. This means that a try with n catch blocks is considered as n try-catch blocks. This concept is generalized in most mainstream languages, sometimes using different names (for instance, a catch block is called an "except" block in Python). In this paper, we ignore the concept of "finally" block [12] which is more language specific and much less used in practice [6].

3.1.1 Source Independence Contract

Motivation When an 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 responsibility is to recover from 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 at a different location.

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

Listing 1: An Example of Source-Independent Try-Catch Block.

```
try{
 String arg = getArgument();
 String key = format(arg);
 return getProperty(key , isCacheActivated
       );
}catch(MissingPropertyException e){
  return "missing property";
}
```

Listing 2: An Example of Source-Dependent Try-Catch Block.

```
boolean isCacheActivated = false;
try{
  isCacheActivated = getCacheAvailability
        ();
  return getProperty(key ,
        isCacheActivated);
}catch(MissingPropertyException e){
  if( isCacheActivated ){
    return "missing property";
  }else{
    throw new CacheDisableException();
} }
```

Listing 3: An Example of Purely-Resilient Try-Catch Block.

```
try{
  return getPropertyFromCache(key);
}catch(MissingPropertyException e){
  return getPropertyFromFile(key);
}
```

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.

For now, we loosely define "proceeds equivalently": if the system is still in error, it means that the error kind is the same; if the system has recovered, it means that the available functionalities are the same.

For example, Listing 1 shows a try-catch that *satisfies* the source-independence contract. If a value is missing in the application, an exception is thrown and the method returns a default value "missing property". The code of the catch block (only one return statement) clearly does not depend on the application state. The exception can be thrown by any of the 3 statements in the try, and the result will still be the same.

On the contrary, Listing 2 shows a try-catch that *violates* the source-independence contract. Indeed, the result of the catch process depends on the value of *isCacheActivated*. If the first statement fails (throws an exception), the variable *isCacheActivated* is false, then an exception is thrown. If the first statement passes but the second one fails, *isCacheActivated* can be *true*, then the value *missing property* is returned. The result of the execution of the catch depends on the state of the program when the catch begins (here it depends on the value of the *isCacheActivated* variable). In case of failure, a developer cannot know if she will have to work with a default return value or with an exception. This catch is indeed source-dependent.

We will present a formal definition of this contract and an algorithm to verify it in Section 3.3. We will show that both source-independent and source-dependent catch blocks exist in practice in Section 4.

Discussion How can it happen that developers write source-dependent catch blocks? Developers discover some exception risks at the first run-time occurrence of an exception at a particular location. In this case, the developer adds a try-catch block and puts the exception raising code in the try body. Often, the try body contains more code than the problematic statement in order to avoid variable scope and initialization problems. However, while implementing the catch block, the developer still assumes that the exception can only be thrown by the problematic statement, and refers to variables that were set in previous statements in the try block. In other words, the catch block is dependent on the application state at the problematic statement. If the exception comes from the problematic statement, the catch block works, if not, it fails to provide the expected recovery behavior.

The source independence contract shows that if an unanticipated exception happens the catch block would still be able to recover the application state. This means that the try-catch is able to handle unanticipated exceptions, hence it is resilient w.r.t. the chosen definition of resilience (see Section 2.2).

3.1.2 Pure Resilience Contract

Motivation 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.

By system state equivalence, we mean that the effects of the plan A on the system are similar to those of plan B from a given observation perspective. If the observation perspective is a returned value, the value from plan A is semantically equivalent to the value of plan B (e.g. satisfies an "equals" predicate method in Java).

For example, Listing 3 shows a purely resilient try-catch where a value is required, the program tries to access this value in the cache. If the program does not find this value, it retrieves it from a file. We will present a formal definition of this contract and an algorithm to verify it in Section 3.3.

Usage There are different use cases of purely resilient try-catch blocks. We have presented the use case of caching for pure resilience in Listing 3. One can use purely resilient try-catch blocks for performance reasons: a catch block can be a functionally equivalent yet slower alternative. The efficient and more risky implementation of the try block is tried first, and in case of an exception, the catch block takes over to produce a correct result. Optionality is another reason for pure resilience. Whether or not an exception occurs during the execution of an optional feature in a try block, the program state is valid and allows the execution to proceed normally after the execution of the try-block.

Discussion The difference between source-independence and pure resilience is as follows. Source-independence means that *under error* the try-catch has always the same observable behavior. In contrast, pure resilience means that *in nominal mode and under error* the try-catch block has always the same observable behavior. This shows that pure resilience subsumes source-independence: by construction, purely resilient catch blocks are source-independent. The pure resilience contract is a loose translation of the concept of recovery block [13] in mainstream programming languages. A purely resilient try-catch is able to handle unanticipated exceptions in any situation while still performing the expected operation. Hence it is resilient w.r.t. the chosen definition of resilience (see Section 2.2).

Although the "pure resilience" contract is strong, we will show in Section 4 that we observe purely resilient try-catch blocks in reality, without any dedicated search: the dataset under consideration has been set up independently of this concern. The source independence and pure resilience contracts are not meant to be mandatory. The try-catch blocks can satisfy one, both, or none. We only argue that satisfying them is better from the viewpoint of resilience, according to our technical definition of resilience given in Section 2.2.

3.2 The Short-circuit Testing Algorithm

We now present a technique, called "short-circuit testing", which allows one to find source-independent and purely-resilient try-catch blocks.

Definition Short-circuit testing consists of dynamically injecting exceptions during the test suite execution in order to analyze the resilience of try-catch blocks.

The system under test is instrumented so that the effects of injected exceptions are logged. This data is next analyzed to verify whether a try-catch block satisfies or violates the two contracts aforementioned. The injected exceptions represent unanticipated situations.

According to our definition of resilience and test suites as specification, everything that is in the test suite is anticipated. To identify unanticipated exceptions, we need to artificially create runtime cases that are not in the test suite. This is exactly the key point of short-circuit testing: it complements the test suite with unanticipated scenarios. Short-circuit testing allows us to study the resilience of try-catch blocks in unanticipated scenarios.

We call this technique "short-circuit testing" because it ressembles electrical short-circuits: when an exception is injected, the code of the try block is somehow short-circuited. The name of software short-circuit is also used in the Hystrix resilience library[4].

What can we say about a try-catch when a test passes while injecting an exception in it? We use the test suite as an oracle of execution correctness: if a test case passes under injection, the new behavior triggered by the injected exception is in accordance with the specification. Otherwise, if the test case fails, the new behavior is detected as incorrect by the test suite.

3.2.1 Algorithm

Our algorithm for short-circuit testing is given in Figure 1. Our algorithm needs an application A and its test suite TS.

First, a static analysis extracts the list of existing try-catch blocks. For instance, the system extracts that method foo() contains one try with two associated catch blocks: they form two try-catch blocks (see Section 3.1). In addition, we also need to know which test cases specify which try-catch blocks, i.e. the correspondence between test cases and try-catch blocks: a test case is said to specify a try-catch if it uses it. To perform this, the algorithm collects data about the standard run of the test suite under consideration. For instance, the system learns that the try block in method foo() is executed on the execution of test #1 and #5. The standard run of short-circuit testing also collects fine-grain data about the occurrences of exceptions in try-catch blocks during the test suite execution (see Section 3.3.1).

[4] see https://github.com/Netflix/Hystrix

Input: An Application A, a test suite TS specifying the behavior of A.
Output: a matrix M (try-catch *times* test cases, the cells represent test success or failure.

```
begin
    try_catch_list ← static_analysis(A)          ▷ retrieve all the try-catch of the application
    standard_behavior ← standard_run(TS)         ▷ get test colors and try-catch behaviors
    for t ∈ try_catch_list                       ▷ For each try-catch tc in the application
    do
        prepare_injection(tc)                    ▷ prepare the try-catch tc by setting an injector
                                                 ▷ which will throw an exception of the type caught by tc
                                                 ▷ at the beginning of each execution of the try tc
        as ← get_test_using(tc, standard_behavior)   ▷ retrieve all tests in the test suite TS
                                                     ▷ using the try of the try-catch tc
        for a ∈ as                               ▷ For all test a which use the current try
        do
            pass ← run_test_with_injection(a)    ▷ get the result of the test under injection
            M[tc, a] = pass                      ▷ store the result of the test a under injection in tc
    return M
```

Figure 1: The Short-Circuit Testing Algorithm. Exception injection is used to collect data about the behavior of catch blocks.

Then, the algorithm loops over the try-catch pairs (recall that a try with n catch blocks is split into n conceptual pairs of try/catch). For each try-catch pair, the set of test cases using t, called as, is extracted in the monitoring data of the standard run. The algorithm executes each one of these tests while injecting an exception at the beginning of the try under analysis. This simulates the worst-case exception, worst-case in the sense that it discards the whole code of the try block. The type of the injected exception is the statically declared type of the catch block. For instance, if the catch block catches "ArrayIndexOutOfBoundsException", an instance of "ArrayIndexOutOfBoundsException" is thrown.

Consequently, if the number of catch blocks corresponding to the executed try block is N, there is one static analysis, one full run of the test suite and N runs of as. In our example, the system runs its analysis, and executes the full test suite once. Then it runs tests #1 and #5 with fault injection twice. The first time the injected exception goes in the first catch block, the second time, it goes, thanks to typing, in the second catch block.

3.2.2 Output of Short-circuit Testing

The output of our short-circuit testing algorithm is a matrix M which represents the result of each test case under injection (for each try-catch). M is a matrix of boolean values where each row represents a try-catch block, and each column represents a test case. A cell in the matrix indicates whether the test case passes with exception injection in the corresponding try-catch. This matrix is used to evaluate the exception contract predicates described next in Section 3.3.

Short-circuit testing is performed with source code transformations. Monitoring and fault injection code is added to the application under analysis. Listing 4 illustrates how this is implemented. The injected code is able to throw an exception in a context dependent manner. The injector is driven by an exception injection controller at runtime.

3.3 Resilience Predicates

We now describe four predicates that are evaluated on each row of the matrix to assess whether: the try-catch is source-independent (contract satisfaction), the try-catch is source-dependent (contract violation), the try-catch is purely-resilient (contract satisfaction), the try-catch is not purely-resilient (contract violation).

As hinted here, there is no one single predicate p for which $contract[x] = p[x]$ and $\neg contract[x] = \neg p[x]$. For both contracts, there are some cases where the short-circuit testing procedure yields not enough data to decide whether the contract is satisfied or violated. The principle of the excluded third (*principium tertii exclusi*) does not apply in our case.

3.3.1 Definition of Try-catch Usages

Our exception contracts are defined on top of the notion of "try-catch usages". A try-catch usage refers to the execution behavior of try-catch blocks with respect to exceptions. We define three kinds of try-catch usages as follows: (1) No exception is thrown during the execution of the try block (called "pink usage") and the test case passes; (2) An exception is thrown during the execution of the try block and this exception is caught by the catch (called "white usage") and the test case passes; (3) An

exception is thrown during the execution of the try block but this exception is not caught by the catch (called “blue usage" – this exception may be caught later or expected by the test case if it specifies an error case).

In these usages, the principle of the excluded third (*principium tertii exclusi*) applies: a try-catch usage is either pink or white or blue. Note that a single try-catch can be executed multiple times, with different try-catch usages, even in one single test. This information is used later in this Section to verify the contracts.

3.3.2 Source Independence Predicate

The decision problem is formulated as: given a try-catch and a test suite, does the source-independence contract hold? The decision procedure relies on two predicates.

Predicate #1 ($source\_independent[x]$): Satisfaction of the source independence contract: A try-catch x is source independent if and only if for all test cases that execute the corresponding catch block ($white\_usage$), it still passes when one throws an exception at the worst-case location in the corresponding try block.

Formally, this reads as:

$$source\_independent[x] = \forall a \in A_x | \forall u_a \in usages\_in(x,a) | (is\_white\_usage[u_a] \implies pass\_with\_injection[a,x]) \quad (1)$$

In this formula, x refers to a try-catch (a try and its corresponding catch block), A_x is the set of all tests executing x (passing in the try block), u is a try-catch usage, i.e. a particular execution of a given try-catch block, $usages\_in(x,a)$ returns the runtime usages of try-catch x in the test case a, $is\_white\_usage[u]$ evaluates to true if and only if an exception is thrown in the try block and the catch intercepts it, $pass\_with\_injection$ evaluates to true if and only if the test case t passes with exception injection in try-catch x.

Predicate #2 ($source\_dependent[x]$):Violation of the source independence contract: A try-catch x is not source independent if there exists a test case that executes the catch block ($white\_usage$) which fails when one throws an exception at a particular location in the try block.

This is translated as:

$$source\_dependent[x] = \exists a \in A_x | \forall u_a \in usages\_in(x,a) | (is\_white\_usage[u_a] \wedge \neg pass\_with\_injection[a,x]) \quad (2)$$

Pathological cases: By construction $source\_dependent[x]$ and $source\_independent[x]$ cannot be evaluated to true at the same time (the decision procedure is sound). If $source\_dependent[x]$ and $source\_independent[x]$ are both evaluated to false, it means that the procedure yields not enough data to decide whether the contract is satisfied or violated.

```
1  try{
2     // injected code
3     if(Controller.isCurrentTryCatchWithInjection())
4       if(Controller.currentInjectedExceptionType() ==
              Type01Exception.class ){
5           throw new Type01Exception();
6       }else if(Controller.currentInjectedExceptionType() ==
              Type02Exception.class ){
7           throw new Type02Exception();
8       }
9
10    ... //normal try body
11 ...
12 } catch (Type01Exception t1e) {
13 ... //normal catch body
14 } catch (Type02Exception t2e) {
15 ... //normal catch body
16 }
```

Listing 4: Short-circuit testing is performed with source code injection. The injected code is able to throw an exception in a context dependent manner. The injector can be driven at runtime.

3.3.3 Pure Resilience Predicate

The decision problem is formulated as: given a try-catch and a test suite, does the pure-resilience contract hold? The decision procedure relies on two predicates.

Predicate #3 ($resilient[x]$): Satisfaction of the pure resilience contract A try-catch x is purely resilient if it is covered by at least one pink usage and all test cases that executes the try block pass when one throws an exception at the worst-case location in the corresponding try block. In other words, this predicate holds when all tests pass even if one completely discards the execution of the try block.

Loosely speaking, a purely resilient catch block is a “perfect plan B".

This is translated as:

$$resilient[x] = (\forall a \in A_x | pass\_with\_injection[a,x]) \wedge (\exists a \in A_x | \exists u_a \in usages\_in(x,a) | is\_pink\_usage[u_a]) \quad (3)$$

where is_pink_usage[u] evaluates to true if and only if no exception is thrown in the try block.

Predicate #4 ($not\_resilient[x]$): Violation of the pure resilience contract A try-catch x is not purely resilient if there exists a failing test case when one throws the exception at a particular location in the corresponding try block.

This predicate reads as:

$$not\_resilient[x] = \exists a \in A_c | \neg pass\_with\_injection[a,x] \quad (4)$$

Pathological cases By construction $resilient[x]$ and $not\_resilient[x]$ cannot be evaluated to true at the same time (the decision procedure is sound). Once again, if they are both evaluated to false, it means that the procedure yields not enough data to decide whether the contract is satisfied or violated.

	# executed try-catch	# purely resilient try-catch	# source-independent try-catch	# source-dependent try-catch	Unknown w.r.t. resilience	Unknown w.r.t. source independence	# Stretchable try-catch
commons-lang	49	3/49	18/49	5/49	1/49	26/49	16/18
commons-codec	14	0/14	12/14	0/14	0/14	2/14	12/12
joda time	18	0/18	4/18	0/18	2/18	14/18	4/4
spojo core	1	1/1	1/1	0/1	0/1	0/1	1/1
sonar core	10	0/10	9/10	1/10	1/10	0/10	7/9
sonar plugin	6	0/6	3/6	0/6	0/6	3/6	3/3
jbehave core	42	2/42	7/42	2/42	9/42	33/42	7/7
shindig-java-gadgets	80	2/80	30/80	12/80	21/80	38/80	26/30
shindig-common	21	1/21	8/21	4/21	4/21	9/21	8/8
total	241	9	92	24	38	125	84/92

Table 1: The number of source independent , purely resilient and stretchable catch blocks found with short-circuit testing. Our approach provides developers with new insights on the resilience of their software.

3.4 Improving Software Resilience with Catch Stretching

We have defined two formal criteria of software resilience and an algorithm to verify them (Section 3.3). How to put this knowledge in action?

For both contracts, one can improve the test suite itself. As discussed above, some catch blocks are never executed and others are not sufficiently executed to be able to infer their resilience properties (the pathological cases of Section 3.3). The automated refactoring of the test suite is outside the scope of this paper.

3.4.1 Definition of Catch Stretching

We now aim at improving the resilience against unanticipated exceptions, those exceptions that are not specified in the test suite and even not foreseen by the developers. According to our definition of resilience, this means improving the capability of the software under analysis to correctly handle unanticipated exceptions.

One solution to force the contract satisfaction is to reduce the size of try blocks, so that they only contain statements involving exceptions during test suite execution. This is a trivial solution, it does not improve resilience. It does actually the opposite: less exceptions can be caught.

The other solution is to transform the catch blocks so that they catch more exceptions than before. This is what we call "catch stretching": replacing the type of the caught exceptions. For instance, replacing `catch(FileNotFoundException e)` by `catch(IOException e)`. The extreme of catch stretching is to parametrize the catch with the most generic type of exceptions (e.g. `Throwable` in Java, `Exception` in .NET).

In other words, In situations where unanticipated exceptions appear, the control flow would be broken, and the program would likely crash. This is what we consider as "incorrect". After catch stretching, the program would not crash, and this is what we consider as "better".

Let us now examine why this simple transformation catch is meaningful with respect to unanticipated exceptions.

We claim that *all source-independent catch blocks are candidates to be stretched*. This encompasses purely-resilient try-catch blocks as well since by construction they are also source-independent (see Section 3.1). The reasoning is as follows.

3.4.2 Catch Stretching Under Short-Circuit Testing

By stretching source independent catch-blocks, the result is equivalent under short-circuit testing. In the original case, injected exceptions are of type X and caught by `catch(X e)`. In the stretched case, injected exceptions are of generic type *Exception* and caught by `catch(Exception e)`. In both cases, the input state of the try block is the same (as set by the test case), and the input state of the catch block is the same (since no code has been executed in the try block due to fault injection). Consequently, the output state of the try-catch is exactly the same. Under short-circuit testing, catch stretching yields strictly equivalent results.

3.4.3 Catch Stretching and Test Suite Specification

Let us now consider a standard run of the test suite and a source-independent try-catch. In standard mode, with the original code, there are two cases: either all the exceptions thrown in the try block under consideration are caught by the catch (case A), or at least one exception traverses the try block without being caught because it is of an uncaught type (case B). In both cases, we refer to exceptions normally triggered by the test suite, not injected ones.

In the first case, catch stretching does not change the behavior of the application under test: all exceptions that were caught in this catch block in the original version are still caught in the stretched catch block. In other words, the stretched catch is still correct according to the specification. And it is able to catch many more unanticipated exceptions: it corresponds to our definition of resilience. On those source-independent try-catch of case (A), *catch stretching improves the resilience of the application.*

We now study the second case (case B): there is at least one test case in which the try-catch x under analysis is traversed by an uncaught exception. There are again two possibilities: this uncaught exception bubbles to the test case, which expects the exception (it specifies that an exception must be thrown and asserts that it is actually thrown). If this happens, we don't apply catch stretching. Indeed, it is specified that the exception must bubble, and to respect the specifications we must not modify the try-catch behavior. The other possibility is that the uncaught exception in try-catch x is caught by another try-catch block y later in the stack. When stretching try-catch x, one replaces the recovery code executed by try-catch y by executing the recovery code of try-catch x. However, it may happen that the recovery code of x is different from the recovery code of y, and that consequently, the test case that was passing with the execution of the catch of y (the original mode) fails with the execution of the catch x.

To overcome this issue, we propose to again use the test suite as the correctness oracle. For source-independent try-catch blocks of case B, one stretches the catch to "Exception", one then runs the test suite, and if all tests still pass, we keep the stretched version. As for case A, the stretching enables to handle more unanticipated exceptions while remaining correct with respect to the specification. Stretching source-independent try-catch blocks of both case A and case B improves the resilience.

3.4.4 Summary

To sum up, improving software resilience with catch stretching consists of: First, stretching all source-independent try-catch blocks of case A. Second, for each source-independent try-catch blocks of case B, running the test suite after stretching to check that the transformation has produced correct code according to the specification. Third, running the test suite with all stretched catch blocks to check whether there is no strange interplay between all exceptions.

We will show in Section 4.3 that most (91%) of source-independent try-catch blocks can be safely stretched according to the specification.

4 Empirical Evaluation

We have presented two exception contracts: pure resilience and source independence (Section 3.1). We now evaluate those contracts from an empirical point of view. Can we find real world try-catch blocks for which the corresponding test suite enables us to prove their source independence? Their pure resilience capability? Or to prove that they violate of those exception contracts.

4.1 Dataset

In this paper, we analyze the specification of error-handling in the test suites of 9 Java open-source projects: Apache commons-lang, Apache commons-code, joda-time, Spojo core, Sonar core, Sonar Plugin, JBehave Core, Shindig Java Gadgets and Shinding Common. The selection criteria are as follows. First, the test suite has to be in the top 50 of most tested exceptions according to the SonarSource Nemo ranking[5]. SonarSource is the organization behind the software quality assessment platform "Sonar". The Nemo platform is their show case, where open-source software is continuously tested and analyzed. Second, the test suite has to be runnable within low overhead in terms of dependencies and execution requirements.

The line coverage of the test suites under study has a median of 81%, a minimum of 50% and a maximum of 94%. This dataset contains a total of 767 catch blocks.

4.2 Relevance of Contracts

The experimental protocol is as follows. We run the short-circuit testing algorithm described in Section 3.2 on the 9 reference test suites described before. As seen in Section 3.2, short-circuit testing runs the test suite n times, where n is the number of executed catch blocks in the application. In total, we have thus 241 executions over the 9 test suites of our dataset.

Table 1 presents the results of this experiment. For each project of the dataset and its associated test suite, it gives the number of executed catch blocks during the test suite execution, purely resilient try-catch blocks, source-independent try-catch blocks, and the number of try-catch blocks for which runtime information is not sufficient to assess the truthfulness of our two exception contracts.

4.2.1 Source Independence

Our approach is able to demonstrate that 92 try-catch blocks (sum of the fourth column of Table 1) are source-independent (to the extent of the testing data). This is worth noticing that with no explicit ways for specifying them and no tool support for verifying them, some developers still write catch blocks satisfying this contract.

[5] See http://nemo.sonarsource.org

This shows that our contracts are not purely theoretical: they reflect properties of error-handling code that can be found in real software.

Beyond this, the developers not only write some source-independent catch blocks, they also write test suites that provide enough information to decide with short-circuit testing whether the catch is source-independent or not.

Our approach also identifies 24 try-catch blocks that are source-dependent, i.e. that violate the source-independence predicate. Our approach makes the developers aware that some catch blocks are not independent of the source of the exception: *the catch block implicitly assumes a state resulting from the execution of several statements at the beginning of the try.* Within the development process, this is a warning. The developers can then fix the try or the catch block if they think that this catch block should be source independent or choose to keep them source-dependent, in total awareness. It is out of the scope of this paper to automatically refactor source-dependent try-catch blocks as source-independent.

For instance, a source-dependent catch block of the test suite of sonar-core is shown in Listing 5. Here the "key" statement is the *if (started == false)* (line 6). Indeed, if the call to *super.start()* throws an exception before the variable *started* is set to true (*started = true* line 15), an exception is thrown (line 7). On the contrary, if the same DatabaseException occurs after line 15, the catch block applies some recovery by setting default value (*setEntityManagerFactory*). Often, source-dependent catch blocks contain if/then constructs. To sum-up, short-circuit testing catches assumptions made by the developers, and uncover causality effects between the code executed within the try block and the code of the catch block.

Finally, our approach highlights that, for 24 catch blocks (fifth column of Table 1), there are not enough tests to decide whether the source-independence contract holds. This also increases the developer awareness. This signals to the developers that the test suite is not good enough with respect to assessing this contract. This knowledge is directly actionable: for assessing the contracts, the developer has to write new tests or refactor existing ones. In particular, as discussed above, if the same test case executes several times the same catch block, this may introduce noise to validate the contract or to prove its violation. In this case, the refactoring consists of splitting the test case so that the try/catch block under investigation is executed only once.

```
1  public class MemoryDatabaseColector extends
       AbstractDatabaseColector {
2    public void start(){
3      try{
4        super.start(); // code below
5      }catch (DatabaseException ex) {
6        if (started==false) // this is the source-dependence
7          throw ex;
8        setEntityManagerFactory();
9      }}}
10
11 public void start(){
12   ...
13   // depending on the execution of the following statement
14   // the catch block of the caller has a different behavior
15   started = true;
16   ...}}
```

Listing 5: A Source-Dependent Try-Catch Found in Sonar-core using Short Circuit Testing.

4.2.2 Pure Resilience

We now examine the pure-resilience contracts. In our experiment, we have found 9 purely resilient try-catch blocks in our dataset. The distribution by application is shown in the third column of Table 1.

Listing 6 shows a purely resilient try-catch block found in project spojo-core using short-circuit testing. The code has been slightly modified for sake of readability. The task of the try-catch block is to return an instantiable Collection class which is compatible with the class of a prototype object. The plan A consists of checking that the class of the prototype object has an accessible constructor (simply by calling `getDeclaredConstructor`). If there is no such constructor, the method call throws an exception. In this case, the catch block comes to the rescue and chooses from a list of known instantiable collection classes one that is compatible with the type of the prototype object. According to the test suite, the try-catch is purely resilient: always executing plan B yields passing test cases.

The pure resilience is much stronger than the source independence contract. While the former states that the catch has the same behavior wherever the exception comes from, the latter states that the correctness as specified by the test suite is not impacted in presence of unanticipated exceptions. Consequently, it is normal to observe far fewer try-catch blocks verifying the pure resilience contract compared to the source-independent contract. Despite the strength of the contract, this contract also covers a reality: perfect alternatives, ideal plans B exist in real code.

One also sees that there are some try-catch blocks for which there is not enough execution data to assess whether they are purely resilient or not. This happens when a try-catch is only executed in white try-catch usages and in no pink try-catch usage. By short-circuiting the white try-catch usages (those with internally caught exceptions), one proves it source-independence, but we also need to short-circuit a nominal pink usage of this try-catch to assess that plan B (of the catch block) works instead of plan A (of the try block). This fact is surprising: this shows that some try-catch blocks are only specified in error mode (where exceptions are thrown) and not in nominal mode (with the try completing with no thrown exception). This also increases the awareness of

```java
// task of try−catch:
// given a prototype object
Class clazz = prototype.getClass();
// return a Collection class that has an accessible constructor
// which is compatible with the prototype's class
try {
    // plan A: returns the prototype's class if a constructor exists
    prototype.getDeclaredConstructor();
    return clazz;
} catch (NoSuchMethodException e) {
    // plan B: returns a known instantiable collection
    // which is compatible with the prototype's class
    if (LinkedList.class.isAssignableFrom(clazz)) {
        return LinkedList.class;
    } else if (List.class.isAssignableFrom(clazz)) {
        return ArrayList.class;
    } else if (SortedSet.class.isAssignableFrom(clazz)) {
        return TreeSet.class;
    } else {
        return LinkedHashSet.class;
    }
}
```

Listing 6: A Purely-Resilient Try-Catch Found in spojo-core (see SpojoUtils.java)

the developers: for those catch blocks, test cases should be written to specify the nominal usage.

4.3 Catch Stretching

We look at whether, among the 92 source-independent try-catch blocks of our dataset, we can find stretchable ones (stretchable in the sense of Section 3.4, i.e. for which the caught exception can be set to "Exception"). We use source code transformation and the algorithm described in Section 3.4.

The last column of Table 1 gives the number of stretchable try-catch blocks out of the number of source-independent try-catch blocks. For instance, in commons-lang, we have found 18 candidates source-independent try-catch blocks. Sixteen (16/18) of them can be safely stretched: all test cases pass after stretching.

Table 1 indicates two results. First, most (91%) of the source-independent try-catch blocks can be stretched to catch all exceptions. In this case, the resulting transformed code is able to catch more unanticipated exceptions while remaining correct with respect to the specification.

Second, there are also try-catch blocks for which catch stretching does not work. As explained in Section 3.4, this corresponds to the the case where the stretching results in hiding correct recovery code (w.r.t. to the specification), with new one (the code of the stretched catch) that proves unable to recover from a traversing exception.

In our dataset, we encounter all cases discussed in Section 3.4. For instance in joda-time, all four source-independent try-catch blocks represent are never traversed by an exception – case A of Section 3.4.3. (for instance the one at line 560 of class ZoneInfoCompiler). We have shown that analytically, they can safely be stretched. We have run the test suite after stretching, all tests pass.

We have observed the two variations of case B (try-catch blocks traversed by exceptions in the original code). For instance, in sonar-core, by stretching a NonUniqueResultException catch to the most generic exception type, an IllegalStateException is caught. However, this is an incorrect transformation that results in one failing test case.

Finally, we discuss the last and most interesting case. In commons-lang, the try-catch at line 826 of class ClassUtils can only catch a ClassNotFoundException but is traversed by a NullPointerException during the execution of the test ClassUtilsTest.testGetClassInvalidArguments. By stretching ClassNotFoundException to the most generic exception type, the NullPointerException is caught: the catch block execution replaces another catch block upper in the stack. Although the stretching modifies the test case execution, the test suite passes, this means that the stretching is correct with respect to the test suite.

4.4 Summary

To sum up, this empirical evaluation has shown that the short-circuit testing approach of exception contracts enables to increase the knowledge one has on a piece of software. First, it indicates source-independent and purely resilient try-catch blocks. This knowledge is actionable: those catch blocks can be safely stretched to catch any type of exceptions. Second, it indicates source-dependent try-catch blocks. This knowledge is actionable: it says that the error-handling should be refactored so as to resist to unanticipated errors. Third, it indicates "unknown" try-catch blocks. This knowledge is actionable: it says that the test suite should be extended and/or refactored to support automated analysis of exception-handling.

5 Discussion

There are two important points to be discussed with respect to the short circuit testing algorithm: where the exception is injected and the special case of statically verified checked exceptions in Java (and other languages such as .NET). We discuss at the end of the section the threats to the validity of our empirical results.

5.1 Injection Location

In short-circuit testing we inject worst-case exceptions (see Section 3.2), where "worst-case" means that we inject exceptions at the beginning of the try block. Thus, the injected exception discards the whole code of the try block.

Another possibility would be to inject exceptions at different locations in the try block (for instance, before

line 1 of the try block, after line 1, after line 2, etc.). Let us call this algorithm "fine-grain injection". It would have the following property. First, it would take longer; instead of executing each test once with worst-case injection, this algorithm would execute each test for each possible injection location. Second, if the satisfaction or violation of the contracts is undecidable with short-circuit testing, it would still be undecidable with fine-grain injection because the verifiability of contracts depends on the test suite coverage of try-catch only and not on the location (see section 3.3). Also, this algorithm could not invalidate the source dependence of catch blocks (if a test case fails with short-circuit, it would still fail with fine-grain injection which encompasses the worst case). The most interesting property of fine-grain injection is that it could show that some catch blocks that are characterized as source independent by short-circuit testing are actually source dependent. Let us show that it is theoretically possible yet unlikely.

Let us consider the code in Listing 7 and a test case which asserts that this method returns 0 in a specific error scenario. In this error scenario, the exception comes from the call to `myMethod()`. In this case, when entering the catch block, `a=true` because the last assigment to `a` is `a=bar()` and `bar` returns `true`. Consequently the catch block returns `null` and the test passes. With short-circuit testing an exception is thrown at *injection location #1* when `a` is still equals to `true` (assigned just before the try block). As a result the catch block still returns `null` as expected by the test, which passes. The behavior of the catch block is the same as the expected one, so short-circuit testing assesses that this catch block is source independent. The fine-grain injection algorithm would identify 2 additionnal injection locations *injection location #2 and #3*. It would execute the test 3 times, each time with injection at a different location. In the first run, it works just as short-circuit testing (injection in *injection location #1*). In the second run an exception is injected at *injection location #2*, thus `a = false` because the last assigment to `a` is `a=foo()` and `foo` returns `false`. Consequently the catch block returns -1 instead of 0. The behavior of the catch is different under injection, the test case fails (-1 returned instead of 0). Because of a failing test case under injection, the catch block under test would be proven source dependent (as said by Predicate #2 in Section 3.3.2).

This example shows that there may be a sequence of program states that are either recoverable or unrecoverable. The state switching occurs not only between the statements of the try block but also within the code executed in the called methods. Consequently, a meaningful fine-grain injection would not only happen at each statement of the try under analysis but also between the statements of the called methods, in a recursive manner. In other words, for worst-case injection, there is only one injection required for reasoning on the resilience, in

```
1   boolean a = true;
2   try {
3       // injection location #1
4       a = foo();//returns false
5       // injection location #2
6       a = bar();//returns true
7       // injection location #3
8       return myMethod();//exception thrown
9   } catch (Exception e) {
10      if (a){
11          return 0;
12      }else{
13          return -1;
14      }
15  }
```

Listing 7: An example of try-catch block categorized differently by worst-case exception injection (short-circuit testing) and fine-grain exception injection

fine grain injection, there is a myriad of injected exceptions for the same try block (it is common to have one thousand or more recursively executed statements within the same try block execution[6]). Hence, we recommend worst-case injection, which is sound for the detection of contract violations and has a predictable and affordable computational cost (one test suite run by analyzed try-catch).

The same reasoning applies to single-statement try blocks (a try block with only one line of code). It is not equivalent to receive a thrown exception from this single statement and to inject a worst-case exception at the beginning of the try block. Within the recursively called methods and executed statements, there may still be a sequence of recoverable and unrecoverable states. Short-circuit testing of single-statement try blocks ensures that the catch block makes no assumption on what is internally done during the execution of the statement. Indeed, the real-world code of Listing 5 shows an example of a single-statement try block, which is identified as source-dependent thanks to short circuit testing.

5.2 Checked and Unchecked

Short-circuit testing is independent of the programming language and is applicable to the common exception models (C++ , Python, Ruby, ...). However, our experiments are done in Java, where there are two kinds of exceptions : checked and unchecked. Unchecked exception are standard exceptions, similar to other languages, and may occur anywhere in the code. On the contrary, checked exceptions are subject to explicit declaration and static verification. Consequently, the developers are always aware of all possible locations where a checked exception may occur (otherwise the code does not compile).

[6]Note that we do not take in consideration that an exception may happen within a call to the standard library. In theory, all intermediate states during a library call (and inside a native call) are also candidate to exception injection, which is really expensive to compute.

```
1  try {
2      parametriseStep(); // <-- set the expected value to
           StepAsString
3      return successful(stepAsString).withParameterValues(
           parametrisedStep);
4  } catch (InvocationTargetException e) {
5      return failed(stepAsString, new UUIDExceptionWrapper(
           stepAsString, failureCause)).withParameterValues(
           parametrisedStep);
6  }
```

Listing 8: Violation of the Source Independence Contract in jbehave-core (StepCreator.java).

This statically verified exceptional behavior has an impact on short-circuit testing.

Short-circuit testing injects the exception at the beginning of the try block. For checked exceptions, it results in a statically and dynamically valid thrown exception (we are still in the scope of the try block). From the viewpoint of the programmer it is artificial: the programmer knows that this exception can only happen as of the first statement that declares throwing this checked exception. In other words, she is sure that it is impossible that this exception comes from the beginning of the try. However, even if the exception itself is artificial, it enables us to assess the resilience contracts which allows us to perform catch stretching (see Section 3.4).

With catch stretching, in the original case, injected exceptions are of type X and caught by `catch(X e)`. In the stretched version, the caught exception is of the most generic type *Exception*. Hence, the stretched version of a catch block rescuing a checked exception is also able to rescue an unchecked exception. Contrary to the injected checked exception, those unchecked exceptions are not impossible, they may happen in unanticipated scenarios. Consequently, short-circuit testing is also valuable for checked exceptions because it indirectly allows improving the resilience against unanticipated unchecked exceptions.

5.3 Threats to Validity

The main threat to the construct validity lies in bugs in our implementation of short-circuit testing. To mitigate this threat, over the course of our experiments, we have regularly thoroughly analyzed try-catch blocks for which satisfaction or violation of one of the contracts was identified. The analysis consists of reading the try-catch, the surrounding code and the test cases that execute it. In the end, this gives us good confidence that there is no serious bug in the experimental code.

The internal validity of our experiment is threatened if the test case behavior we observe is not causally connected to the exception that are thrown. Since we use mature determistic software (a Java Virtual Machine) and a deterministic dataset, this is actually unlikely that spurious exceptions mix up our results.

Finally, concerning the external validity, we have shown that in open-source Java software, there exists source-dependent, source-independent and purely-resilient try-catch blocks. This may be due to the programming language or the dataset. We believe that it is unlikely since our contracts relate to language and domain-independent concepts (contracts, application state, recovery).

6 Related work

Segal et al. [21, 3] invented Fiat, an early validation system based on fault injection. Their fault model simulates hardware fault (bit changes in memory). Kao et al. [15] have described "Fine", a fault injection system for Unix kernels. It simulates both hardware and operating system software faults. In comparison, we inject high-level software faults (exceptions) in a modern platform (Java). Bieman et al. [5] added assertions in software that can be handled with an "assertion violation" injector. The test driver enumerates different state changes that violate the assertion. Their technique improves branch coverage, especially on error recovery code. This is different from our work since: we do not manually add any information in the system under study (tests or application). Fu et al. [8] described a fault injector for exceptions similar to ours in order to improve catch coverage. In comparison to [5] and [8], we do not aim at improving the coverage but at identifying the try-catch blocks satisfying exception contracts.

Sinha [22] analyzed the effect of exception handling constructs (throw and catch) on different static analyses. In contrast, we use dynamic information for reasoning on the exception handling code. The same authors described [23] a complete tool chain to help programmers working with exceptions. The information we provide (the list of source-independent, purely-resilient try-catch blocks and so forth) is different, complementary and may be subject to be integrated in such a tool.

Candea et al. [7] used exception injection to capture the error-related dependencies between artifacts of an application. They inject checked exceptions as well as 6 runtime, unchecked exceptions. We also use exception injection but for a different goal: verifying the exception contracts.

Ohe et al. [18] described an exception monitoring system that resembles ours. Beyond the monitoring system we also provide a strategy, two contracts and four predicates to verify two exception contracts.

Ghosh and Kelly [10] did a special kind of mutation testing for improving test suites. Their fault model comprises "abend" faults: abnormal ending of catch blocks. It is similar to short-circuiting. We use the term "short-circuit" since it is a precise metaphor of what happens. In comparison, the term "abend" encompasses many more kinds of faults. In our paper, we claim that the new observed behavior resulting from short-circuit test-

ing should not be considered as mutants to be killed. Actually we claim that short-circuiting should remain undetected for sake of source independence and pure resilience: if short-circuiting is detected, it means that at least one try-catch is not source-independent. Consequently, if one values and agrees with the source independence contract, short-circuiting should remain undetected.

Fu and Ryder [9] presented a static analysis for revealing the exception chains (exception encapsulated in one another). In contrast, our approach is a dynamic analysis. We do not focus on exception chains, we propose an analysis of source-independence and pure resilience. Mercadal [16] presented an approach to manage error-handling in a specific domain (pervasive computing). This is forward engineering. On the contrary, we reason on arbitrary legacy Java code, we identify resilient locations and provide techniques to improve their resilience.

Zhang and Elbaum [26] have recently presented an approach that amplifies test to validate exception handling. Their work has been a key source of inspiration for ours. Short-circuit testing is a kind of test amplification. While the technique is the same, the problem domain we explore is really different. They focus on exceptions related to external resources. We focus on any kind of exceptions in order to verify resilience contracts.

7 Conclusion

In this paper, we have explored the concept of software resilience against unanticipated exceptions. We have formalized two resilience properties: source-independence and pure-resilience. We have devised an algorithm, called short-circuit testing, to verify them. Finally, we have proposed a source code transformation called "catch stretching" that improves the ability of the application under analysis to handle unanticipated exceptions. Our empirical evaluation on 9 open-source applications show that those contracts characterize the exceptional behavior of real code: there exist try-catch blocks that satisfy and violate the contracts.

Our future work consists in further exploring how to improve the resilience of software applications: the scope of try blocks can be automatically adapted while still satisfying the test suite; the purely resilient catch blocks could probably be used elsewhere because they have a real recovery power; the resilience oracle has not to be only a test suite, but for example metamorphic relations or production traces.

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] J. H. Barton, E. W. Czeck, Z. Z. Segall, and D. P. Siewiorek. Fault injection experiments using fiat. *IEEE Transactions on Computers*, 39(4):575–582, 1990.

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

[5] J. M. Bieman, D. Dreilinger, and L. Lin. Using fault injection to increase software test coverage. In *Seventh International Symposium on Software Reliability Engineering*, pages 166–174. IEEE, 1996.

[6] B. Cabral and P. Marques. Exception handling: A field study in Java and .Net. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 151–175. Springer, 2007.

[7] G. Candea, M. Delgado, M. Chen, and A. Fox. Automatic failure-path inference: A generic introspection technique for internet applications. In *Proceedings of the 3rd IEEE Workshop on Internet Applications*, pages 132–141. IEEE, 2003.

[8] C. Fu, R. P. Martin, K. Nagaraja, T. D. Nguyen, B. G. Ryder, and D. Wonnacott. Compiler-directed program-fault coverage for highly available java internet services. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2003.

[9] C. Fu and B. G. Ryder. Exception-chain analysis: Revealing exception handling architecture in java server applications. In *Proceedings of the 29th International Conference on Software Engineering*, 2007.

[10] S. Ghosh and J. L. Kelly. Bytecode fault injection for java software. *Journal of Systems and Software*, 81(11):2034–2043, 2008.

[11] J. B. Goodenough. Exception handling: Issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

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

[13] J. J. Horning, H. C. Lauer, P. M. Melliar-Smith, and B. Randell. *A program structure for error detection and recovery*. Springer, 1974.

[14] J.-C. Laprie. From dependability to resilience. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2008.

[15] W. lun Kao, R. K. Iyer, and D. Tang. Fine: A fault injection and monitoring environment for tracing the unix system behavior under faults. *IEEE Trans. Software Eng.*, 19(11):1105–1118, 1993.

[16] J. Mercadal, Q. Enard, C. Consel, and N. Loriant. A domain-specific approach to architecturing error handling in pervasive computing. In *Proceedings of the Conference on Object Oriented Programming Systems Languages and Applications*, 2010.

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

[18] H. Ohe and B.-M. Chang. An exception monitoring system for java. In *Rapid Integration of Software Engineering Techniques*, pages 71–81. Springer, 2005.

[19] P. Rovner. Extending modula-2 to build large, integrated systems. *Software, IEEE*, 3(6):46–57, 1986.

[20] W. Savitch. *JAVA An Introduction to Problem Solving and Programming*. Pearson, 2012.

[21] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, R. Dancey, A. Robinson, and T. Lin. Fiat-fault injection based automated testing environment. In *Proceedings of the Eighteenth International Symposium on Fault-Tolerant Computing*, pages 102–107. IEEE, 1988.

[22] S. Sinha and M. J. Harrold. Analysis and testing of programs with exception handling constructs. *IEEE Transactions on Software Engineering*, 26(9):849–871, 2000.

[23] S. Sinha, A. Orso, and M. J. Harrold. Automated support for development, maintenance, and testing in the presence of implicit flow control. In *Proceedings of the 26th International Conference on Software Engineering*, pages 336–345. IEEE, 2004.

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

[25] K. S. Trivedi, D. S. Kim, and R. Ghosh. Resilience in computer systems and networks. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, ICCAD '09, pages 74–77, New York, NY, USA, 2009. ACM.

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