Automatic Detection and Masking of Non-Atomic Exception Handling

Christof FETZER Karin HÖGSTEDT
AT&T Labs – Research
Florham Park, NJ, USA
{christof, karin}@research.att.com

Pascal FELBER
Institut EURECOM
Sophia Antipolis, France
felber@eurecom.fr

Abstract

Developing robust applications is a challenging task. Although modern programming languages like C++ and Java provide sophisticated exception handling mechanisms to detect and correct runtime error conditions, exception handling code must still be programmed with care to preserve application consistency. In particular, exception handling is only effective if the premature termination of a method due to an exception does not leave an object in an inconsistent state. We address this issue by introducing the notion of failure atomicity *in the context of exceptions and novel techniques to* automatically *detect and mask* non-atomic exception handling. *These techniques can be applied to applications written in several different programming languages, and can be used even when the application's source code is not available. We perform experimental evaluation on both C++ and Java applications to demonstrate the effectiveness of our techniques and measure the overhead that they introduce.*

1 Introduction

Developing robust software is a challenging task. A robust program has to be able to detect and recover from a variety of faults like the temporary disconnection of communication links, resource exhaustion, and memory corruption. For programmers, robust software has the connotation of *elegance* [17]: robust software has to be able to recover from faults without substantially increasing the code complexity. An increase in code complexity increases the probability of design and coding faults and can thus decrease the robustness of the software.

Language-level *exception handling* mechanisms allow programmers to handle errors with only one test per block of code. In programming languages without exception handling, such as C, programmers have to check for error return codes after each function call. The use of exception handling mechanisms can simplify the development of robust programs.

Although the use of exceptions simplifies the detection of failures, the elegance of language-level exception handling mechanisms might lead to the neglect of recovery issues (for an example, see [6]). The premature exit of a method due to an exception might leave an object in an inconsistent state. If this inconsistency is not solved in the error handling code, it might prevent a later recovery, and thus decrease the robustness of the program. In this paper, we show how to automatically detect and correct such state inconsistencies.

Problem Description. Modern programming languages, like C++ and Java, provide explicit exception handling support. When a semantic constraint is violated or when some exceptional error condition occurs, an exception is *thrown*. This causes a non-local transfer of control from the point where the exception occurred to a point, specified by the programmer, where the exception is *caught*. An exception that is not caught in a method is implicitly propagated to the calling method. The use of exceptions is a powerful mechanism that separates functional code from the error handling code and allows a clean path for error propagation. It facilitates the development of applications that are robust and dependable by design.

Exception handling code must however be programmed carefully to ensure that the application is in a consistent state after catching an exception. Recovery is often based on retrying failed methods. Before retrying, the program might first try to correct the runtime error condition to increase the probability of success. However, for a retry to succeed, a failed method also has to leave changed objects in a consistent state. Consistency is ensured if any modification performed by the method prior to the exception's occurrence is reverted before the exception is propagated to the calling method. This behavior is hard to implement because, when catching exceptions, a programmer has to consider all possible places where an exception might be thrown, and has to make sure that none of these exceptions can cause a state inconsistency.

We address in this paper the challenging issue of ensuring that failed methods always leave objects in a consistent state after throwing an exception. We classify methods as either *failure atomic* or *failure non-atomic*, depending on whether they do or do not preserve state consistency, respectively. Informally, we say that the *exception handling is atomic* if it ensures failure atomicity. Otherwise, we say that *exception handling is non-atomic*. Our main objectives are to find mechanisms that help identify all failure non-atomic methods, and to develop techniques to automatically transform these methods into failure atomic methods.

Approach. To address the issue of failure non-atomic methods, we propose a system to systematically test and validate the exception handling code of applications. Our system automatically injects both declared (i.e., anticipated) and undeclared (i.e., unexpected) exceptions at runtime, and evaluates if the exception handling code ensures failure atomicity. It notifies the programmer about any failure non-atomic method, as in many situations minor code modifications (e.g., changing the order of some instructions, or introducing temporary variables) are sufficient to transform a failure non-atomic method into a failure atomic method. In cases where this is not possible, our system can automatically generate wrappers to render a given method failure atomic with the use of checkpointing and rollback mechanisms.

Our infrastructure for detecting and masking non-atomic exception handling comes in two flavors, which support the C++ and Java programming languages. The C++ version is optimized for performance, but requires access to the application's source code. The Java version is less efficient, as it uses a combination of load-time and runtime reflection mechanisms, but it works with applications for which source code is not available.

The contribution of our paper is twofold. First, we formalize the failure atomicity property in the context of exceptions. Second, we introduce novel techniques for *automatically* detecting and masking non-atomic exception handling. These techniques can be applied to both C++ and Java applications, and do not always require access to the application's source code. We present experimental results that demonstrate the effectiveness and the performance overhead of our techniques.

The organization of this paper is as follows: In Section 2, we first discuss related work. Section 3 introduces the failure atomicity problem, and Section 4 presents our approach for detecting and masking failure non-atomic methods. In Section 5 we discuss the implementation details of our system, and Section 6 elaborates on the performance of our C++ and Java infrastructures. Section 7 concludes the paper.

2 Related Work

Exception handling has been investigated for several decades. Goodenough [14] proposed to add explicit programming language constructs for exception handling in 1975, and Melliar-Smith and Randell [24] introduced the combination of recovery blocks [4] and exceptions to improve the error handling of programs in 1977.

Exception handling is still actively investigated. For example, a complete issue of ACM SIGAda Ada Letters [1] was recently dedicated to exception handling, and a 2001 Springer LNCS book addresses advances in exception handling [26]. One of the major issues addressed by researchers is a better separation of functional code and exception handling code. [25] proposes to combine exception handling and reflection to increase this division. [22] studies the use of aspect-oriented programming for reducing the amount of code related to exception handling.

Although the goal of exception handling code is to increase the robustness of programs, it has been noted by Cristian in [8] that exception handling code is more likely to contain software bugs (called *exception errors* [23]) than any other part of an application. This can be explained intuitively by two factors. First, exceptions introduce significant complexity in the application's control flow, depending on their type and the point where they are thrown. Second, exception handling code is difficult to test because it is executed only rarely and it may be triggered by a wide range of different error conditions. Furthermore, 50% of security vulnerabilities are attributed to exception handling failures [23]. Therefore, eliminating exception failures would not only lead to more robust programs, but also more secure programs.

Several approaches have been proposed to address the issue of exception errors [23]: code reviews, dependability cases, group collaboration, design diversity, and testing. Testing typically results in less coverage for the exception handling code than for the functional code [8]. The effectiveness of dependability cases, design diversity, and collaboration for reducing exception handling errors has been studied in [23]. In this paper we introduce a novel approach based on exception injection to address certain kinds of exception errors. We do not consider our approach as a replacement of other approaches; we rather believe that it complements techniques like dependability cases and collaboration. The advantages of our approach lie essentially in its highly automated operation and fast detection of functions that contain certain exception errors.

The robustness of programs can be evaluated using fault injection techniques [3]. There exist software-implemented, hardware-implemented, and simulation-based fault injectors. Our tool performs software-implemented fault injections. Software-implemented fault injectors have been investigated

for various types of failures, such as memory corruption [27, 2], invalid arguments [20], or both [9]. There are also various techniques for injecting faults. Some tools like FERRARI [18] and Xception [7] inject faults without modifying the applications. Tools like DOCTOR [16] modify the application at compile time, and others during runtime.

Our tool injects faults in the form of exceptions, by modifying the application either at compile time or at load time. Unlike FIG [5], which tests the error handling of applications by returning error codes to system calls, our tool only injects application-level exceptions.

Our tool does not only evaluate the robustness of programs by performing exception injections, but it also automatically corrects the problems discovered by the fault injections. The automatic wrapping of shared libraries based on injection results has been previously demonstrated in [11]. In this paper, we address different types of failures (exception handling vs. invalid arguments) and hence, we use different fault injection and wrapping techniques.

3 Problem Description and Motivation

Robust software has to be able to detect and recover from failures that might occur at runtime. One way of performing failure recovery is by taking advantage of the exception handling mechanism that is provided in many programming languages. Using this mechanism, a method can signal to its caller that it has encountered a failure, be it memory depletion or an unexpected result of a calculation, by throwing an exception. The exception can then be caught by the caller, which provides the programmer with an opportunity to recover from the failure and consequently to increase the robustness of the application.

Failure recovery is however likely to fail, unless extreme care is taken during the programming of the exception handling code. Due to the incomplete execution of the method that threw the exception, one or more objects might be in inconsistent states. Unless consistent states are restored, the application might crash or terminate with an incorrect result.

In this paper, we present a system to help programmers detect which methods might leave an object in an inconsistent state when an exception is thrown. Our system can also automatically revert an object back to a consistent state by automating the "checkpoint, execute, and roll-back on exception" idiom, if the programmer so desires. This is further explained in Sections 4 and 5.

Before describing our system in more detail, we formally introduce the notions of *object graphs*, *failure non-atomic methods*, and *failure atomic methods*.

Definition 1. *An* object graph *is a graph where each node is either an object or an instance of a basic data type (like an integer or a pointer). The values of the instance variables of an object are represented as children of the object node. If a node represents the value of a variable, the node is also labeled with the name of the variable. If a node contains a non-null pointer, the node has exactly one child that represents the referenced object or the referenced instance of a basic data type. If two non-null pointers are pointing to the same object or instance, their nodes in the object graph share the same child node. If the node is a null pointer, the node does not have any children.*

The *object graph of object* `o` is the object graph where `o` is the root node.

Definition 2. *Let* `C` *be a class. A method* `m` *of class* `C` *is* failure non-atomic *if there exists an object* `o` *of class* `C` *and an execution E, such that the object graph of* `o` *before* `m` *is invoked on* `o` *in E is different from the object graph of* `o` *right after* `m` *has returned with an exception. A method is* failure atomic *if it is not failure non-atomic.*

4 Approach

Our approach to identify and transform failure non-atomic methods consists of two phases: a *detection* and a *masking* phase. The detection phase uses automated injection of exceptions to identify failure non-atomic methods, and the masking phase transforms failure non-atomic methods into failure atomic methods.

4.1 Detection Phase

The goal of the detection phase is to determine which methods are failure non-atomic. This detection is done with the help of automated experiments.

Automated Experiments. We use automated fault injection experiments to determine if methods are non-atomic. Experiments are run on test programs or end-user applications that call the methods we want to investigate. We automatically transform the code of these programs to inject exceptions at specific points of their execution. As the last step in the detection phase, these exception injector programs are then run to generate a list of the failure non-atomic methods to be used as input to the masking phase. This process consists of steps 1 through 3 as shown in Figure 1.

Step 1: To create an exception injector program P_I from a program P, we first determine which methods are called by P, and for each of them, the exceptions that may be thrown (this includes all the exceptions *declared* as part of the method's signature, as well as generic runtime exceptions that can be thrown by any method). The Analyzer then creates an *injection wrapper* for all methods called during the execution of P.

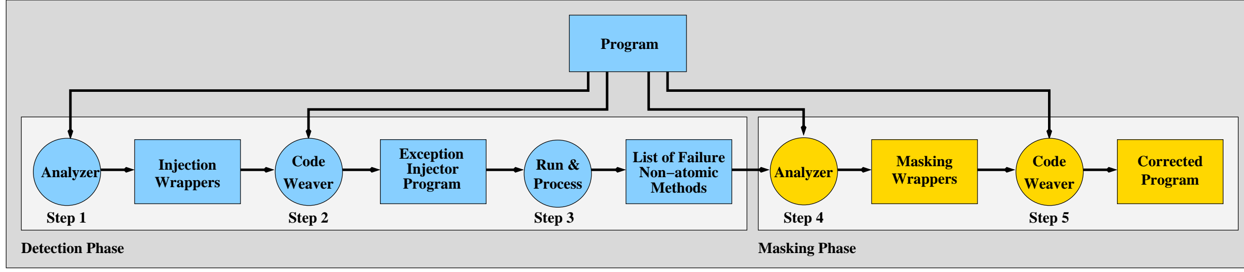


Figure 1: We automatically transform applications to inject exceptions in their execution, and we use the experimental results to correct the applications.

Assume that method m declares exceptions of types E_1, E_2, ... E_k and may also throw runtime exceptions E_{k+1}, ... E_n. The Analyzer creates an injection wrapper `inj_wrapper_m` for m, which either throws one of these exceptions, or calls method m. In the injection wrapper of m, there are n potential injection points as shown in Listing 1. We determine whether to throw an exception at any of these injection points using a global counter (`Point`), incremented every time the control flow reaches a potential injection point; an exception is injected when the counter reaches a preset threshold value (`InjectionPoint`).

```
1  return_type  inj_wrapper_m  (...)  throw (E1, E2  ,...,  Ek) {
2    if (++Point ==  InjectionPoint  ) throw E1();
3    if (++Point ==  InjectionPoint  ) throw E2();
4    ...
5    if (++Point ==  InjectionPoint  ) throw En();
6    objgraph_before  = deep_copy( this );
7    try {
8      return  m (...);
9    } catch   (...) {
10     objgraph_after  = deep_copy( this );
11     if  ( objgraph_before  !=  objgraph_after )
12       mark("m", "nonatomic",  InjectionPoint  );
13     else  // atomic  in  this  call
14       mark("m", "atomic",  InjectionPoint );
15     throw;
16   }
17 }
```

Listing 1: Pseudo-code for the injection wrapper of method `m` (detection phase). This code injects exceptions into callers of `m`.

```
1  return_type  atomic_m (...) {
2    objgraph  = deep_copy( this );
3    try {
4      return  m (...);
5    } catch   (...) {
6      replace ( this ,  objgraph );
7      throw; //  rethrow  exception
8    }
9  }
```

Listing 2: Pseudo-code for the atomicity wrapper of method `m` (masking phase).

Step 2: After the Analyzer has created the injection wrappers for all methods called by P, the Code Weaver makes sure that the wrappers, as opposed to the original methods, are called. Modifications can be performed on the program's source files (source code transformation), or by directly instrumenting the application's code or bytecode (binary code transformation). The result of this transformation is an exception injector program P_I, and the two approaches are discussed in more detail in Sections 5.1 and 5.2, respectively.

Step 3: Once the exception injector program P_I is created, we execute it repeatedly. We increment the threshold `InjectionPoint` before each execution to inject an exception at a different point in each run. Each wrapper intercept all exceptions and checks if the wrapped method is failure non-atomic before propagating the exception to the caller.

To determine whether a method m is failure non-atomic, the injection wrapper `inj_wrapper_m` (Listing 1) makes a deep copy of the state of the invoked object before calling method m (all arguments that are passed in as non-constant references are also part of this copy.) This copy represents a snapshot of the object graph of the invoked object (see Section 3). If m returns with an exception, the wrapper catches the exception and compares the snapshot of the object's previous state with its current state. If both object graphs are identical, we mark the method as failure atomic (for this injection); otherwise, we mark it as failure non-atomic. Since different injections may result in different classifications for a method, we classify a method m as failure atomic if and only if it is never marked as failure non-atomic, i.e., if and only if for each injection the "before" and "after" object graph are identical. The output of this phase is a list of the failure non-atomic methods called in the original program.

4.2 Masking Phase

The goal of the masking phase is to transform the failure non-atomic methods identified during the injection phase into failure atomic methods. By doing so, the resulting program becomes more robust, since the incomplete execution of a method due to an exception does not result in an inconsistent program state. The masking phase consists of two steps (Step 4 and 5 in Figure 1), which are described next.

Step 4: The failure non-atomic methods are automatically transformed into equivalent failure atomic methods. The

Analyzer performs this task by generating an *atomicity wrapper* `atomic_m` for each method in the list of failure non-atomic methods provided by the detection phase. This wrapper exhibits failure atomic behavior to its callers. Its pseudo-code is given in Listing 2.

Step 5: After the Analyzer has generated an atomicity wrapper for each of the method that should be transformed, the Code Weaver transforms the original program P into an equivalent (corrected) program P_C by replacing each call to such a method m by a call to its atomicity wrapper `atomic_m`. This process is similar to the one in Step 2.

The implementation details of both the detection and the masking phases are discussed in Section 5.

4.3 To Wrap or Not To Wrap

There are situations in which a failure non-atomic method should *not* be wrapped during the masking phase. First, although very unlikely, the failure non-atomic behavior of a method might have been intended by the programmer. Since transforming a failure non-atomic method into a failure atomic method changes its semantics, the transformation might *cause* an incorrect result or crash, instead of avoiding it. To deal with this situation, our system provides an easy-to-use web interface that allows the programmer to indicate which methods (that are classified as failure non-atomic) should not be transformed.

Second, a failure non-atomic method might easily be manually transformed into a failure atomic method, e.g., by swapping lines of code or by using temporary variables. In that case, the programmer might prefer to rewrite the method himself, since the resulting code is likely to be more efficient. After the programmer corrects these methods, he can re-run the detection phase to test the modifications.

Third, a method m might be classified as failure non-atomic even though it is impossible for it to exhibit failure non-atomic behavior. This may happen in situations where the programmer has explicitly ruled out that a specific method m can throw exceptions. Because this assumption is not known to our Analyzer, the injection wrapper for method m will contain an injection point (see Section 4.1). Consequently, the callers of method m might be classified as failure non-atomic, due to an exception thrown by m, although such exceptions cannot happen at runtime.

This conservative classification is a consequence of the limitations of our current Analyzer implementation, which does not attempt to determine whether it is possible for a runtime exception to occur in a given method. We plan to address this issue in the future. Meanwhile, it should be noted that this conservative classification does not result in an incorrect program behavior, but merely in an unnecessary loss in performance due to unnecessary checkpointing during the masking phase. To address this limitation, we allow the programmer to indicate that certain methods never throw exceptions using a web interface. All methods that were classified as failure non-atomic solely because of the exceptions injected in the "exception-free" methods are then re-classified as failure atomic. Note however that it is often hard for a programmer to determine whether a method is exception-free, since all the various runtime conditions that might lead to an exception being thrown are not necessarily known.

Fourth, a method might exhibit failure non-atomic behavior only because the methods it calls are failure non-atomic. We call such methods *conditional failure non-atomic* methods:

Definition 3. *A* conditional failure non-atomic *method is a failure non-atomic method that would be failure atomic if all the methods that it calls (directly or indirectly) were failure atomic. All other failure non-atomic methods are* pure failure non-atomic *methods.*

During the execution of the corrected program (produced by the masking phase), all methods called by a conditional failure non-atomic method m will exhibit failure atomic behavior. Thus, by definition, method m is no longer failure non-atomic and it is not necessary to wrap it. Therefore, distinguishing between pure and conditional failure non-atomic methods can help us improve the performance of the corrected program.

To distinguish conditional from pure failure non-atomic methods, we examine the order in which methods were reported as failure non-atomic during exception propagation for each run of the exception injector program (Step 3 in Figure 1). If there exists a run in which method m is the first method to be marked as failure non-atomic, then m is pure failure non-atomic. Indeed, any failure non-atomic method called by m would be detected and reported before m because of the way exceptions propagate from callee to caller (see Listing 1).

4.4 Limitations

The approach that we use to detect and mask failure non-atomic methods has some limitations. First, it does not handle methods with external side effects, e.g., writing to a file. Because external side effects are not covered by the definition of failure atomicity, our approach can neither detect nor mask such methods.

Second, our system does not explicitly deal with concurrent accesses in multi-threaded programs. With applications that incorporate adequate (conservative) concurrency control support, our injection and checkpointing mechanisms should still produce consistent results. For other applications, one could address this limitation by restricting the amount of parallelism in the system and enforcing restrictive concurrency control policies.

5 Implementations

We have investigated two approaches for implementing our system, using source code and binary code program transformation techniques. The first approach requires access to the source code of a program, while the second does not. However, binary code transformation is not necessarily possible with all programming languages, and the resulting instrumented programs generally suffer from higher performance overhead than with source code transformation.

Both kind of transformations can be aided by the use of aspect oriented programming [19], which allow programmers to easily capture and integrate crosscutting concerns in their applications. We have used AspectC++ [28] for our source code transformations, and we plan to use AspectJ [19] for a future version of our Java bytecode transformation engine (AspectJ does not currently implement bytecode weaving).

5.1 Source Code Transformation

We have implemented a first prototype of our system that performs source code transformations to inject and mask non-atomic exception handling in C++ applications. This prototype uses an aspect-oriented language extension for C++ (AspectC++ [28]) for source code weaving. We describe below our implementation along the same five steps as in Section 4.

Step 1: We use the C/C++ interpreter CINT [15] to parse the source code of a given program. We use the type information provided by CINT to generate the checkpointing code and wrappers for each method. The wrappers are implemented as aspects. We then generate, for each class, a function `deep_copy` to checkpoint the state of an instance of that class.

Step 2: We use AspectC++ to transform the source code of the program into an exception injector program. AspectC++ weaves the wrappers (given as aspects) with the source code of the program in such a way that each call to a method m instead calls the wrapper of m.

Step 3: We execute the exception injector program iteratively to inject exceptions at all possible injection points (for the given program input). The results of online atomicity checks are written out to log files by the injection wrappers. These log files are then processed offline to classify each method.

Step 4: As in step 1, we use CINT to create wrappers (implemented as aspects) for all failure non-atomic methods. In addition to the `deep_copy` function, we generate a function `replace` to restore the state of a previously checkpointed object.

Step 5: As in step 2, we use AspectC++ to weave the wrappers into the source program code.

Limitations. Due to restrictions of C++ and the tools we are using, our implementation has a few limitations. First, CINT does not support templates and ignores exception specifications. A better parsing tool could easily solve this problem. Second, checkpointing C++ objects is not trivial. In particular, C++ allows pointer manipulations that make it hard, in some situations, to discover the complete object graph of an object at runtime. While there exist techniques to address this problem (e.g., by checkpointing the whole address space of the process, or by using the underlying memory management interface), these techniques are often prohibitively expensive or complex. Note that, in the worst case, checkpointing incomplete object graphs may impact the completeness of our detection system, but will never cause failure atomic methods to be reported as failure non-atomic.

Third, unlike Java, C++ does not enforce thrown exceptions to be declared as part of the method's signature. Hence, the C++ exception injector might have to inject a wide range of different exception types in application that do not declare exceptions. This problem can be solved using source code analysis or through automated fault injection experiments.

Fourth, one needs to clean up memory that is implicitly discarded when rolling back to an object checkpoint. To do so, our tool adds an automatic reference counting mechanism to objects. However, this mechanism only works for acyclic pointer structures. For cyclic pointer structures one can use an off-the-shelf C++ garbage collector.

5.2 Binary Code Transformation

With languages that offer adequate reflection mechanisms, it is possible to add functionality to an application without having access to its source code, by applying binary code transformations. We have followed this second approach in the Java version of our infrastructure for detecting and masking non-atomic exception handling.

To inject and mask failures in Java classes, we have developed a tool, called the Java Wrapper Generator (JWG), which uses load-time reflection to transparently insert pre- and post-filters to any method of a Java class. These generic filters allow developers to add crosscutting functionality (as with aspect-oriented programming) to *compiled* Java code in a transparent manner. Filters are attached to specific methods at the time the class is loaded by the Java virtual machine, by using bytecode instrumentation techniques based on the BCEL bytecode engineering library [12]. Filters can be installed at the level of the application, individual classes, instances, or methods. They can modify the behavior of a method by catching and throwing exceptions, bypassing execution of the active method, or modifying incoming and outgoing parameters.

The Java implementation of our framework works along the same lines as its C++ counterpart, with just a few notable differences. Wrappers are attached to the application at load-time, by instrumenting the classes' bytecode. These wrappers have been programmed to be generic, i.e., they work with any class; they obtain type information about classes, methods, parameters, and exceptions at runtime using Java's built-in reflection mechanisms. The methods that checkpoint and restore the state of an object are also generic; they essentially perform a deep copy of the object's state using Java's reflection and serialization mechanisms.

Limitations. A major limitation of our Java binary code transformation implementation is that a small set of core Java classes (e.g., strings, integers) cannot be instrumented dynamically. This limitation applies to all systems that perform Java bytecode transformations, and is not specific to our implementation. It can be overcome by instrumenting the bytecode of core classes offline and replacing their default implementations by the instrumented versions.

6 Experimental Results

To validate our exception injection tool, we first developed a set of synthetic "benchmark" applications in C++ and Java. These benchmarks are functionally identical in both languages, and contain the various combinations of (pure/conditional) failure (non-)atomic methods that may be encountered in real applications. We used these benchmarks to make sure that our system correctly detects failure non-atomic methods during the detection phase, and effectively masks them during the masking phase. These applications were used for performance experiments presented in the section.

We then performed stress tests and assessed the robustness of some legacy applications. For that purpose, we tested two widely-used Java libraries implementing regular expressions [13] and collections [21]. Such libraries are basic building blocks of numerous other applications and are thus expected to be robust. We also tested Self★ [10], a component-based framework in C++ that we are currently developing. We ran experiments with several applications that use Self★ to help us detect failure non-atomic methods and improve the robustness of the framework.

Table 1 lists the number of classes and methods used in the applications we used for our experimental evaluation, together with the total number of exceptions injected during the detection phase (note that this value corresponds to the number of method and constructor calls during the execution of the test programs). We ran separate experiments for each individual application; however, because of the inheritance relationships between classes and the reuse of methods, some classes have been tested in several of the experiments.

	Application	#Classes	#Methods	#Injections
C++ Applications	*adaptorChain*	16	44	10122
	stdQ	19	74	9585
	xml2Ctcp	5	19	6513
	xml2Cviasc1	23	102	12135
	xml2Cviasc2	23	89	13959
	xml2xml1	18	70	8068
Java Applications	*CircularList*	8	58	5912
	Dynarray	7	50	2528
	HashedMap	10	40	3271
	HashedSet	8	32	1149
	LLMap	10	41	7543
	LinkedBuffer	8	38	2737
	LinkedList	9	62	7500
	RBMap	11	55	7133
	RBTree	9	51	8056
	RegExp	4	32	1015

Table 1: C++ and Java application statistics.

Experiments were conducted following the methodology described in Section 4: we generated an exception injector program for each application, and ran it once for each method execution in the original program, injecting one exception per run. The C++ experiments were run on a 866 MHz Pentium 3 Linux machine (kernel 2.4.18) with 512 MB of memory and the Java tests were run using Java 1.4 on a 1.7 GHz Pentium 4 Windows 2000 machine with 512 MB of memory.

6.1 Fault Injection Results

We first computed the proportion of the methods defined and used in our test applications that are failure atomic, conditional failure non-atomic, and pure failure non-atomic.

The C++ results, presented in Figures 2(a), show that the proportion of "problematic" methods, i.e., those that are pure failure non-atomic, remains pretty small. This may indicate that the Self★ applications tested have been programmed carefully, with failure atomicity in mind. In contrast, the Java results, presented in Figure 3(a), exhibit a different trend. The proportion of pure failure non-atomic, is pretty high, as it averages 20% in the considered applications. The proportion of conditional failure non-atomic methods is smaller, but still significant. These relatively high numbers tell us that our system is indeed needed, and that the programmer could eliminate many potential problems by, either manually or automatically, making these methods failure atomic. Using the input of the fault injector, we managed indeed to reduce the number of pure failure non-atomic methods in the Java "LinkedList" application from 18 (representing 7.8% of the calls) to 3 (less than 0.2% of the calls) with just trivial modification to the code, and by identifying methods that never throw exceptions (see Section 4.3).

Figures 2(b) (C++) and 3(b) (Java) represent the same data, weighted by the number of invocations to each method.

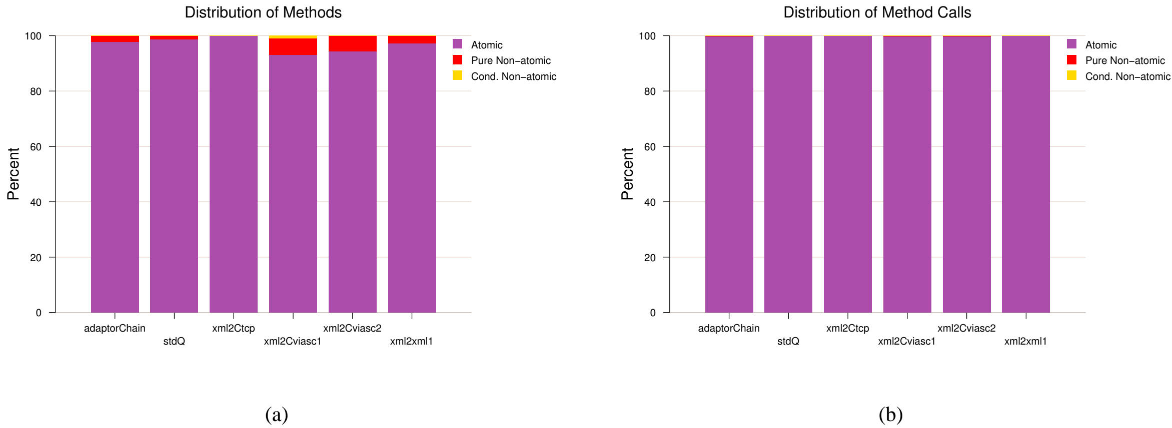


Figure 2: Method classification as a percentage of the number of (a) methods defined and used, and (b) method calls, in each C++ application.

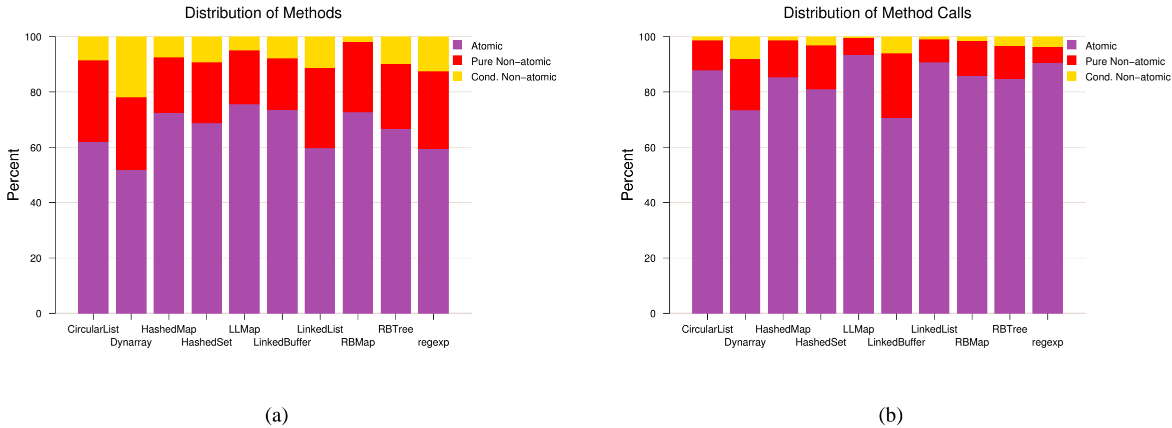


Figure 3: Method classification as percentage of the number of (a) methods defined and used, and (b) method calls, in each Java application.

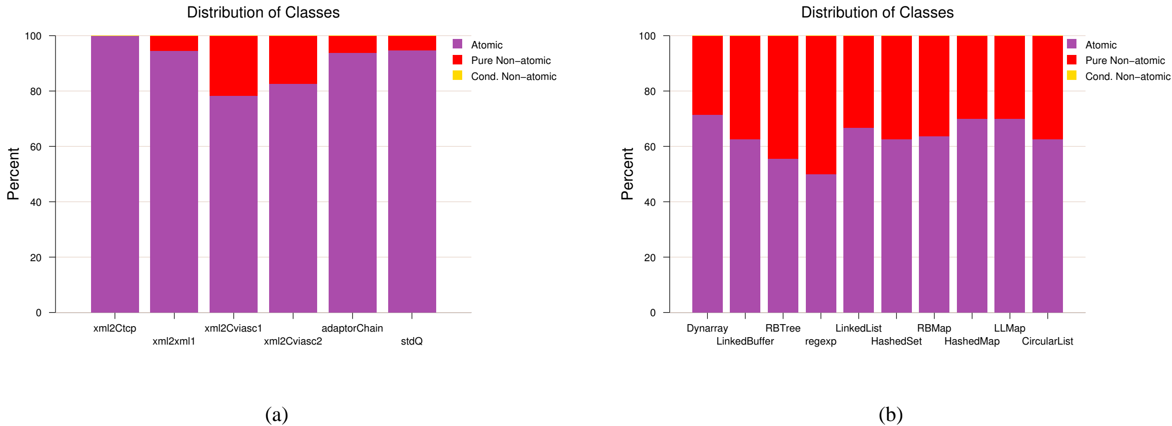
Results show that failure non-atomic methods are called (proportionally) less frequently than failure atomic methods. This trend may be explained by the fact that bugs in methods frequently called are more likely to have been discovered and fixed by the developer. Since problems in methods that are infrequently called are harder to detect during normal operation, our tool is quite valuable in helping a programmer find the remaining bugs in a program. For example, the pure failure non-atomic methods of the "xml2Cviasc" applications are called very rarely, and would probably not have been discovered without the automated exception injections of our system.

Figure 4 shows the proportion of the classes in our test applications that are failure atomic (i.e., only contain failure-atomic methods), pure failure non-atomic (i.e., contain at least one pure failure non-atomic method), and conditional failure non-atomic (i.e., all other classes). The results clearly demonstrate that failure non-atomic methods are not confined in just a few classes, but spread across a significant proportion of the classes (up to 25% for C++ tests, and from 30 to 50% for Java tests).

6.2 Fault Masking Results

The performance of our automated masking mechanism is highly dependent of the frequency of calls to the transformed methods (see Figure 5). Obviously, we have to pay a higher performance penalty as the percentage of calls to the transformed methods increases. The overhead also grows with the size of the checkpoints. As there is no upper bound on the size of objects, this overhead cannot be bounded.

Nevertheless, in the programs we have investigated, we have observed that the checkpoint sizes and the percentage of failure non-atomic method calls remain small. For example, the largest percentage of calls to failure non-atomic methods in our C++ applications was less than 0.4% (Figure 2(b)). In the Java programs, the pure non-atomic methods that we could not easily render failure atomic (by performing trivial modifications) accounted for less than 0.2% of the calls. As long as the object sizes and the percentage



(a) (b)

Figure 4: Distribution of the classes as a percentage of the number of classes defined by the (a) C++, and (b) Java applications.

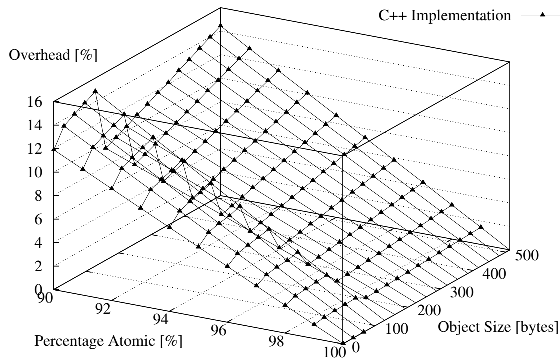


Figure 5: Performance overhead of C++ masking as a function of checkpointed object size and percentage of failure atomic method calls. Each data point is the median of 40 runs and the processing time per method in the original program is about $0.5\mu s$.

of failure non-atomic method calls is reasonably small, we can obtain reasonable performance. For very large object sizes, one could use copy-on-write mechanisms to speed up the checkpointing. For high ratios of failure non-atomic method calls, it would be preferable to use the detection phase of our system to manually correct as many of the failure non-atomic methods as possible.

7 Conclusion

In this paper, we have introduced the failure atomicity problem and proposed a system that addresses it. Our system can automatically detect which methods are failure non-atomic, and then automatically turn them into failure atomic methods. To discover failure non-atomic methods, we inject exceptions into each method executed in an application at runtime, and we compare the state of the objects before the method call and after the exception. Methods that cause an object to enter an inconsistent state are classified as failure non-atomic. To transform failure non-atomic methods into failure atomic methods, we take a snapshot of the state of the object before the method is called; if an exception is thrown, we reinstate that state before propagating the exception to the caller.

Our exception injection system alerts the programmer when finding failure non-atomic methods. In many situations, the programmer can correct the problem by applying simple modifications to his code (such as reordering a couple of statements). In other cases, more elaborate modifications are required to implement failure atomicity; in those situations, the programmer can use the automatic masking mechanisms provided by our system.

We have implemented our infrastructure for detecting and masking non-atomic exception handling in both Java and C++. Experimental results have shown that our system is effective and can be of great help for the developer of robust applications.

References

[1] Exception handling for a 21st century programming language proceedings. *ACM SIGAda Ada Letters*, XXI(3), 2001.

[2] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Goofi: Generic object-oriented fault injection tool. In *Proc. International Conference on Dependable Systems and Networks (DSN 2001)*, Gothenburg, Sweden, 2001.

[3] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J. C. Fabre, J. C. Laprie, E. Martin, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Trans. on Software Eng.*, 16(2):166–182, 1990.

[4] J. Xu B. Randell. The evolution of the recovery block concept. In M. Lyu, editor, *Software Fault Tolerance*, pages 1–21. Wiley, 1995.

[5] P. Broadwell, N. Sastry, and J. Traupman. Fig: A prototype tool for online verification of recovery mechanisms. In *ACM ICS SHAMAN Workshop*, Ney York, NC, June 2002.

[6] T. Cargill. Exception handling: A false sense of security. *C++ Report*, 6(9), November-December 1994.

[7] Joao Carreira, Henrique Madeira, and Joao Gabriel Silva. Xception: A technique for the experimental evaluation of dependability in modern computers. *Software Engineering*, 24(2):125–136, 1998.

[8] F. Cristian. Exception handling and tolerance of software faults. In M. Lyu, editor, *Software Fault Tolerance*, pages 81–107. Wiley, 1995.

[9] J.-C. Fabre, M. Rodriguez, J. Arlat, and J.-M. Sizun. Building dependable cots microkernel-based systems using mafalda. In *2000 Pacific Rim International Symposium on Dependable Computing (PRDC'00)*, pages 85–94, Los Angeles, California, December 2000.

[10] C. Fetzer and K. Högstedt. Self*: A component based data-flow oriented framework for pervasive dependability. In *Eighth IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS 2003)*, Jan 2003.

[11] C. Fetzer and Z. Xiao. An automated approach to increasing the robustness of C libraries. In *International Conference on Dependable Systems and Networks*, Washington, DC, June 2002.

[12] The Apache Software Foundation. BCEL: Byte Code Engineering Library. http://jakarta.apache.org/bcel.

[13] The Apache Software Foundation. Regexp. http://jakarta.apache.org/regexp.

[14] John B. Goodenough. Exception handling: issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, 1975.

[15] Masaharu Goto. CINT C/C++ interpreter, available at http://root.cern.ch/root/Cint.html.

[16] S. Han, K. Shin, and H. Rosenberg. Doctor: An integrated software fault injection environment for distributed real-time systems, 1995.

[17] Definition of "robust" in jargon file 4.3.3, 2003.

[18] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. Ferrari: A tool for the validation of system dependability properties. In *Proc. of 22nd International Symposium on Fault Tolerant Computing (FTCS-22)*, pages 336–344, Boston, Massachusetts, 1992. IEEE.

[19] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In Mehmet Akşit and Satoshi Matsuoka, editors, *Proceedings European Conference on Object-Oriented Programming*, volume 1241, pages 220–242. Springer-Verlag, Berlin, Heidelberg, and New York, 1997.

[20] Nathan P. Kropp, Philip J. Koopman Jr., and Daniel P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Symposium on Fault-Tolerant Computing (FTCS)*, pages 230–239, 1998.

[21] Doug Lea. Collections. http://gee.cs.oswego.edu/dl/classes/collections/.

[22] Martin Lippert and Cristina Videira Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd international conference on Software engineering*, pages 418–427. ACM Press, 2000.

[23] R.A. Maxion and R.T. Olszewski. Eliminating exception handling errors with dependability cases: a comparative, empirical study. *IEEE Transactions on Software Engineering*, 26(9):888 – 906, 2000.

[24] P. M. Melliar-Smith and B. Randell. Software reliability: The role of programmed exception handling. In *Proceedings of an ACM conference on Language design for reliable software*, pages 95–100, 1977.

[25] S. E. Mitchell, A. Burns, and A. J. Wellings. Mopping up exceptions. *ACM SIGAda Ada Letters*, XXI(3):80–92, 2001.

[26] A. Romanovsky, C. Dony, J. Lindskov Knudsen, and A. Tripathi, editors. *Advances in Exception Handling Techniques*. Springer Verlag, 2001.

[27] Z. Segall, D. Vrsalovic, D. Siewiorek, D. Yaskin, J. Kownacki, J. Barton, D. Rancey, A. Robinson, and T. Lin. Fiat — fault injection based automated testing environment. In *Proc. 18th Int. Symp. on Fault-Tolerant Computing (FTCS-18)*, pages 102–107, Tokyo, Japan, June 1988.

[28] Olaf Spinczyk, Andreas Gal, and Wolfgang Schrder-Preikschat. AspectC++: an aspect-oriented extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, Sydney, Australia, February 18-21 2002.