Developing Systems that Handle Exceptions

Alexander B. Romanovsky, Christophe Dony, Jorgen Lindskov Knudsen, Amita Tripathi

Developing Systems that Handle Exceptions

Proceedings of ECOOP'05 Workshop on Exception Handling in Object-Oriented Systems , 2005

Editors: ROMANOVSKY A. , DONY C. , KNUDSEN J.-L. , TRIPATHI A.

Technical Report No 05-050

Department of Computer Science

LIRMM (Laboratoire d'Informatique, de Robotique et Micro-Electronique de Montpellier)

Montpellier-II University

161 rue Ada,

34392 Montpellier France

Proceedings of the

ECOOP Workshop

Exception Handling in Object Oriented Systems: Developing Systems that Handle Exceptions

Glasgow, UK
July 25, 2005

ecoop
2005
European Conference
on Object-Oriented Programming

Table of Contents

Introduction

Modern systems are becoming more complex, and the number of exceptional situations they have to cope with is increasing. The most general way of dealing with these problems is employing exception handling techniques. While a number of object-oriented mechanisms for handling exceptions have been proposed, there are still serious problems with applying them in practice due to complexity of exception code design and analysis, to their improper use, to failure to employ exception handling at the appropriate development phases, to shortage of methodologies supporting correct use of exception handling as well as lack of mechanisms specific to various application domains and design paradigms.

Building on what was achieved in ECOOP'2000 and ECOOP'2003 workshops, this year's forum is intended to provide an opportunity to discuss research on exception handling and fault tolerance in all areas of object-based and component-based software development and use. It aims to cover the entire software life cycle as well as various types of applications, techniques and linguistic mechanisms. In particular, the following topics will be discussed:

- Modelling and applications with exceptions. How can such applications be checked, verified and proved? Which formal models are appropriate? We encourage submissions focusing on modelling system exceptional behaviour with UML-like languages and tools.
- Perceived complexity of using and understanding exception handling. Do programmers misuse exception handling? If so, why? Why do programmers and practitioners often believe that exception handling complicates system design and analysis? Why is exception handling the last mechanism for programmers to learn and to use?
- Novel exception handling and fault tolerance solutions for the new computing contexts: mobile and agent-based systems, pervasive and ambient environments, self-repairing, adaptive and open systems, distributed and asynchronous systems, transaction-based and multi-threaded programs.
- Software architectures which support exception handling and design patterns which help to develop systems that handle exceptions systematically.
- Programming constructs for exception handling. Are Java checked exceptions an appropriate solution for module specification? What are the post-Java constructs for exception handling? Which forgotten or unused past solutions should be brought back? Are they suitable for Java or C# like statically typed languages?
- Experience reports which illustrate benefits to be derived from as well as difficulties involved in using exception handling, summarise practical results of employing advanced exception handling models and the best practices in applying exception handling for developing modern applications.

We are pleased to present this workshop program with two invited talks and thirteen papers covering the above topics.

We want to thank Bertrand Meyer and Andrew Black for presenting invited lectures at this workshop. We also want to thank Devdatta Kulkarni for his help in preparing the workshop proceedings.

Workshop chairs:

Alexander Romanovsky, University of Newcastle upon Tyne
Christophe Dony, Universite Montpellier-II
Joergen Lindskov Knudsen, Mjolner Informatics A/S
Anand Tripathi, University of Minnesota, Minneapolis

Workshop Organizers

Alexander Romanovsky
School of Computing Science
University of Newcastle upon Tyne
Newcastle upon Tyne, NE1 7RU, UK
email: alexander.romanovsky@ncl.ac.uk

Christophe Dony
Universite Montpellier-II
LIRMM Laboratory
161 rue Ada
34392 Montpellier Cedex 5, France
email: dony@lirmm.fr
www: http://www.lirmm.fr/~dony

Jorgen Lindskov Knudsen
Mjolner Informatics A/S
Helsingforsgade 27
DK-8200 Arhus N
Denmark
email: jlk@daimi.au.dk
www: http://www.mjolner.dk/~jlk

Anand Tripathi
Department of Computer Science
University of Minnesota
Minneapolis, MN 55455 USA
email: tripathi@cs.umn.edu

ECOOP 2005 Workshop

Exception Handling in Object Oriented Systems:
Developing Systems that Handle Exceptions

July 25, 2005. Glasgow, UK

http://homepages.cs.ncl.ac.uk/alexander.romanovsky/home.formal/ehoos2005.htm

Program

09.00-10.30 Session 1.

09.00-09.05. Welcome (workshop Co-chairs)

09.05-09.50. Bertrand Meyer (ETH Zurich and Eiffel Software). Disciplined exceptions. Invited talk.

09.50-10.10 Denis Caromel, Guillaume Chazarain (INRIA Sophia Antipolis). Robust Exception Handling in an Asynchronous Environment

10.10-10.30 Chen Fu, Barbara G. Ryder (Rutgers University). Testing and Understanding Error Recovery Code in Java Applications

10.30-11.00 - coffee

11.00-12.40. Session 2.

11.00-11.20 Stijn Mostinckx, Jessie Dedecker, Tom Van Cutsem, Wolfgang De Meuter (Vrije Universiteit). Conversations for Ambient Intelligence

11.20-11.50 Jan Ploski, Wilhelm Hasselbring (University of Oldenburg). The Callback Problem in Exception Handling

11.50-12.10 Matti Rintala (Tampere University of Technology). Handling multiple concurrent exceptions in C++ using futures

12.10-12.30 Michael J. Zastre, R. Nigel Horspool (University of Victoria). Two Techniques for Improving the Performance of Exception Handling

12.30-12.50 Alexei Iliasov, Alexander Romanovsky (University of Newcastle upon Tyne). CAMA: Structured Coordination Space and Exception Propagation Mechanism for Mobile Agents

12.50-14.00. Lunch

14.00-15.30 Session 3.

14.00-14.20 Alfredo Capozucca, Barbara Gallina, Nicolas Guelfi, Patrizio Pelliccione (University of Luxembourg). Modeling Exception Handling: a UML2.0 Platform Independent Profile for CAA

14.20-14.40 Donna Malayeri, Jonathan Aldrich (Carnegie Mellon University). Practical Exception Specifications

14.40-15.30. Discussion session: the role of exception handling

Starts with the invited talk: Andrew P. Black (Portland State University). Exception Handling: The Case Against (14.40-15.00 – invited presentation)

15.30-16.00. - coffee

16.00-17.30. Session 4.

16.00-16.20 Anand Tripathi, Devdatta Kulkarni, Tanvir Ahmed (University of Minnesota). Exception Handling Issues in Context Aware Collaboration Systems for Pervasive Computing

16.20-16.40 Luis E. Leyva del Foyo, Pedro Mejia-Alvarez, Dionisio de Niz (CINVESTAV-IPN and ITESO). Aligning Exception handling with design by contract in embedded real-time systems development

16.40-17.00 Fernando Castor Filho, Cecilia Mary Fischer Rubira, Alessandro Garcia (University of Campinas and Lancaster University). A Quantitative Study on the Aspectization of Exception Handling

17.00-17.20 John Tang Boyland (University of Wisconsin-Milwaukee). Position Paper: Handling "Out Of Memory" Errors

17.20-17.30. wrap up

Invited Lecture

Disciplined Exceptions

Bertrand Meyer
ETH Zurich and Eiffel Software

Abstract

The general idea behind exceptions is to notify programs of abnormal cases occurring execution, and allow them to recover. The proper use of exceptions, and the proper design of an exception mechanism in a programming language, require a precise definition of what makes a case "abnormal".

The Eiffel approach follows from an analysis of these issues, based on the concept of contract. Essentially, a contract violation causes the failure of an operation, which in turns interrupts the current execution by triggering an exception. The exception handling mechanism is also a consequence of these observations. Starting from these ideas I will present a general enquiry into the notion of error, and examine how one should handle exceptions in concurrent object-oriented programming.

Robust Exception Handling in an Asynchronous Environment

Denis Caromel and Guillaume Chazarain

INRIA Sophia Antipolis, CNRS - I3S - Univ. Nice Sophia Antipolis
BP 93, 06902 Sophia Antipolis Cedex - France
First.Last@inria.fr
http://ProActive.ObjectWeb.org

Abstract. While distributed computing is becoming more and more mainstream, it becomes apparent that error handling is an aspect that deserves more attention. We place ourselves in the context of distributed objects, and we try to hide the network behind the usual remote method call semantic.
More precisely, we concentrate on the case where method calls are asynchronous, which complicates the error handling. We start with a state of the art in this research field, and then propose our approach, detailing the problems we faced and how we solved them.
Our achievement was to provide the well-known way of handling exceptions using the `try/catch` construction to programs written using asynchronous method calls. Unfortunately, the usage is not totally transparent because of Java limitations. The main part of the approach is to build an exception mask stack following the Java one.

1 Introduction

Before the introduction of exceptions, errors were commonly returned along the same path as normal return values, it was the responsibility of the caller to check the return value. Exceptions, however, have a dedicated channel [1], it is a subpath of the returned value path but it deserves as much interest as its parent. So, the introduction of exceptions was a major milestone, but when dealing with distributed computing they are often neglected. Efforts are under way to improve this situation, for instance with [2] and [3].

We concentrate on the Java implementation of exceptions, which brings two interesting aspects: exceptions are checked thanks to the `throws` annotation, and it's possible to bypass this check using a subclass of `RuntimeException` as the exception type.

The most prominent combination of distributed computing and exceptions may be Java RMI's `RemoteException` which forces every remote method to add a `RemoteException` to its `throws` clause, and the assorted `try/catch` blocks.

These obligations have negative effects: constrained to wrap remote method calls with `try/catch` blocks, programmers sloppily tended to place empty `catch` blocks, thus swallowing the exception. So all of this care in the Java exceptions

system resulted in less robust programs. The solution from the C# camp is to totally get rid of the `throws` keyword, so all exceptions are unchecked. This difference between C# and Java is still subject to discussion, our take is that incomplete `throws` lists should be reported as a warning by the compiler, not an error. So one gets to choose between robust or toy code simply by fixing or ignoring the warning.

2 ProActive

ProActive [4] is a GRID Java library (Source code under LGPL license) for parallel, distributed, and concurrent computing, also featuring mobility and security in a uniform framework. With a reduced set of simple primitives, ProActive provides a comprehensive API allowing to simplify the programming of applications that are distributed on Local Area Network (LAN), on cluster of workstations, or on Internet Grids.

ProActive is used as a base for our implementation, so we present here the aspects needed to understand the context.

As with every library, there is always some confusion as how to call the end user, since it's actually most of the time a programmer. So we use the term programmer as the library user.

ProActive (like RMI) works with the method call abstraction, so we always have a caller and a callee. We name them using the client/server metaphor, although these roles are not static in ProActive, but it is more common in the distributed world.

These method calls are made to active objects. These are objects with a thread to serve requests from their incoming queue. When possible, ProActive method calls are asynchronous. If the method return type is `void` the call is then one-way, in that we will not wait for any reply and the call will be asynchronous in the message passing semantic.

Asynchronous method calls return a placeholder object of type compatible with the actual return type. This placeholder is called a future and is dynamically updated with the real returned result upon availability. Trying to manipulate the future object before its return will result in a wait-by-necessity, that is, the operation will block until the result arrives, at which point the operation will be resumed.

ProActive is developed with some constraints in mind, to which the presented system must adhere. These constraints are that we prevent ourselves from tampering with the bytecode/JVM or preprocessing the source code before compilation. Only pure Java code is allowed. Breaking these constraints makes the programmer's code harder to debug because the executed code is not the one he actually wrote.

2.1 Exception Handling in ProActive

Unlike most other projects presented here, ProActive does not treat all exceptions the same way. It makes a distinction between functional exceptions and non-functional ones. To put it simply, functional exceptions are thrown and

caught in the ProActive code, and non-functional ones are thrown by ProActive to signal problems impacting the ProActive layer. The latter exceptions are caught using a mechanism called NFE (Non-Functional Exceptions) [5] briefly presented in section 2.1.2.

2.1.1 Functional Exceptions

In ProActive, methods that declare exceptions in their signature are always called synchronously. The mechanism presented here is designed to overcome this limitation.

Concerning runtime exceptions, as we cannot predict if a method call will throw one or not we put the potential `RuntimeException` in the future object, and throw it when the future is accessed. For one-way calls, we cannot have such a mechanism (because no reply is expected), so we simply print a stack trace when a `RuntimeException` is caught, this behaviour will be improved by the proposed mechanism.

2.1.2 Non-Functional exceptions

Non-Functional exceptions in ProActive are classified within different categories and handlers are associable with these exception categories at different priority levels. These handlers run on one side of the call or the other depending on the exception type. This mechanism permits to achieve some interesting behavior like a disconnected mode for a mobile environment.

3 Related Work

We can see a trend in most attempts at asynchronous remote method invocation that tackled exceptions, which is the use of callbacks. Basically, a callback is associated with an exception, in one way or another, and when this exception is thrown the callback is called. The aforementioned ProActive NFE mechanism fits into this case. We will see the benefits of this approach, its limitations, and some ways to go past them.

3.1 Exception Callbacks

3.1.1 The JR Programming Language

The JR [6] programming language extends Java to provide a rich concurrency model, based on that of the SR [7] concurrent programming language. JR provides dynamic remote virtual machine creation, dynamic remote object creation, remote method invocation, asynchronous communication, rendezvous, and dynamic process creation.

The extension of interest to us is the `handler` [8] keyword. It is used in combination with the `send` keyword. The latter keyword is used to introduce the asynchronous calls extension. So, with these two keywords a callback is associated with the forthcoming exception at call time. As we are in an object oriented environment, it's not simply a function that is used as a callback, but a whole object implementing callback methods like the `myHandler` instance in Figure 1.

```
IOHandler myHandler = new MyHandler();
...
send readFile("/path/to/my/file") handler myHandler;
...
```

Fig. 1. Example using the `handler` keyword in JR

This annotation is mandatory for every asynchronous call introduced by the `send` primitive that can throw exceptions. The argument to `handler` is an instance of a class responsible for handling any exception that may be thrown according to the method signature. The JR compiler statically checks this. So the handler provides methods for every exception type throwable by the called method. These callbacks receive as a single argument the actual thrown exception.

In this realization, asynchrony is not transparent because it's explicitly requested by the programmer. On the other hand it can be cumbersome to annotate each asynchronous call that can throw exceptions as in the example.

3.1.2 ARMI

The ARMI [9] project covers not just exception handling but aims at providing an asynchronous remote method invocation mechanism for Java. To deal with exceptions, it proposes two solutions, one of which uses a callback mechanism.

ARMI uses future objects, like ProActive, as a way to transparently introduce asynchrony, the callback mechanism works by registering <exception-type, exception-handler> pairs with the future. With this solution they reach the same level of granularity as the JR approach, but the goal is different since asynchrony serves as an optimization here.

3.1.3 Conclusion on Callbacks Mechanisms

The first advantage of callbacks mechanisms is that it's quite easy to implement. The second one is that it gives the feeling of letting the programmer all the freedom to implement his behavior to handle the exception since he chooses the function to be called upon an exceptional event. This feeling is quite misleading because the main usage of exceptions is impossible with callbacks: unwinding the call stack up to a certain point. An asynchronous environment makes this even harder since the call stack may have gone away when the exception is thrown. But when we use asynchrony as an optimization it is important to keep the ability to unwind the stack as this is taken for granted in a synchronous environment.

The other limitation with callbacks is the context representation. When using the `try/catch` pair, the error handling code is written next to the code to protect, so it sounds normal to have access to all local variables when handling the exception. On the other hand, with callbacks this is not possible since these variables are not part of the context information given to the callback.

Modern exception implementations (like the Java one) alleviate this problem by letting the designer build exceptions as first class object, thus adding every

contextual information he finds useful to them. This is not enough since the exception thrower may not anticipate how the exception catcher will handle it. Moreover we argue that the most interesting context is the one provided by the local variables on the stack. For example, the local variables in the `try` block preceding the `catch` handling the exception can be useful in this handling. Unfortunately, this information is inaccessible to an exception callback.

Another advantage offered by the `try/catch` usage in Java is the automatic unlocking of mutexes acquired using the `synchronized` primitive.

3.2 Closures

With exceptions, we have the problem of capturing some environmental context which is handled by closures as present in functional languages. A closure can be assimilated to the association of a function and a captured environment. In Figure 2 we can see a small example of an error handler defined as a closure. Our local variable `some-value` is captured when the `error-handler` closure is created, and when the closure is executed, not only does it sees the actual value of the variable, but it can modify it. The context capture is complete, we just need to unwind the stack thanks to some continuations and we can handle exceptions using callbacks but with the good properties of the standard `try/catch`.

```
; Actually this should do serious work instead of just calling
; the error handler.
(define (try-something-with-error-handler error-handler)
     (error-handler))

; some-value is our local variable we'll have access to.
; Unlike in Java, the error handler code has a read/write
; access to the environment captured when creating the closure.
(let* ([some-value 1]
       [error-handler (lambda ()
                        (if (= some-value 2)
                            (set! some-value 3)))])
     (set! some-value 2)
     (try-something-with-error-handler error-handler)
     (display some-value))

=> 3
```

Fig. 2. Scheme example of a closure

Closures can be simulated in Java with inner classes but the dirty tricks needed to work around the `final` limitation [10] make them impractical to handle exceptions, compared to the `catch` block.

3.3 Exception in the Future

As we are concerned with asynchrony, future objects are commonly used, so another approach to asynchronous exception handling is to save the exception in the future and throw it upon access.

3.3.1 ARMI

This solution is the second one implemented in ARMI. The authors notice that the result may be accessed far away from the actual method call. With this in mind, we lose the benefits of reusing the common `try/catch` model since the exception may be thrown in a totally different `try/catch` block from the one containing the method call.

3.3.2 Java 1.5

In its 1.5 version (code named Tiger), Java features non transparent future objects [11]. One of the differences with the other approaches is that this one is absolutely user explicit. This is certainly a drawback in our context of asynchrony as an optimization, but by forcing the programmer to manually retrieve the resulting object from the future, he won't have any surprise about where his exception is thrown.

3.3.3 Mandala

The Mandala [12] project provides the RAMI [13] package which is an asynchronous RMI implementation. There are different asynchrony models more or less transparent to the programmer, but with various limitations.

It provides both exception handling techniques: callbacks usage, and encapsulation of the exception in the future. In totally transparent asynchronous mode, the exception is handled by simply setting the result in the future to `null` which will trigger a `NullPointerException` upon access to it.

3.4 Other Approaches

3.4.1 Proxyc

Proxyc [14] is another attempt at providing future objects in a transparent way thanks to some bytecode rewriting. Their solution to the exception handling problem is to make synchronous all asynchronous method calls that can throw exceptions. They suggest to use instead some wrapper method that would handle the exception, this wrapper would be an asynchronous call.

They propose another approach using the fact that they are rewriting the bytecode. As they have to insert code to get the actual result from the future object, they introduce the possibility of letting the programmer specify some exception handling code to place at these points.

3.4.2 RMIX

RMIX[15] is a communication framework for Java, based on the Remote Method Invocation (RMI) paradigm. Its feature of concern to us is asynchronous method calls, and its exception handling solution is quite original. The fact that an asynchronous method raised an exception is a sticky bit in the object, no more asynchronous calls are accepted until this bit is cleared. The ways to clear this bit are: either by handling an exception from a synchronous call, or by manually resetting it.

3.5 Summary

The conclusion we can draw after reviewing these projects are twofold. The first one is that there is a mutual exclusion between transparently providing a mechanism and keeping the bytecode intact. The second conclusion is that callbacks are not enough as a solution.

4 Explicit handling of the exception mask stack

First we provide an overview of the proposed approach, then it will be shown step by step how to use it. Its working will be detailed. The main aspect of the mechanism is to insert some wait-by-necessity in appropriate moments.

4.1 Overview

The goal is to provide an exception handling mechanism for both functional exceptions and non-functional ones. We also want to reuse the well known `try/catch/finally` construction since this is the only way in Java to manipulate (unwind) the call stack. Furthermore, nobody should be offended by an error handling code located in a `catch` block.

The mechanism is useful for asynchronously launching a method call even if its signature includes exceptions. A `try/catch` block is associated with this call, not necessarily the toplevel one at the moment of the call, but as an optimization we use the first one on the stack that can catch one of the declared exceptions. In order not to break the semantics associated with exceptions, we add two rules.

These rules are the main principle of the system:

- when accessing the future result, the potential exception is thrown,
- we do not leave the `try` block associated with the method call before the future returns.

The second rule implies that we may have to wait for some pending futures at the end of the `try` block. Actually, this is not enough, as we take great care to ensure that the exception is caught in its corresponding `catch` block. This must be specifically enforced by the mechanism, for example in some nested `try/catch` instances, as shown in Figure 3.

```
class ParentException extends Exception {}
class ChildException extends ParentException {}

try {
    A a = ro.foo(); // throws ChildException
    /* Here we must wait for 'a' */
    try {
        a.bar();
    } catch (ParentException pe) { ... }
} catch (ChildException ce) { ... }
```

Fig. 3. `tryWithCatch()` can be a wait-by-necessity point

If in that code, the `ro.foo()` call ends up with a `ChildException`, we don't want this exception to be thrown when the future is accessed through the `a.bar()` call. Doing so would wrongly let the exception be handled by the nested `catch (ParentException pe)` block instead of the rightful `catch (ChildException ce)` one. It's important to notice that the situation would be the same if it were `ParentException` that was a subclass of `ChildException` because in this case the exception thrown by `ro.foo()` could be a `ParentException`.

In Java, the target catch of a given exception is dynamically determined, so we must take care to prevent exceptions from being handled in unexpected `catch` blocks.

In order to be activated, the mechanism must be called in the code to protect at some key points, all of them shown in Figure 4.

```
public class RemoteObject {
    public DangerousThing dangerousMethod() throws DangerousException {}

    public static void main(String[] args) {
        ProActive.tryWithCatch(DangerousException.class); // Here (1)
        try {
            DangerousThing[] dt;
            RemoteObject[] ro;

            for (int i = 0; i < dt.length; i++)
                dt[i] = ro[i].dangerousMethod();

            ProActive.endTryWithCatch(); // Here (2)
        } catch (DangerousException de) { ... }
        finally {
            ProActive.removeTryWithCatch(); // And finally here (3)
        }
    }
}
```

Fig. 4. Complete example with the mechanism

The only goal of all those added method calls is to let ProActive know the state of the exception mask stack. Since this information mostly pertains to the client, it is implemented on the client side and very little is actually transmitted on the wire.

Note that those instructions are explicitly needed because of the lack of reification of the Java exception mask. The programmer and the middleware runtime have no access to it.

4.2 Before the Block

Before every `try` willing to put at work the mechanism, ProActive must be informed about which exceptions types will be caught in this block. This is done using the `tryWithCatch()` method taking as parameter an exception class or

an array of exception classes. These are the exceptions caught in the subsequent `catch` blocks.

As we reimplement a stack unwinding based exception system, this method pushes an exception handler on the stack. As an optimization, it also recomputes the current exception mask so that we know for each ProActive call if its exceptions will be handled by the mechanism or not.

4.3 During the Block

Here, we expect to asynchronously call methods throwing exceptions corresponding to the advertised exception types, otherwise the system would be quite pointless.

If an asynchronous method call ends with an exception, it is saved in the future in order to be thrown upon access, or when for some reason we have to throw the current exception.

If the method call was synchronous because of its return type (primitive type like `int` or a `final` class), the behavior is not changed from before the mechanism: the call is synchronous and the potential exception is thrown at its end.

In the one-way asynchronous case, the only moment where we could throw the exception is when the method `endTryWithCatch()` is called. This is suboptimal because the programmer is left with a subtle dilemma concerning one-way calls: either he makes big `try/catch` blocks so that the `endTryWithCatch()` comes late and a large window is left for calls in parallel before seeing the exception, or he wants his exception soon and he has to make small `try/catch` blocks.

The proposed solution to this problem consists in two Java methods used to throw an exception if one has arrived. We provide two methods because one is blocking and the other is not. The non-blocking one (`throwArrivedException()`) simply consults the current state and throws the potential exception, whereas the blocking one (`waitForPotentialException()`) is in some sort a barrier in that it waits for all asynchronous calls that could throw an exception in their return, and then throws the potential exception too.

4.4 At the End of the Block

The last instruction in the `try` block must be a call to `endTryWithCatch()`. The job of this method is to wait for the first of those events:

- every call in the current block that could throw an exception declared in the `tryWithCatch()` parameter has normally returned,
- an exception arrives.

This method also pops the current exception mask on the stack. If an exception is thrown before this call, it will be bypassed. We'll see in a moment how we handle this situation.

5 Implementation: Problems & Solutions

Trying to provide asynchronous exceptions in a way resembling to synchronous exceptions is not trouble free. We identify three main problems, more or less tied to our implementation in Java.

5.1 Source Code Modification

Java has the refinement of presenting exceptions as first class objects, thus, giving them all the features found in objects. Unfortunately, their handling (`try`, `catch`, `finally`) is totally impenetrable. That's why, the major part of the proposed mechanism is to reimplement an exception stack, side by side with Java's original.

In this situation, the main hazard to watch for is an inconsistency of the exception stacks with respect to the actual user stack. The ProActive view of the stack needs not be complete, only the subset containing asynchronous method calls is needed. Nevertheless, assuming the programmer takes care to annotate its exception stack manipulations with the aforementioned ProActive method calls, we can be out of sync as a result of a thrown exception unwinding the stack. Not only we cannot detect when an exception is thrown, but also we cannot figure out up to where did we unwind the stack.

Even when we ourselves throw an asynchronous exception, we cannot be sure at which level of the stack it will end up. This is because our view of the stack is incomplete.

Our solution to all these problems resides in asking the programmer to add a ProActive call in the `finally` block of every instrumented `try/catch` blocks. When an exception unwinds the stack, every `finally` block on the way upward will be executed, popping our stack at every step.

This problem has its origin in what is perceived as a Java shortcoming. Java does not provide methods to inspect the exception mask stack. It only provides the call stack thanks to a method, ironically, in the `Throwable` class (i.e. `getStackTrace()`). On the other hand, this shortcoming can be accepted if the goal is to hide the exception handling mechanism in order to optimize [16] it in the JIT.

Another source of desynchronization is a change in the exception handling code not propagated to the system, for example, a `catch` clause is suppressed, but the `tryWithCatch()` call continues to advertise it. We fix this problem by providing an automatic annotator. The annotations can be automatically computed given the source code, so we wrote a tool to do exactly that. This is not considered as a preprocessing but as a help to the programmer, that he is free not to use.

5.2 Throwing Exceptions

In the mechanism we rely on the possibility to launch any exception at any time, as shown by the `throwArrivedException()` method. On the other hand, we are in a Java environment, which enforces the `throws` declaration in method signatures to inform about the type of exceptions that can be thrown. This aspect has been a showstopper in many attempts at asynchronous exceptions, (as noticed by [9]).

The first point to consider in that problem is that the `throws` rules are enforced only by the compiler, so if we bypass the compiler checks, nobody else will prevent us from throwing exceptions.

The chosen approach to bypass the compiler is to dynamically create the bytecode at runtime with the help of libraries like ASM [17]. It permits to write

a method that throws any exception without checking the `throws` declaration, that we will happily leave empty. So we write our class with ASM and we load it thanks to the well known technique of calling `defineClass()` by reflection.

Now that we have our method that can throw any exception without the associated `throws` declaration, we have to call it. We cannot do it directly because as it is created dynamically it does not exist at compile time, and we would end with code that does not compile. Another solution would be to call the method using the reflection API, but the method in this API to call any method not only has a non-empty `throws` declaration but also wraps the actual exception in a `InvocationTargetException`. So the reflection way is useless there.

We had to fallback on another solution, this time using Java's dynamic binding. Let's say we have an interface containing a single method (used to throw an exception) with a, very importantly, empty `throws` declaration. The dynamically built class implements this interface, and the method actually throws the exception thanks to the lack of verification of the `throws` declaration. In the code we keep an instance of this interface, so the code compiles. Then we create an instance of the dynamically built class by reflection and we affect it to the aforementioned variable. When we want to throw an exception we simply have to call the method on this variable, it will compile since the method is in the interface, and thanks to Java's dynamic binding the dynamically built method is the one to be actually executed.

5.3 Consecutive Exceptions

Trying to implement asynchronous exceptions, side by side with Java exceptions can lead to strange situations. In the synchronous world of exceptions, when something bad happens an exception is thrown and the control flow leaves the offending stack frame, and of course the subsequent calls in this stack frame are skipped until the block with the matching `catch`.

With asynchrony, the whole point is to avoid waiting for the end of the call, so it is possible to receive two consecutive exceptions, potentially having the same cause (e.g. a network failure occured and several remote calls aborted).

The best we can do in this case is to report the first exception, and discard the subsequent exceptions. In the synchronous case, the second call would not have been launched, so its exception is useless. The problem is that we may end up in an incoherent state by launching calls even though the previous ones threw exceptions. We rely on the wait-by-necessity semantic between futures which produces a dependency graph in some sort to gain some correctness. Our solution is to assume that the dependency between values is the same as the one between exceptions, which seems natural.

Another considered approach is to throw the exception as soon as possible. That is to say, each time a ProActive method is called, if there is a pending exception we throw it. This is technically feasible, but can be problematic for the programmer since exceptions can then be thrown in unpredictable ways. So this approach is still left aside as experience is gained on the subject.

6 Conclusion

By turning asynchronous the method call mechanism, error handling becomes tricky. The most common solution is the usage of callbacks, but as we have seen it is not as powerful as it seems. Our approach is to trade a little bit of asynchrony in exchange of some control about where exceptions are caught. Of course this solution has a cost that needs to be evaluated. This cost is application dependent, and as before, the code needs some thinking to benefit the most from asynchrony. Typically, the result from a method call will be used in the same `try` block so as to avoid testing the result validity, this way it's not the end of the `try` block that will cause the wait-by-necessity but the return value usage.

References

1. Goodenough, J.B.: Exception handling: issues and a proposed notation. Commun. ACM **18** (1975) 683–696
2. Souchon, F., Dony, C., Urtado, C., Vauttier, S.: A proposition for exception handling in multi-agent systems. In: Proceedings of the 2nd international workshop on Software Engineering for Large-Scale Multi-Agent Systems, Portland, Oregon, USA (2003) 136–143
3. Romanovsky, A., Xu, J., Randell, B.: Exception handling and resolution in distributed object-oriented systems. In: ICDCS '96: Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96), Washington, DC, USA, IEEE Computer Society (1996) 545
4. Caromel, D., Klauser, W., Vayssière, J.: Towards seamless computing and metacomputing in Java. Concurrency: Practice and Experience **10** (1998) 1043–1061
5. Caromel, D., Genoud, A.: Non-functional exceptions for distributed and mobile objects. (In Romanovsky, A., Dony, C., Knudsen, J., Tripathi, A., eds.: Proceedings of the Exception Handling in Object-Oriented Systems workshop at ECOOP 2003)
6. Keen, A.W., Ge, T., Maris, J.T., Olsson, R.A.: Jr: Flexible distributed programming in an extended java. ACM Trans. Program. Lang. Syst. **26** (2004) 578–608
7. Olsen, R.A., Andrews, G.R., Coffin, M.H., Townsend, G.M.: Sr: A language for parallel and distributed programming. Technical Report TR 92-09, University of Arizona (1992)
8. Keen, A.W., Olsson, R.A.: Exception handling during asynchronous method invocation (research note). In: Euro-Par '02: Proceedings of the 8th International Euro-Par Conference on Parallel Processing, London, UK, (Springer-Verlag)
9. Raje, R.R., Williams, J.I., Boyles, M.: Asynchronous Remote Method Invocation (ARMI) mechanism for Java. j-CPE **9** (1997) 1207–1211
10. Logan, P.: Closures that work around final limitation (cunningham & cunningham wiki). (`http://c2.com/cgi/wiki?ClosuresThatWorkAroundFinalLimitation`)
11. Lea, D.: Concurrency utilities. (`http://www.jcp.org/en/jsr/detail?id=166`)
12. Vignéras, P.: Vers une programmation locale et distribuée unifiée au travers de l'utilisation de conteneurs actifs et de références asynchrones. PhD thesis, Université de Bordeaux 1, LaBRI (2004)
13. Vignéras, P.: Rami user's guide. (`http://mandala.sf.net/docs/rami.pdf`)
14. Pratikakis, P., Spacco, J., Hicks, M.: Transparent proxies for java futures. In: OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications, New York, NY, USA, ACM Press (2004) 206–223

15. Kurzyniec, D., Sunderam, V.S.: Semantic aspects of asynchronous RMI: The RMIX approach. In: Proc. of 6th International Workshop on Java for Parallel and Distributed Computing, in conjunction with IPDPS 2004, Santa Fe, New Mexico, USA, IEEE Computer Society (2004)
16. Suganuma, T., Yasue, T., Nakatani, T.: A region-based compilation technique for a java just-in-time compiler. In: PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation, New York, NY, USA, ACM Press (2003) 312–323
17. Bruneton, E., Lenglet, R., Coupaye, T.: Asm: a code manipulation tool to implement adaptable systems. Adaptable and extensible component systems (2002)

Testing and Understanding Error Recovery Code in Java Applications

Chen Fu and Barbara G. Ryder

Division of Computer and Information Sciences
Rutgers, the State University of New Jersey
110 Frelinghuysen Road, Piscataway, NJ 08854-8019, USA
{chenfu,ryder}@cs.rutgers.edu

Abstract. Server applications are expected to handle lower level faults and keep them from bringing down the whole system. Java provides a program-level exception handling mechanism in response to error conditions (that are translated into *exceptions* by Java VM). However, exception handling code is often widely scattered throughout an application and untested. This paper presents a program visualization tool *ExTest* that shows precisely all the handlers for exceptions triggered by certain kinds of operations, and for each of these handlers all the witness paths of how the operation would be triggered. Thus *ExTest* helps programmers understand the exception handling behavior of Java programs and also facilitates testing exception handling code.

1 Introduction

The Java programming language provides a program level exception handling mechanism in response to error conditions that happen during program execution. Subsystem faults (e.g. disk failure, network congestion) are translated into exceptions (e.g. `java.io.IOException`, `java.net.SocketException`) by the Java Virtual Machine [1]. Proper handling of these exceptions in program code is extremely important for reliability and fault-tolerance in server applications built in Java.

An exception handling mechanism helps separate exception handling code from code that implements functionalities during normal execution. However, exception handling code that deals with certain kinds of faults is still widely scattered over the whole program and mixed with other exception handling code, or even irrelevant code, making it hard to understand the behavior of the program under certain system fault conditions.

Moreover, exception handling blocks, especially those corresponding to system faults, are often left untested, for the reason that they can not be triggered by just tuning input data of the program. In our previous work [2], we proposed a white box testing metric for the exception handling behavior of the program. Supporting program analysis algorithms together with a testing framework using fault injection [3] were presented. Although very precise analysis is used in identifying exception def-uses, false positives can not be fully eliminated. It is a tedious and difficult job to identify the real false positives when they can not be exercised during the test.

In this paper we present *ExTest* – a program visualization tool built on top of Eclipse[4]. Based on program analysis introduced in [2], it groups together handlers

that handle exceptions triggered by a set of fault-sensitive operations[1]. Thus it facilitates navigation of the program code that relates to exceptions triggered by certain operations of interest. It also shows all program paths via which these operations can be reached from some call site in the `try` block, helping a user to understand the exception handling structure, and to identify spurious exception def-uses.

The rest of this paper is organized as following: In Sec. 2 we give a brief overview of exception def-use analysis that was introduced in [2]. Section 3 shows the structure and the functionality of *ExTest* in detail. In Sec. 4 we discuss the related work in the area of understanding and improving exception handling code in Java programs. Our future research work is discussed in Sec. 5.

2 Background

In [2], we proposed a def-use testing methodology for exception handling code, which is analogous to the *all-uses* metric of traditional def-use testing [5]. We repeat some of the key concepts here: In any given program execution, each fault-sensitive operation may produce an exception that reaches some subset of the program's `catch` blocks. We can treat fault-sensitive operations as the definition points of exceptions, and `catch` blocks as uses of exceptions. We define *exception-catch (e-c) links*:

Definition 1. (*e-c link*): *Given a set P of fault-sensitive operations that may produce exceptions, and a set C of* `catch` *blocks in a program, we say there is an e-c link(p, c) [2] between $p \in P$ and $c \in C$ if p could possibly trigger c; we say that a given e-c link is* **experienced** *in a set of test runs T, if p actually transfers control to c by throwing an exception during a test in T.*

Currently in the experiments conducted, we select P to contain all the native methods in JDK library that do network or disk I/O. In the rest of this paper we make this assumption unless explicitly stated otherwise[2].

Figure 1 shows the organization of our exception def-use testing system, which composed of two kinds of program analysis: static (compile time) and dynamic (run time) analysis. The static analysis calculates the possible *e-c links* for a program, which will be introduced shortly. The dynamic analysis monitors program execution, calles for fault injection to trigger an exception at an appropriate time, and records test coverage: The compiler uses the set of *e-c links* to identify where to place instrumentation that will communicate with the fault injection engine during execution. This communication will request the injection of a particular fault when execution reaches the `try-catch` block of an *e-c link*. The injected fault will cause an exception to be thrown upon execution of the fault-sensitive operation of the *e-c link*. The compiler also instruments the code to record the execution of the corresponding `catch` block. The tester runs the program and gathers the *observed e-c links* from each run. The testing goal is to drive the program into different part of the code so that fault injection can help exercise all the *e-c links* found in the program. Finally, the test harness calculates the overall coverage information for this test suite: the *observed e-c links* vs. the possible *e-c links*.

[1] Either a `throw` statement or a native method that may be affected by some fault – a hardware or OS failure – and produce some exception.

[2] The analysis and the testing framework are not dependent on the P selected.

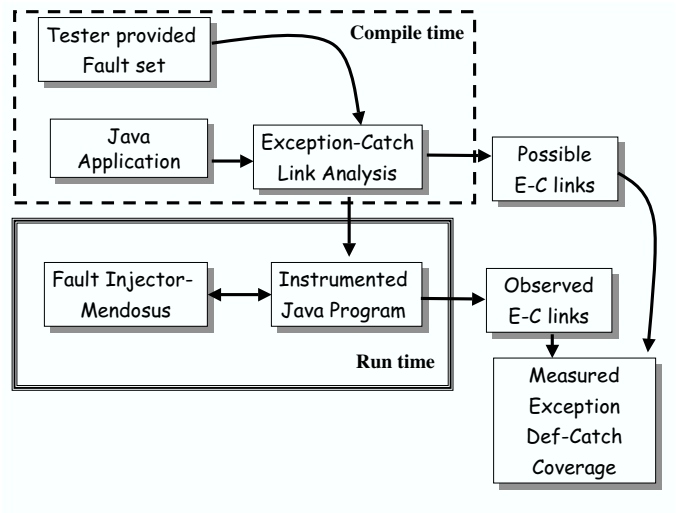


Fig. 1. Exception def-use Testing Framework

Next we introduce the static analysis that finds the possible *e-c links* in a Java program. Two algorithms are developed: ***Exception-flow*** and ***DataReach*** analysis [2].

Exception-flow is a dataflow analysis, similar to Reaching Definitions [6], but runs on call graphs instead of control flow graphs. Each $p \in P$ is propagated along the call edges in the reverse direction until some `try-catch` block c is met that encloses the call site and catches the exception thrown by p; thus an *e-c link* (p, c) is recorded. Moreover, during the propagation process, in each method (i.e. call graph node) to which p propagates, we can record the immediately *previous* method, thus indicating where p comes from. The exception propagation path can be collected *on demand* after the analysis finishes.

It is obvious that using a more precise analysis for call graph construction such as points-to analysis [7, 8] helps to reduce the number of infeasible *e-c links* found by exception-flow analysis. However, in practice even a very precise call graph building algorithm introduces many infeasible *e-c links*. Figure 2 is an example of typical uses of the Java I/O packages. Figure 3 illustrates how infeasible *e-c links* are introduced even given a fairly precise call graph. As we can see, the `catch` in `readFile` only handles exceptions result from disk failure and the `catch` in `readNet` only handles those triggered by network faults. But exception-flow information is merged in `BufferedInputStream.read1()` and propagated to both `readFile` and `readNet`.

We developed ***DataReach*** analysis to reduce the number of infeasible *e-c links* produced. The intuition was to use data reachability, usually obtained using points-to analysis, to confirm control-flow reachability. For example, continuing with Fig. 2, we can prove `SocketInputStream.read()` is **not** reachable from the call site `fsrc.read()` in method `readFile`, by showing that during the lifetime of the call `fsrc.read()`, no object of type `SocketInputStream` may be either loaded from any static/instance field of some class/object, or created by a `new` statement. In this way we can show that all the control flow paths associated with this *e-c link* are not

```
void readFile(String s){
  byte[] buffer = new byte[256];
  try{
    InputStream f =new FileInputStream(s);
    InputStream fsrc=new BufferedInputStream(f);
    for (...)
      c = fsrc.read(buffer);
  }catch (IOException e){ ...}
}
void readNet(Socket s){
  byte[] buffer = new byte[256];
  try{
    InputStream n =s.getInputStream();
    InputStream ssrc=new BufferedInputStream(n);
    for (...)
      c = ssrc.read(buffer);
  }catch (IOException e){ ...}
}
```

Fig. 2. Code Example for Java I/O Usage

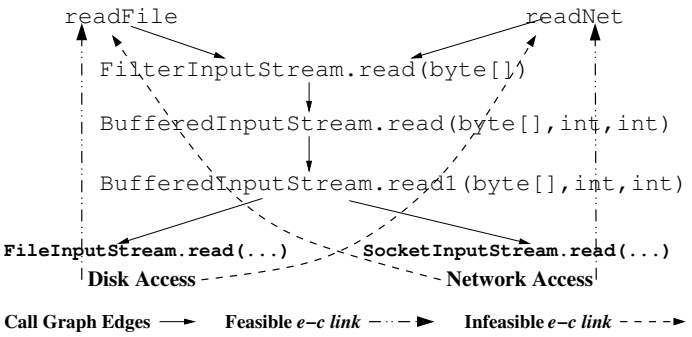


Fig. 3. Call Graph for Java I/O Usage

feasible, so that the infeasibility of the *e-c link* from `SocketInputStream.read()` to the `catch` block in `readFile` is proved. In general, *DataReach* tries to prove the infeasibility of each *e-c link* output by *Exception-flow* analysis, and only outputs those that it cannot prove to be infeasible. Our experiments showed that *DataReach* improved the precision of the system significantly; it reduced the number of possible *e-c links* by 41% on average in 6 benchmarks used in [2].

3 Visualization Tool — ExTest

In addition to being used by the exception def-use testing system, the information produced by these analysis, if carefully organized and visually displayed in an integrated development environment (IDE), can greatly facilitate both testing and program understanding of the exception handling code. We developed an Eclipse plug-in – *ExTest*, which invokes these analysis and organizes the output data into tree views to let users browse *e-c links* and trace exception propagation paths.

3.1 Motivation

When studying the exception handling behavior of Java programs, we found that, as mentioned in Sec. 1, exception handlers that deal with certain kinds of faults are often scattered in the program and mixed with handlers that take care of other kinds of error conditions.

```
void readString(String s){
 String buffer = s;
 try{
  InputStream n =new StringBufferInputStream(s);
  InputStream in=new BufferedInputStream(n);
  for (...)
   c = in.read(buffer);
 }catch (IOException e){ ...}
}
```

Fig. 4. Unreachable `catch` Block

For instance, a `catch` clause that handles an I/O exception may appear at each program point where some I/O channel is active. Each of these `catch` clauses may handle I/O exceptions triggered by different fault-sensitive operations (e.g. file write or socket creation), as shown in Fig. 2. Worse, some of these `catch` clauses never handle any I/O exception: Suppose in the program containing the code in Fig. 2, there is another method `readString`, shown in Fig. 4[3]. The `catch` block in this method handling I/O exception will **never** be triggered, because the code in the corresponding `try` block only reads from a string buffer in the memory – no actual I/O operation involved. Yet the `try-catch` structure is necessary for the program to compile.

If a programmer wants to learn this program's behavior under disk failure, she needs to find all the `catch` clauses that may handle exceptions that result from disk faults. Suppose a powerful lexical search tool with Java language knowledge as well as program specific information (e.g. type) is available. Then she can easily locate all the `catch` clauses that handle `IOException` or more general types of exceptions, but she still has to manually inspect at least all three `try-catch` blocks in both Fig. 2 and Fig. 4, and sometimes code reachable from them too, instead of just the one in method `readFile` that actually handles the exception result from disk failure. The problem becomes much more severe in non-toy Java server applications.

Using the analysis mentioned in Sec. 2, we can compute all the potential *e-c links* of the program. Each *e-c link* (p, c) tells us the fault-sensitive operation that triggers the exception and where it is handled. Thus, we can help solving the above problem by grouping *e-c links* according to their p value. Since the number of fault-sensitive operations that relate to disk I/O is small, one can just browse *e-c links* starting with

[3] This program – a single Java class containing the `main` method calling all of these three methods (also defined in this class) in Fig. 2 and 4 – is used as a running example in the following discussion about *ExTest*.

these operations to to get a good estimate of all the `try-catch` blocks that are related to disk I/O.

Ours is a program analysis which computes a safe approximation of program behavior. False positives are unavoidable, which means for some of the *e-c links* (p, c), the exception thrown at p never reaches c. It is up to the human programmer to decide whether an *e-c link* is actually spurious. This is especially important for exception def-use testing, because spurious *e-c links* can never be exercised during any test. Our program analysis provides exception propagation path data for all *e-c links*. Displaying these paths visually in Eclipse IDE helps to identify the spurious ones. In order for identifying spurious *e-c links*, all propagation paths for a given *e-c link* must be shown.

3.2 Tool Structure

Our program analysis are implemented as modules in the Soot Java Analysis and Transformation Framework [8] version 2.0.1. Upon user request, *ExTest* (i) calculates environments of the current working program (e.g. class paths) using Eclipse services, (ii) starts another process running Soot with our modules enabled, and (iii) reads the output data of the Soot modules after the process finishes.

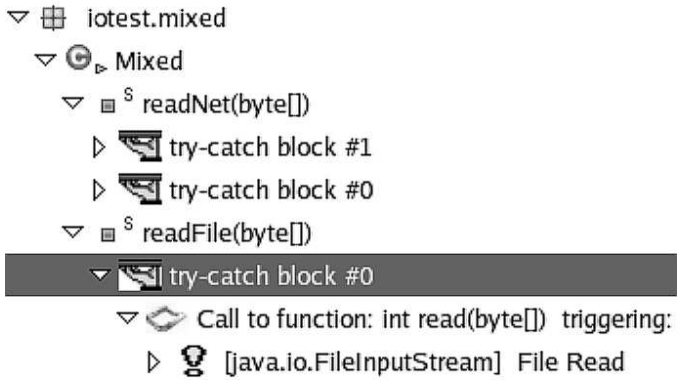
In the Eclipse IDE, we want the users to be able to explore the *e-c links* (e.g. browsing all the `catch` clauses and their relationships with the fault-sensitive operations) as well as the witness paths that demonstrate the feasibility of an *e-c link*. The data generated by the Soot modules are organized in an XML file, which contains all the *e-c links* found in the given program and information about the paths – needed by *ExTest* to perform the intended functionality.

3.3 Browsing *e-c links*

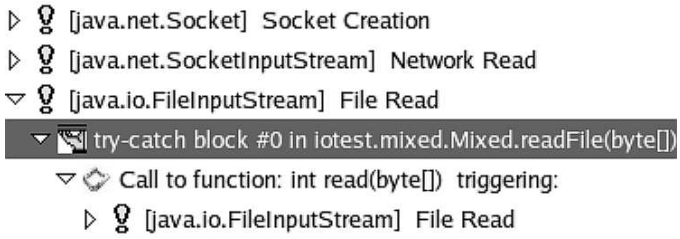
Each record of an *e-c link* (p, c) in the output data of our Soot modules contains the following information: the ID of p, the position of c in source code and the call site(s) in the corresponding `try` block which may lead to the execution of p. These *e-c links* can be grouped in two ways: by p or by c. We implemented both of them by means of two different views in Eclipse: the *Handlers* view and the *Triggers* view.

Figure 5(a) shows the *Handlers* view: a tree-view in Eclipse where *e-c links* are grouped by the `try-catch` blocks. These `try-catch` blocks are further grouped by their definition positions: the methods, classes, packages in which they are defined. Each `try-catch` block can be expanded to show all the fault-sensitive operations that may trigger exceptions reaching the `catch`. The last `try-catch` block in the figure is highlighted and expanded. It is defined in package `iotest.mixed`, class `Mixed` and method `readFile`. We can see that one method call in the `try` block reaches a fault-sensitive operation in the JDK: "File Read".

Figure 5(b) shows the *Triggers* view, where the *e-c links* are grouped by the fault-sensitive operations. By expanding the "File Read" operation we can see that only one `try-catch` block in the program handles an exception thrown by read of a file. So if a user is interested in program behavior under a disk fault, she can just concentrate on this one `catch` block.



(a) *Handlers* View



(b) *Triggers* View

Fig. 5. Tree Views of *e-c links*

Thanks to the environment provided by Eclipse IDE, these two tree views can be interactively explored. The `try-catch` block, the statements in the `try` block that may lead to the fault-sensitive operation, etc., can be opened and highlighted in the Java source file editors, upon double click on the corresponding items in the view. For example, in both views, we can see the actual code for the `try-catch` block #0 by double clicking on the line.

3.4 Displaying All Paths for an *e-c link*

We also want to display the paths that show how p in an *e-c link* (p, c) can be reached from the `try` block that corresponds to c. Selecting and displaying only one (the shortest) path for each *e-c link* is not enough, especially with the presence of the false positives. In order for a programmer to decide that an *e-c link* is spurious, she has to make sure that **all** the control flow paths from the corresponding `try` to p are actually infeasible. So it is necessary for *ExTest* to display **all** these paths to be practically useful. But the total number of paths may be exponential to the size of the program [9]! Clearly, the approach of gathering and dumping all these paths into an output file after the analysis finishes will not scale.

As mentioned in Sec. 2, *Exception-flow* analysis records the propagation paths of each $p \in P$ by annotating nodes on the call graph. These annotations can be trivially changed into annotations on call edges. Since the set of fault-sensitive operations P is pre-selected according to the kind of faults that are of interest to the user (not depending

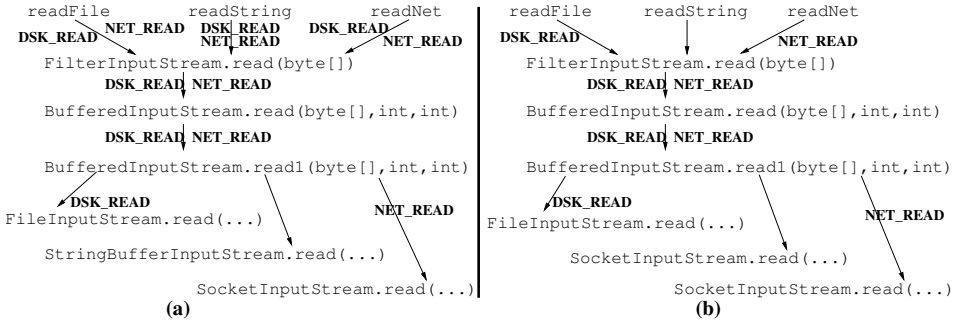


Fig. 6. Annotated Call Graph

on the program being analyzed), the size of the annotated call graph is at most linear in the size of the original call graph. Figure 6(a) shows the annotated call graph for the example in Figs. 2 & 4. Edges of the call graph are annotated with IDs of the fault-sensitive operations according to the result of *Exception-flow* analysis.

The problem with this approach is that the results of *DataReach* are ignored. As stated before, *Exception-flow* analysis alone would leave too many false positives in the graph; with this data, the user must manually explore many unnecessary call edges to decide that a certain *e-c link* is infeasible. So we need to take the advantage of *DataReach* to reduce this workload.

Recall that *DataReach* proves that some of the *e-c links* are infeasible by showing the infeasibility of the all the control flow paths of these *e-c links*. To be able to incorporate its result into the annotated call graph, we modified *DataReach* so that for each *e-c link* (p, c), the annotations of p on all the call edges associated with (p, c) are *confirmed* only if we *cannot* prove the infeasibility of (p, c) (i.e. (p, c) survives *DataReach* test). During the output of the call graph, only the *confirmed* annotations are written with the graph, thus many spurious annotations can be removed. Figure 6(b) shows the annotated call graph with the unconfirmed annotations removed.

With the annotated call graph, the paths can be generated on demand in *ExTest*. Suppose one user chooses to trace the paths of some *e-c link* (p, c). *ExTest* can retrieve from the graph all the outgoing edges departing from the `try` block that are annotated with p, and the target methods can be displayed to the user. Then the user can choose to trace one of these methods, *ExTest* can retrieve all the outgoing edges from that method that are annotated with p and display the target methods of these edges. This process can be repeated until the fault-sensitive operation p itself is reached.

Fig. 7. Exception Propagation Path

Figure 7 is the expanded view of the last *e-c link* shown in Fig. 5(b). Only one witness path was discovered by the analysis, which precisely reflects the analysis result shown in Fig. 6.

However, we are not always so lucky in bigger programs; paths in these programs can get very complicated, especially inside the JDK library classes that make heavy use of polymorphism. Figure 8 shows part of the *Triggers* view displayed when browsing *e-c links* in one of the testing benchmarks used in [2] – a FTP server written in Java [10]. Witness paths of one *e-c link* are partly expanded in the figure, with the fault-sensitive operation `SocketInputStream.read()` highlighted.

As can be seen from the figure, the "fan out" of some of the nodes along the paths is large (e.g., `InputStreamReader.close()`). Furthermore, many of the methods appear more than once, which indicates the possibility of recursion introducing a path with unbounded length. Since these paths are extracted out of a call graph, expanding the second appearance of a method on a path brings exactly the same set of children in the tree view. This is wasteful and introduces unnecessary complexity into the view. Manually identifying the recursion in a complex view like this is not trivial. Therefore we have automated recursion detection in *ExTest*. As shown in Fig. 8, many methods are annotated with "..." and they are **not** expandable, which shows that the method has been called recursively and further expansion is not necessary.

If we only show only one witness path of the *e-c link*– the natural selection would be to show the shortest one – the view can be greatly simplified, but the real complexity of the problem would be hidden from the user. With only one path shown, the user is not helped in identifying infeasible paths; however, expanding and highlighting the shortest path automatically among all paths may help in understanding the overall program structure quickly. We are now working on implementing this feature in *ExTest*.

4 Related Work

This paper presents a tool to help understand and maintain the exception handling feature of Java programs, based on *exception-catch link* analysis. There is much previous research work in both exception handling analysis and its usage in various software engineering tools. Here we will discuss only the works that are most closely related to the tool discussed in this paper[4].

There are tools built to improve exception handling in programs, for example avoiding exception handling through subsumption, or finding unhandled exceptions for a given method. Jo et. al [11] presented an interprocedural set-based [12] exception-flow analysis for checked exceptions. A tool [13] was built based on this analysis which shows, for a selected method, uncaught exceptions and their propagation paths. It is unclear from the paper whether a certain path for each exception is selected and displayed, or if all of the paths are displayed. Experiments show that this is more accurate than an intraprocedural JDK-style analysis on a set of benchmarks five of which contain more than 1000 methods. Robillard et. al [14] described a dataflow analysis that propagates both checked and unchecked exception types interprocedurally. Their tool *Jex* illustrate

[4] More discussion about research results in fault injection, exception analysis and optimizations, infeasible paths detection and program testing can be found in the related work section of [2].

Fig. 8. Exception Propagation Path

exception handling structure in application code. It analyzes exception control flow and thus identifies exception subsumption. These analysis mentioned above are less precise than ours. Their call graph is constructed using class hierarchy analysis, which yields very imprecise results [15, 16]. Even if a fairly precise call graph[5] were provided for their analysis, the precision of the results would be resemble those of our *Exception-flow* analysis using the same call graph. So they are not capable of identifying that the `catch` clause in Fig. 4 can never be triggered.

Reimer and Srinivasan [17] introduced *SABER*, part of which targets at a wide range of exception usage issues in order to improve exception handling code in large J2EE applications and ultimately to increase robustness of the program. These issues include swallowed exceptions, single `catch` for multiple exceptions, a handler too far away from the source of the exception and costly handlers. Warnings are given to the programmer upon recognizing one of these problems. Unfortunately, analysis used to identify these potentially problematic code areas are not introduced. Their work, if combined with our analysis, might prune some of the warning messages to reduce the workload of user, and produce more valuable information at the same time. For instance, suppose the `catch` clause in Fig. 4 is empty, since our analysis shows that it can never be triggered, warnings relative to this `catch` clause can be suppressed. Moreover, when facing a `try` block with many call sites that potentially throw the same type of exception, our analysis can show whether these exceptions are of the same origin and with similar propagation paths, which in turn helps deciding whether the `try` block needs to be split.

5 Conclusions and Future Work

We present a program visualization tool that facilitates navigating code related to the exception handling feature of Java programs, based on the program analysis used in exception def-use analysis [2]. We want to reveal all information needed to the user, while carefully organizing the data to help human browsing and reasoning.

Despite of our current efforts, Fig. 8 shows us that exploring program code based on conservative static program analysis results can be difficult. One way to alleviate the situation is to use more precise analysis (but possibly more expensive) to reduce the size of the result data.

The current implementation of *ExTest* only displays the results of the static analysis in [2]. Our next step is to display visually both the static and dynamic analysis results, because we believe that it will give user more intuition and hints when browsing. We are working on combining two kinds of dynamic information into the current views: testing coverage data of *e-c links* and dynamic call graphs/trees. Once we incorporate these data, we can highlight *e-c links* not covered during the test in the *e-c link* views (shown in Fig. 5) to draw the user's attention to these untested parts of the program. Furthermore, when a user chooses to explore the paths of a certain *e-c link*, with the dynamic call graph/tree, we can graphically show which of the edges are actually executed during the test. On this view, the users might want to concentrate on the "fringe"

[5] For example, a call graph constructed using *Spark*, a field sensitive context insensitive points-to analysis module in Soot.

of the already executed edges, to see why the program is taking the wrong "branch". Thus *Extest* can help the user either to compose another test case or decide that edge is actually infeasible.

References

1. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification. Second Edition. Addison Wesley (1999)
2. Fu, C., Milanova, A., Ryder, B.G., Wonnacott, D.G.: Robustness Testing of Java Server Applications. IEEE Transactions on Software Engineering **31** (2005)
3. Nagaraja, K., Li, X., Zhang, B., Bianchini, R., Martin, R.P., Nguyen, T.D.: Using fault injection to evaluate the performability of cluster-based services. In: Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS 2003), Seattle, WA (2003)
4. `http://www.Eclipse.org`: The Eclipse IDE (2005)
5. Rapps, S., Weyuker, E.: Selecting software test data using data flow information. IEEE Transactions on Software Engineering **SE-11** (1985) 367–375
6. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers – Principles, Techniques and Tools. Addison Wesley (1988)
7. Rountev, A., Milanova, A., Ryder, B.G.: Points-to analysis for java using annotated constraints. In: Proceedings of the Conference on Object-oriented Programming, Languages, Systems and Applications. (2001) 43–55
8. Lhoták, O., Hendren, L.: Scaling Java points-to analysis using Spark. In Hedin, G., ed.: Compiler Construction, 12th International Conference. Volume 2622 of LNCS., Warsaw, Poland, Springer (2003) 153–169
9. Ball, T., Larus, J.R.: Efficient path profiling. In: MICRO 29: Proceedings of the 29th annual ACM/IEEE international symposium on Microarchitecture, Washington, DC, USA, IEEE Computer Society (1996) 46–57
10. Sortokin, P.: Ftp server in java (2003)
11. Jo, J.W., Chang, B.M., Yi, K., Cho, K.M.: An uncaught exception analysis for Java. Journal of Systems and Software (2004) in press.
12. Heintze, N.: Set-based analysis of ml programs. In: Proceedings of the ACM Conference on Lisp and Functional Programmig. (1994) 306–317
13. Chang, B.M., Jo, J.W., Her, S.H.: Visualization of exception propagation for java using static analysis. In: SCAM'02: Proceedings of the Second IEEE International Workshop on Source Code Analysis and Manipulation. (2002)
14. Robillard, M.P., Murphy, G.C.: Static analysis to support the evolution of exception structure in object-oriented systems. ACM Transactions on Software Engineering and Methodology (TOSEM) **12** (2003) 191–221
15. Dean, J., Grove, D., Chambers, C.: Optimization of object-oriented programs using static class hierarchy. In: Proceedings of 9th European Conference on Object-oriented Programming (ECOOP'95). (1995) 77–101
16. Bacon, D., Sweeney, P.: Fast static analysis of C++ virtual functions calls. In: Proceedings of ACM SIGPLAN Conference on Object-oriented Programing Systems, Languages and Applications (OOPSLA'96). (1996) 324–341
17. Reimer, D., Srinivasan, H.: Analyzing exception usage in large java applications. In: EHOOS'03: ECOOP2003 - Workshop on Exception Handling in Object Oriented Systems. (2003)

Conversations for Ambient Intelligence

Stijn Mostinckx*, Jessie Dedecker**, Tom Van Cutsem**, and
Wolfgang De Meuter
{smostinc, jededeck, tvcutsem, wdmeuter}@vub.ac.be

Programming Technology Lab
Department of Computer Science
Vrije Universiteit Brussel

Abstract. The development of programs for mobile devices inhabiting Ambient Intelligent network constellations is notoriously difficult. These difficulties stem from the fact that the limitations imposed by the hardware need to be dealt with in an ad hoc fashion. In this paper we advocate the use of *conversations* as a general programming model to develop applications for Ambient Intelligence. We illustrate how conversations tackle the different concerns of ambient-oriented software development.

1 Introduction

The past few years Ambient Intelligence (AmI) has begun to seep into society. Whereas the vision as a whole remains futuristic, the introduction of ever cheaper, smaller and more powerful mobile devices – such as cellular phones and PDAs – cannot go unnoticed. These devices also harbour the necessary wireless network provisions that allow them to escape their isolation and collaborate in open, highly dynamic network settings. Whereas technically feasible, collaboration between different devices remains cumbersome due to the sheer complexity of the software that governs such collaborations. One particularly challenging problem consists of finding abstraction mechanisms for the collaboration between different mobile devices, with respect to the different hardware limitations overshadowing the interaction.

This paper advocates the use of *conversations* [Ran75] – a well-known exception handling technique – as a programming model for developing AmI applications. Upon conducting a careful examination of the hardware limitations imposed by mobile devices, we identify the main concerns to be addressed in *ambient-oriented* software. Subsequently we investigate how conversations can be employed to address these concerns and which extensions and modifications are needed to obtain a full-fledged programming model for ambient-oriented applications.

* Funded by a doctoral scholarship of the Institute for the Promotion of Innovation through Science and Technology in Flanders (IWT-Vlaanderen), Belgium.
** Research Assistant of the Fund for Scientific Research Flanders, Belgium (F.W.O.)

2 Motivation

This section will highlight the main differences between traditional distributed systems and the AmI setting of collaborating devices investigated in this paper. These differences will provide us with a means to evaluate the applicability of conversations in an AmI environment. At present, mobile devices are often characterised by scarce resources such as lower CPU speed, available memory and battery lifespan. However, given the rapid evolution of these devices and their growing resemblance to full-blown computers such as laptops, we consider these issues not to be fundamental characteristics of AmI devices.

The characteristics that we *do* consider to be fundamental are directly related to the peculiarities of the wireless networks that allow the mobile devices to connect with one another. These wireless networks come in two distinct flavours: *nomadic* and *ad hoc* networks. The former network type implies that mobile devices roam while remaining connected through the use of dependable infrastructure. Ad hoc networks on the other hand can be characterised by the absolute lack of central infrastructure to support the interaction. To both network constellations the following observations can be applied:

Volatile Connections A first important difference between traditional distributed systems and an AmI setting of collaborating devices is that the latter can no longer rely on stable network connections. For ad hoc networks with no infrastructure available, connectivity is limited by the range of the wireless facilities. Thus connections may be broken as users move about. When infrastructure is available, roaming users in a nomadic network may still choose to use network facilities periodically, for example to minimise the cost[1] or the battery consumption of upholding a network link.

Ambient Resources In contrast to their counterparts in stationary networks, ambient-oriented applications should not rely on explicit knowledge of the available resources. Instead the availability of resources needs to be discovered dynamically, as the open network dynamically evolves due to the unheralded joining and leaving of devices (which provide specific services).

No Presumed Infrastructure Whereas servers – reliable nodes providing a fixed set of services for their clients – are commonplace in traditional distributed systems, ambient-oriented applications should be able to function without them. Of course, one cannot prohibit software developers to make use of servers in their applications, but the underlying network layer of a programming model for ambient-oriented software should rely only on peer-to-peer networks to accommodate ad hoc network constellations.

Natural Concurrency The need for concurrency naturally arises in a setting populated by mobile devices. It is inconceivable to consider applications that use the dynamics of a network of these devices as a single-threaded application. If this were the case, the disconnection of whatever device that currently holds the running thread would freeze an entire network of devices.

[1] Typically, access to reliable network infrastructure – such as a GPRS-network for SmartPhones – requires payment based on the time one remains connected.

We have explored these hardware characteristics in previous work, to isolate some characteristic features of an ambient-oriented programming language [DVM+05]. A result of this experiment was the development of AmbientTalk[2], a minimal, yet realistic[3], programming language kernel for developing ambient-oriented programs. AmbientTalk introduces actors [Agh86] to encode processes which communicate with one another using asynchronous messages. Such asynchronous communication limits the effects of failing communication links. AmbientTalk additionally provides a basic service discovery mechanism that is based unification of patterns published by the providers and potential users of a service. A thorough introduction of the language is clearly outside the scope of this position paper and can be found in the aforementioned article [DVM+05]. The Ambient Actor model, which is the formal basis for AmbientTalk is detailed in [DV04].

While developing some examples in AmbientTalk, we have come across four concerns that programmers encounter during the development of an ambient-oriented application. The first two concerns are directly related to the exchange of messages between different parties, whereas the latter two are related to the particular network constellations under consideration.

Synchronisation In response to the hardware phenomena described above, ambient-oriented languages require the use of asynchronous, non-blocking communication primitives. However these primitives place a cognitive burden on the programmer, who has to manually encode the synchronisation points in his application using call-backs. This style of programming is akin to continuation-passing style, which is traditionally considered cumbersome to program in. Therefore an important aspect of an ambient-oriented model is the ease with which one can express synchronisation points in the software.

Exception Handling Present-day applications typically use exceptions extensively as a means to signal exceptional events. The need for exception handling becomes obvious in distributed systems, where failing network connections are signalled through the use of exceptions as well. Given the dynamic networks we are investigating, such network exceptions will occur frequently. The well-known try-catch block cannot be aligned with asynchronous message passing, since exceptions may be signalled to the calling device, long after the try-catch block was exited. Because ambient-oriented programming languages are *obliged* to use asynchronous communication, new means for exception handling need to be explored.

Decentralised Distribution Since one can in principle not rely on the availability of any infrastructure whatsoever, it is important that none of the protocols relies on the use of a reliable server infrastructure. In particular the protocols that facilitate the synchronisation and exception handling issues detailed above, should avoid being dependant on a designated leader. Such decentralisation avoids upheaval when the leader becomes unreachable.

[2] More information on the language as well as access to the experimental virtual machine is available at: `http://prog.vub.ac.be/~jededeck/research/ambienttalk/`

[3] The AmbientTalk virtual machine is developed in pure Java and is currently deployed on QTek 9090 SmartPhones.

Service Discovery Since ambient-oriented applications inhabit inherently dynamic open networks, one cannot encode general explicit knowledge that describes where an object may encounter available services. Similarly, the requirement for decentralised distribution also prohibits the use of a name server-based architecture. Thus, when developing an ambient-oriented program, the ability to "sense" services in one's environment is crucial.

The concerns mentioned above will be recapitulated in section 4 to analyse the suitability of the conversation model as a programming model for ambient-oriented software. First, the next section provides a basic introduction to the original conversation model and its more recent offspring the Coordinated Atomic Action model.

3 The Art of Conversation

Conversations were introduced by Randell as an abstraction to control concurrency and communication between collaborating processes [Ran75]. He observed that dependencies are created between processes as they exchange information with one another. Consequently, the effect of a software failure in a single process easily spreads over all dependent processes, since to restart the faulty process, all dependent processes need to be restarted as well. To alleviate this problem, conversations isolate a group of processes, as illustrated in figure 1.

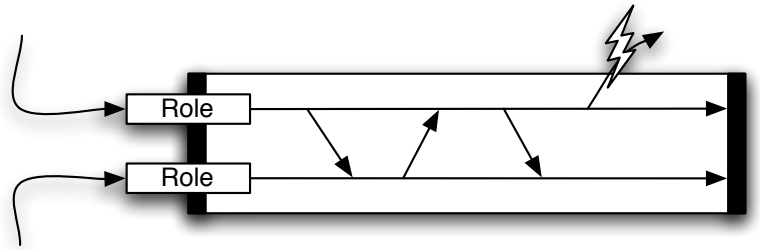


Fig. 1. A traditional conversation isolates its participants from external processes

Figure 1 shows that processes may become *participants* of a conversation by triggering a *role* of the conversation. This role prescribes the behaviour exhibited by the process while it is confined in the conversation's borders. Upon triggering a role – and crossing the left-most barrier or recovery line – the conversation creates a checkpoint of the participant's state. Throughout the conversation, participants may freely communicate with one another, but not with external processes. Such *information smuggling* is prevented to avoid the creation of dependent processes outside of the control of the conversation. Finally, the right-most barrier or test line is used to detect faulty participants. If needed, the conversation restores the checkpoints and restarts itself.

3.1 Coordinated Actomic Actions

Coordinated atomic actions (CA actions) are an object-oriented extension of the traditional conversations described above [XRR+95]. Whereas a thorough discussion of all aspects of the CA action model will lead us too far, we do highlight some of the additional concepts the model introduces.

Forward Error Recovery Whereas traditional conversations respond to errors uniformly by retrying the conversation, the CA action model allows the programmer to specify handlers for exceptions raised by its participants. Upon successfully completing a handler, the CA action terminates normally. Since exceptions may be raised concurrently by different participants, a CA action introduces the notion of a *resolution graph* which maps combinations of exceptions to a corresponding handler. Gathering the different raised exceptions and determining the handler to be triggered by the participants is performed at run-time by the *action manager*.

External Objects Participants of CA actions *may* communicate with external objects, provided that these objects can guarantee transactional semantics. In particular, such external objects should prevent *information smuggling*, by ensuring that the effects of the conversation can be rolled back if necessary. In other words, external objects *"must be atomic and individually responsible for their own integrity"* [VG00].

4 Conversations for Ambient Intelligence

The CAA model described in the previous section has already been applied to a variety of real-world problems [XRR+99,BRR+00,VGR00,ZPR03]. These examples clearly illustrate that conversations are a powerful means of abstraction to structure distributed systems. This section will evaluate conversations with respect to the four concerns of ambient-oriented programming we have identified in section 2. To make this analysis more concrete we outline a minimal AmI scenario to illustrate the different concerns.

> Alan pulls up in front of the restaurant where he has booked a table. As Alan switches off the engine – by retracting his eKey from his dashboard computer to his PDA – a valet approaches offering to park Alan's car. Through his PDA, the valet offers a certificate, which is both a proof that he is employed by the restaurant and a ticket to reclaim the car after the meal. When Alan's PDA validates the certificate, it hands over the car's eKey to the valet.

This scenario, albeit minimal, bears many of the essential characteristics of ambient-oriented software. First of all, the scenario features mobile devices, which connect with one another in an ad hoc fashion. Second, the car parking service offered by the valet is an ambient resource – it becomes available when

Alan pulls up outside the restaurant – which needs to be discovered. Third, the scenario is also interesting since the protocol of interaction between the PDAs of Alan and the valet can be automated. In the remainder of this section, we will illustrate how conversations can be adapted to make this typical AmI scenario possible. Finally, we present some pseudo-code for the resulting conversation.

4.1 Synchronisation

The car parking scenario has two points where the two processes need to synchronise and exchange information, respectively the certificate and the eKey. Conversations provide an apt mechanism to synchronise collaborating processes, since the conversation typically only starts when all roles are attributed to participants. Semantically, this is a very useful property, especially if a participant may supply data, which can be used inside the conversation by all participants, such as the certificate supplied by the valet role[4]:

```
role valet(certificate) {
    // Role body
}
```

Whereas semantically, synchronisation at the start of a conversation is a useful feature, one must take into consideration that technically the actual start of a conversation is determined by the availability of resources. Since an AmI setting is characterised by its dynamic open network settings in which the availability of resources cannot be predicted, the conversation may start an indefinite amount of time after the process has signalled it wants to participate. Since concurrency is a natural phenomenon of our setting, we propose to ensure that the process will only be confined to the boundaries of the conversation once it effectively starts executing its role. As such, the process may still answer asynchronous requests while the other participants are not available yet.

4.2 Exception Handling

The car parking scenario exemplifies the use of exceptions to report on unexpected events, *i.e.* someone may pose as a valet, and present a false certificate to steal Alan's car. Conversations were explicitly designed to handle exceptions in distributed systems, so it should not be surprising that conversations perform this task adequately for ambient-oriented programs as well.

Unlike when using try-catch, it is impossible that an exception is *signalled* when the context in which it was to be handled is already abandoned. This behaviour is due to the fact that processes may not leave a conversation until all participants have completed their role. This logical synchronisation at the end of a conversation can be upheld equally well in an AmI setting. However, since in ambient-oriented software network connections are inherently volatile, blocking participants at the end of a conversation renders the conversations

[4] Other role bodies may access the certificate using valet.certificate

fragile. Therefore, non-functional exceptions [CG03]– *e.g.* signalling that another participant is no longer reachable – should not be handled automatically by the virtual machine, but rather be propagated to the conversation which can then incorporate them in its resolution.

4.3 Decentralised Distribution

Since conversations have been widely used to tackle exception handling in distributed systems, different possibilities to distribute a conversation have already been explored [RZ97]. Whereas *Romanovsky and Zorzo* suggest that both the roles and the action manager of a conversation can be distributed, the solutions discussed in their paper only support distribution of roles. The COALA framework implements distribution of action managers, but still enforces the use of a *leader manager* for coordination. Such a higher authority cannot be reconciled with the AmI criterion that imposes *no presumed infrastructure*. Currently we are exploring different possibilities to relax the reliance on a single leader manager to alert all participants of raised exceptions.

Apart from finding a reliable peer-to-peer protocol to notify all parties of failures, we are also concerned with how the distribution of a conversation-based ambient-oriented program is achieved. We envision the development of such applications as follows: All relevant entities in the program are defined as (ambient) actors. Subsequently, conversations are used to express the collaborations that may occur between these exemplar actors. In the car parking scenario, the role of a valet could thus be specified as follows:

```
role valet(certificate)@CPEmployeeActor {
    // Role body
}
```

The @ syntax shown in the above example confines the participants who wish to take on the role of a valet to the CPEmployeeActor, an exemplar actor representing an employee of the car parking service. The exemplar actor is then cloned to instantiate multiple employees, which can all perform the task of a valet. These clones can then be distributed to the PDA of each employee, which is able to trigger the conversation, if it meets a customer.

In general this mechanism implies that after specifying the different conversations which the exemplar actors may participate in, copies of these exemplar actors can be distributed to all relevant devices. When distributing these cloned actors, one should not be aware of the possible conversations these actors may engage in. The distribution model that we hint at in this section holds the promise of being able to substantiate this vision, without requiring centralised support.

This seemingly oblivious way to obtain distributed conversations, is achieved by introducing *role slots* to represent the roles a conversation attributes to the exemplar actors[5]. These role slots are then used to embody the relation between

[5] The use of role slots is inspired by Slate which introduces them to inform objects of their roles in multimethod invocations [RS05].

actors and the conversations they can participate in. Copies of the exemplar actors contain copies of these role slots to ensure that they can engage in precisely the same conversations as their original.

When distributing ambient actors, the marshalling algorithm will use the role slots to ensure that all conversations in which the actor can participate are made available at the destination node. As such the marshalling algorithm of the language is used to ensure that both the roles and action managers of a conversation are distributed across all nodes that may host participants.

4.4 Service Discovery

Conversations are traditionally not regarded as a means to perform service discovery, yet in the scenario two people meet, exactly because they may interact in a meaningful way. Generalising from this example, we identify conversations as a means to perform *participant discovery*. This implies that processes signal their interest in performing a particular task by participating in a conversation. The conversation is thus conceived as a go-between which brings the process in contact with other potential participants.

Due to the distribution model outlined above, actors are co-located at all times with the different conversations they can participate in. Consequently, participating processes are handled locally without using central infrastructure. Nevertheless, the system should be aware of processes wanting to collaborate. The service discovery system sketched below can be used to fulfil this role.

At the level of the AmbientTalk virtual machine, every device periodically broadcasts its presence, allowing other devices to discover its availability. The service discovery algorithm we propose, reacts to such a notification of presence by transmitting the device's list of active conversations – conversations with at least one role filled in. When receiving such a list of active conversations, the service discovery algorithm will attempt to unify the provided conversations with the ones it hosts on it's own device. Such a unification may produce three different results :

- Match : The unification succeeded and all the conversation's roles have been assigned to participants. Upon encountering a match, the conversation is started.
- Underspecified : The unification succeeded, yet some roles are not filled in. The service discovery algorithm will query all reachable devices, to check whether they can contribute in the conversation. This is necessary to allow for conversations spanning more that two devices. If the conversation, cannot be completed, no matching occurs. No intermediate matching is performed to avoid problems should the partner become unreachable before the remaining roles are filled in.
- Overspecified : The unification failed, for example because both devices filled in the same role. Neither conversation can be started.

Concretely in the scenario, both Alan and the valet will fulfil one role in the conversation: respectively customer and valet. When Alan then pulls up in front

of the restaurant, the ranges of both PDAs intersect, triggering the service discovery algorithm. This unification will indicate a MATCH, and thus initiate the interaction between the devices.

4.5 Scenario revisited

We have analysed how conversations could be conceived as a programming model for ambient-oriented software. Conversations currently do not address all the concerns we have identified as crucial to ambient-oriented software development. However, the ideas we have presented to tackle these problems only involve the support for conversations provided by the underlying virtual machine. Given that the proposed changes are largely invisible to the programmer, we conclude that *conversations provide a potent programming model for ambient-oriented applications.*

In conclusion of this section, we return to the scenario that has guided our analysis. The entire interaction between Alan and the valet may be captured in a single conversation, which is outlined below. The CarParkService conversation essentially establishes the first synchronisation point between the valet and the customer roles.

```
1  conversation CarParkService {
2
3      exception InvalidCertificate();
4
5      method validate(certificate) throws InvalidCertificate{
6          // check the certificate
7      }
8
9      role valet(certificate)@CPEmployeeActor {
10         Parking.valet(certificate);
11     }
12
13     role customer()@PersonActor {
14         validate(valet.certificate);
15         Parking.customer(self.car, self.carkeys)
16     }
```

As can be witnessed the valet at this point makes a certificate available to all participants of the conversation (line 9). While executing its role, the customer will attempt to validate the provided certificate (line 14). Provided that this validation does not throw an exception, another synchronisation point is established between both processes when they join the nested Parking conversation.

```
17     conversation Parking {
18
19         role valet(certificate)@CPEmployeeActor {
20             self.park(customer.car, customer.eKey);
21             self.store("CarRetrieveService", certificate, customer.eKey);
```

```
22              }
23
24              role customer(car, eKey)@PersonActor {
25                  self.store("CarRetrieveService", valet.certificate, eKey);
26              }
27
28          }
```

In the nested Parking conversation, the customer hands over his car and eKey to the valet (line 24), which uses them to park the car (line 20). Furthermore, both participants record the relation between the certificate and the eKey (lines 21 and 25) so that the customer can retrieve his car later on.

```
29          resolve(exceptions) {
30              case :
31                  exceptions.contains(InvalidCertificate@customer):
32                      abort();
33                  exceptions.contains(TimedOut) :
34                      retry();
35          }
36      }
```

Finally, the resolve function is called by the local action managers after all roles have terminated and when at least one role has raised an exception. The exceptions parameter contains all raised exceptions in a collection which can be subsequently queried. Also note the use of the @ syntax which allows one to distinguish exceptions based on the role of the participant that raised them.

For this particular example, we have employed default exception handlers. Clearly, the programmer should be permitted to implement his own **handler** functions to incorporate different application-specific exception handling.

5 Position Statement

This paper first identified the main differences between traditional distributed systems and the new emerging field of Ambient Intelligence. These differences are all related to the essentially different characteristics of the network constellations encountered by ambient-oriented software. First of all, ambient-oriented programs are targeted towards networks populated by mobile devices, between which only *volatile connections* can exist. Furthermore the dynamic nature of such networks, implies that the availability of resources cannot be provided up front, such that the program is required to *discover available resources*. Finally, the dynamic networks under consideration may be formed entirely ad hoc, which implies that an ambient-oriented programming language should assume *no infrastructure*.

The impact of these criteria on the development of software is profound. In this paper we have conjectured that – given an appropriate programming language for ambient-oriented programming – the main concerns when developing an ambient-oriented application are :

1. Expressing synchronisation points between the different processes involved.
2. Consequently handle exceptions raised by either the processes themselves, or the non-functional exceptions (*e.g.* to signal disconnection of a partner process) raised by the distribution layer.
3. How easy is it to deploy the application on a network of mobile devices.
4. How the application can become aware of its dynamic environment.

Based on this conjecture, we have proposed the introduction of *conversations* as a programming model for ambient-oriented software development. Through the use of a simple scenario we have analysed how conversations can address the different concerns outlined above. On the other hand, this analysis also indicated room for improvement of the model. In particular, we explored possible ways to tackle both the scattering of conversations over the available devices, as well as a service discovery mechanism, which allows the scattered conversations to reconnect. Since these improvements only involve the level of the virtual machine, we claim that conversations should be considered as a viable programming model for ambient-oriented software.

References

[Agh86] Gul Agha. *Actors: a Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[BRR+00] D. M. Beder, A. Romanovsky, B. Randell, C. R. Snow, and R. J. Stroud. An application of fault tolerance patterns and coordinated atomic actions to a problem in railway scheduling. *SIGOPS Oper. Syst. Rev.*, 34(4):21–31, 2000.

[CG03] Denis Caromel and Alexandre Genoud. Non-functional exceptions for distributed and mobile objects. In *Workshop on Exception Handling in Object-Oriented Systems (ECOOP 2003)*, 2003.

[DV04] Jessie Dedecker and Werner Van Belle. Actors for mobile ad-hoc networks. In L. Yang, M. Guo, J. Gao, and N. Jha, editors, *Embedded and Ubiquitous Computing*, volume 3207 of *Lecture Notes in Computer Science*, pages 482–494. Springer-Verlag, August 2004.

[DVM+05] Jessie Dedecker, Tom Van Cutsem, Stijn Mostinckx, Wolfgang De Meuter, and Theo D'Hondt. Ambienttalk : A small reflective kernel for programming mobile network applications. Technical report, Vrije Universiteit Brussel, 2005.

[Ran75] B. Randell. System structure for software fault tolerance. In *Proceedings of the international conference on Reliable software*, pages 437–449, 1975.

[RS05] Brian T. Rice and Lee Salzman. The Slate programmer's reference manual, 2005.

[RZ97] A. Romanovsky and A. F. Zorzo. On distribution of coordinated atomic actions. *SIGOPS Oper. Syst. Rev.*, 31(4):63–71, 1997.

[VG00] Julie Vachon and Nicolas Guelfi. Coala: a design language for reliable distributed system engineering. In *Proceedings of the Workshop on Software Engineering and Petri Nets*, Dep. Of Computer Science, University of Aarhus, Denmark, June 2000. 135–154, DAIMI.

[VGR00] Julie Vachon, Nicolas Guelfi, and Alexander B. Romanovsky. Using coala to develop a distributed object-based application. In *International Symposium on Distributed Objects and Applications (DOA)*, pages 195–208, September 2000.

[XRR+95] Jie Xu, Brian Randell, Alexander B. Romanovsky, Cecília M. F. Rubira, Robert J. Stroud, and Zhixue Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *FTCS*, pages 499–508, 1995.

[XRR+99] J. Xu, B. Randell, A. Romanovsky, R. J. Stroud, A. F. Zorzo, E. Canver, and F. von Henke. Rigorous development of a safety-critical system based on coordinated atomic actions. In *FTCS '99: Proceedings of the Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*, page 68, Washington, DC, USA, 1999. IEEE Computer Society.

[ZPR03] A.F. Zorzo, P. Periorellis, and A. Romanovsky. Using coordinated atomic actions for building complex web applications: a learning experience. In *Proceedings of the 8th IEEE International Workshop on Object-Oriented Real-time Dependable Systems (WORDS 2003)*, pages 288–295. Guadelajara, Mexico, 2003.

The Callback Problem in Exception Handling

Jan Ploski[1] and Wilhelm Hasselbring[2]

[1] OFFIS, Escherweg 2, 26121 Oldenburg, Germany,
Jan.Ploski@offis.de,
[2] Carl von Ossietzky University of Oldenburg, Software Engineering Group, 26111 Oldenburg, Germany,
Hasselbring@informatik.uni-oldenburg.de

Abstract. Procedures invoked across program modules to notify about event occurences rather than to request concrete services are termed *callbacks*. They are frequently used to decouple program modules. However, their use complicates exception handling, turning default exception propagation with the established termination model unsatisfactory.
Existing advice on handling exceptions in callbacks either requires implementors to provide a no-throw guarantee or to propagate a generic exception from a callback to its invoker. We demonstrate that neither approach supports adequate exception handling in practice.
To aid programmers, we propose a tactic which mimics default propagation in the callback-less context: If possible, handle an exception locally; otherwise, propagate it to the invoked callback's *clients*, that is, modules affected by the callback's failure. We suggest early explicit specification as a means to locate such modules.

1 Introduction

The Observer design pattern [1] decouples two classes through a notification mechanism based on a specified interface known to both classes. An instance in the *subject* role notifies one or more instances in the *observer* role by sending them a message at each occurence of an agreed-upon event, typically a state change in the subject itself. The subject does not need to know the concrete type of each notified observer.[3] Also, it does not need to know the behaviour implied by the notification. Observers register with the subject prior to notifications and may unregister later on.

Instances of the classic Observer pattern exist in almost every GUI framework, often to implement the more specific Model-View-Controller pattern. More generally, one can employ callback mechanisms to facilitate communication between independent objects by connecting them through a bus which can support synchronous or asynchronous messaging. This solution can be found in various message brokers, for example in implementations of the JMS specification [2].

[3] We refer to classes "knowing" things in the sense of information being available to their programmer at design time.

The actual terms used to describe the overall concept vary considerably: callbacks, observers, listeners, slots, subscribers, views.

The relationship among communicating objects as defined by the Observer pattern gives rise to the problem of dealing with exceptions which might occur during the notification of an observer. Specifically, default exception propagation supported by imperative programming languages is inappropriate in such context, while it suffices in the simpler case of a direct invocation dependency:

In a traditional synchronous invocation scenario, it is useful to propagate failure information in form of an exception object back from an invoked method to its invoker to support adequate exception handling. This is true because the invoker is fully aware of the invoked method's contribution to satisfying the invoker's postcondition. The programmer may therefore either find an alternative way of satisfying the postcondition or broaden the recovery context by restoring local invariants and reporting failure farther up the call chain.

2 Exception Propagation from Callbacks

In contrast with the above scenario, exception propagation from callbacks is challenging. In this section we explain the problem and present two basic approaches which fail to provide a solution to the general case. We illustrate their drawbacks on examples and then briefly discuss practical consequences of their failure.

2.1 Callbacks and Information Hiding

Callbacks enforce information hiding. A callback's invoker is, by design, aware of neither the concrete implementation nor the abstract goals accomplished by the executed notification. More formally, the complete postcondition of a notification is not known at development time of the notifier.

We assume for simplicity that the precondition is empty. In reality, the notifier may be obliged to satisfy certain preconditions, such as executing the notification on a particular thread. Such preconditions can be easily specified as part of the notification interface.

Even if the information about the nature of work performed by the notified object is available, it should be treated as secret for the sake of reusability and independent maintainability of the invoker implementation.

2.2 Basic Approaches

The above considerations motivate the basic approaches to exception propagation from callbacks presented next: The notified objects must either provide a no-throw guarantee [3], or they may only supply limited information about their failure by throwing some generic exception.

Providing a No-throw Guarantee Imposing a no-throw restriction on each notified object is attractive from the point of view of a notifier's implementor, as it frees her from any exception handling responsibilities. However, it requires that the full information necessary for exception handling be available to the notified object. Furthermore, it requires that the constraints on the behaviour of the notified object allow appropriate exception handling. While the first requirement is self-evident, the latter needs further elaboration.

As an example, consider the notification mechanism typical in event-driven programming utilizing GUI frameworks. An application programmer provides concrete implementations of abstract event handler methods specified by a GUI framework, invoked to notify about key strokes and mouse movements, and to request repainting of widgets. To maintain a responsive GUI, event handlers must satisfy soft real-time constraints during their execution. Moreover, because of the sequential nature of event handling, they must not rely on any service provided by other event handlers. These constraints make it impossible to report an exception to the user through the GUI or solicit user input directly in an event handler.

To work around the problem while still maintaining the no-throw guarantee, one must resort to asynchronous exception handling, that is, delegate the processing of exceptions to another thread and postpone it until the remaining registered event handlers have seen the current event.

Propagation to the Invoker If the requirement of a no-throw guarantee turns out difficult to satisfy in practice, one might hope that the second approach mentioned, propagating a generic exception to the notifier, has better prospects. Alas, this is not true. To illustrate why, we depart from the domain of GUI programming and consider a scenario in which components react to a database update triggered by another component.

In this scenario, the updating component manages its own data stored in a database table which is also shared with other components. The components are unaware of each other and are coordinated by an implementation framework. After an update of the shared table, each component is notified by the framework to execute some component-specific action, during which an internal exception might occur. In this case, the exception is propagated to the framework, which in turn propagates it to the component responsible for the update.

While this approach seems reasonable at first glance, the updating component is, by design, unable to handle any exception originating from another (notified) component. In fact, the occurence of such an exception might not at all be related to the update event, provided that the updater only uses the shared table to manage its own private data. Obviously, exception propagation to the notifier (the framework) and consequently to the event originator (the updater) is a design flaw.

While the simplified scenario described above is constructed for the sake of our discussion, it is inspired by a similar problem which occurs in the Eclipse Workbench [4] when a plug-in updates a project description.

2.3 Practical Considerations

At this point, we would like to reflect on how shortcomings in the described approaches affect the quality of exception handling in software. While implementation of GUI event handlers does not exceed the level of training normally expected from a client-side application programmer, asynchronous exception handling and related multi-threading concerns might well do. However, we are not aware of any GUI framework with explicit support for handling exceptions asynchronously. This task is left as a challenge for GUI programmers.

Unfortunately, the transition from the expected (easy) to the actual (difficult) nature of event-driven GUI programming caused by exceptions rapidly leads to inadequate exception handling in real code. Rather than expend a considerable amount of effort, it is more likely that implementors will skip the issue of exceptions altogether. A quick-and-dirty solution to the problem involves logging exceptions, optionally followed by propagation to the GUI framework's default handling mechanism. The latter can obviously do nothing more than display a cryptic error message or terminate the program. Such shortcuts are often taken, because leaving out exception handling seems natural when prototyping GUIs, and its importance continues to be neglected while a prototype evolves into a final product.

3 Proposed Solution

We have described two unsatisfactory approaches to exception handling in callbacks. In this section we present a more sophisticated approach.

Our solution is based on the already mentioned observation that exception propagation in the sequential call chain without callbacks proves sufficient and effective. In short, exceptions should be propagated towards computational contexts which fulfil at least one of the following requirements:

1. The context is (negatively) affected by the exception's occurence ("do not withhold bad news").
2. The context is capable of compensating for the negative consequences of the exception ("do not carry your problems to a place where no help is available").
3. The context is capable of removing the exception's cause to prevent its future occurences ("treat the disease, not the symptoms").

Note that with these rules we neither specifically prescribe nor forbid propagation of an exception to the immediate invoker of a method, as the previous approaches do. After introductory definitions we explain the above recommendations in greater detail.

3.1 Definitions

We adapt the terminology defined by Cristian [5]. Sequential programs (e.g., object methods) are invoked to cause transitions from an initial storage state

to an intended final storage state. A *storage state* is a mapping from object identifiers to object values; these values may model states of real-world entities. A *standard specification*, or *postcondition*, of a program is a predicate over the required initial and intended final storage states. Storage states in the domain of a postcondition form the corresponding *precondition*.

Exceptions are run-time events that occur when a program's execution diverges from its standard specification, regardless of whether the divergence is detected or not. The divergence occurs as soon as a storage state is entered such that the continued execution of the program cannot result in fulfilment of the standard specification.

In another sense, exceptions are named entities which may be used by a software developer to signal detection of divergences described above. Exceptions should be raised to indicate the impossibility of providing a program's specified standard service, in other words, to report a violation of its standard specification, or the program's *failure*. *Exceptional specifications* may be written to describe the intended final storage state reached by a program upon occurence of a named exception.

We would also like to stress that exceptions are *undesired* events. Their signaling should not be used for implementing functional requirements of software, but rather for addressing unavoidable threats that endanger the correctness of a specific implementation. This definition supplements the former one in that it discourages some uses of exception handling mechanisms (cf. [6], [7]).

3.2 Impact of Exceptions

As background information for our first rule of propagating exceptions from callbacks towards affected computational contexts, we now consider the possible consequences of such exceptions.

A callback's standard specification has to be understated at invoker's design time, such that concrete callback implementations may become proper behavioural subtypes [8]. The standard postcondition of a callback, as specified in the callback interface, may be simply *true*, or it may reflect restrictions imposed on its behaviour. In any case, a concrete callback may (and will) only strenghten the standard postcondition.

If an exception occurs during a callback's execution, its standard specification is violated. Thus, the entered final storage state differs from the intended one. Additionally, the entered state may be internally inconsistent, even if exception safety is guaranteed at the level of individual objects. The global consistency requirements for a system can be specified as a predicate over allowable storage states at well-defined points of execution. The predicate can be also viewed as a conjunction of *system invariants*, of which *object invariants* are a special case restricted to specifying relationships between a single object's representation variables. The traditional notion of exception safety is restricted to preserving object invariants.

We next provide a realistic example of a case in which an exception leads to an unintended yet consistent storage state, as well as one in which an inconsistency occurs.

Exception With Preserved System Invariants A preferences dialog in a desktop application lets the user modify some preferences and finally click the "Apply" button to persist them. The "Apply" button then becomes disabled to indicate no outstanding changes. If an exception occurs in the "Apply" event handler, the system will not advance to the next correct state, even though invariants of all objects are preserved (i.e., the "Apply" button remains enabled). The final storage state, even though internally consistent, is unacceptable for the user, who expects that preferences have been saved.

Exception With Broken System Invariants In an application using the MVC pattern, different state aspects of a model object are visualised by separate views. After an exception occurs during state change notification, a view might no longer reflect the model state and no longer show content synchronised with other, successfully updated views. Two system invariants are broken in this case: one over the model and view state, and another one over the state of all views.

3.3 Determining Affected Code

We have shown how an exception occurence in a callback leads to entering an unintended storage state as its immediate consequence. In an object-oriented context we are interested in determining methods with preconditions that become impossible to satisfy through normal execution continued from the entered exceptional storage state. Knowing these methods, one can notify their enclosing objects before their next regular invocation. The rationale is that the objects may then switch over to an alternative implementation, or at least prepare to fail gracefully. This is analogous to propagating an exception to the invoker of a service method.

More formally, we need to find elements of two object sets:

1. M, the set of objects modified during the callback's invocation
2. D, the set of callback's *clients*, defined as objects dependent on the updated state of elements from M

For a failed method invocation in a sequential program without callbacks, M typically consists of objects known to the invoker and D contains the invoker itself or objects controlled by the invoker. Members of these sets can be determined by following the next execution steps that would be taken after the failed invocation. Accordingly, it makes sense to refer to the invoker as the failed method's *client* [6]. However, when callbacks are employed, the method's invoker steps out of its usual client role.

To avoid the algorithmically difficult computation of M and D from the source code alone, we suggest that both sets are specified by the programmer

when the callback's purpose becomes known. The specification should be written regardless of whether exceptions are expected to occur during callback invocations or not. In programming languages which support specification of postconditions, M does not require to be explicitly enumerated, because it follows from a postcondition. However, D still has to be specified explicitly. With this preparation, when later implementations of the invoked methods signal exceptions, the exceptions can be propagated to callback's clients as an early warning that the state on which they depend has become inconsistent or outdated.

To illustrate why the up-front specification of M is worthwhile, we describe an informal, labour-intensive procedure for reconstructing it from source code:

Let M be an initially empty set of affected objects. For every statement s_i of the callback implementation:

1. Let C be the set of method invocations from s_i.
2. For every method invocation c_j from C:
    1. Let o_j be the object on which c_j is invoked and P the set of objects passed in as formal parameters.
    2. If c_j does neither modify the state of o_j nor of any object from P, proceed to c_{j+1}, otherwise:
    3. Add the object(s) with modified state to M.

Finding out whether a method modifies an object's state, lacking specification, requires analysis of all transitively invoked methods. This task becomes non-trivial and error-prone when the invoked methods are themselves callbacks, configured and registered at some other point of execution. On the other hand, if an accurate specification is available for a (callback) method, the analysis of its implementation is not required, speeding up the process.

Updated objects from M could theoretically be examined for their participation in system invariants affected by the callback's invocation in order to construct the callback's clients set D. However, we are unaware of practical approaches for reconstructing system invariants based on static analysis of source code alone (for a brief description of a dynamic approach, see section 4.1). We must therefore assume that informal understanding of the intended role of each element from M and the relationships between them and other objects is utilised in this process. However, such understanding is only likely to exist at the time when the callback is originally added to the system.

These observations highlight the need for early specification of callback effects. However, two related problems remain as topics for further research:

1. The proposed declarative specifications concern the callback's clients, which could in principle remain hidden if we disregarded exception handling. The trade-off for achieved explicitness is a decrease in modularity.
2. The proposed declarative specifications may be difficult to maintain as programs evolve.

3.4 Exception Handlers

According to our second proposed rule ("do not carry your problems to a place where no help is available"), not every exception originating from a callback should be directly propagated to its clients defined in the previous section. Before such propagation occurs, an attempt should be undertaken to find exception handlers that can reduce or even completely eliminate the signaled exception's consequences.

In practice every exception-throwing callback should be wrapped into one that catches all exceptions and decides about their propagation. The wrapper object communicates exception occurences to handlers which can be either specified statically or registered at run-time. If required, notification of handlers may be performed asynchronously, like in the GUI example from section 2.2.

There are two basic courses of action for handlers: restoration of system invariants and enforcing the original callback's postcondition (cf. section 3.2).

The effort expended on the restoration of system invariants is related to the amount of damage present when the exception is signaled. If available, a description of the damage should be encapsulated in the exception itself. Both roll-back and roll-forward recovery can be performed to restore the consistent storage state. It is even more preferable for the callback to perform the necessary clean-up actions *before* signaling the exception, however, it may not be always possible nor sufficient.

The original postcondition can only be enforced by retrying the callback's invocation if the exception's cause was removed by a handler. Otherwise, a less satisfactory exceptional postcondition may still be enforced. The exceptional postcondition may be functionally inferior to the original one, which calls for a redesign by weakening of system invariants, or it may offer lower quality of service (e.g., use a more robust, but slower algorithm).

Depending on the handlers' achievements, (some of) the callback's clients may still need to be notified about the exception's occurence. The communicated information should include relevant countermeasures undertaken by the handlers.

Deciding whether and which clients should be informed, and to what extent, depends on a specific application. For example, if a maskable disk failure occurs in a RAID array, clients which depend on the immediate success of the IO operation do not need to be notified about the exception, however, it is prudent to notify a monitoring client so that the faulty disk is replaced (cf. [9]).

3.5 Removing Causes of Exceptions

Exception handlers can provide two kinds of service: temporary workarounds and permanent fixes.

The first category implies a degradation of service level, which may or may not be noticeable to the clients, like in the above RAID array example.

The "permanent fix" category is more interesting. It requires both determining and removing the fault which causes an exception—two tasks which may be

sometimes economically infeasible. In simple cases, easy configuration changes may be all that is required. In complex cases, significant investments in hardware redundancy and improved system design may be necessary. Execution of permanent fix handlers may require human intervention, and a system may enter prolonged exceptional storage states while the permanent fix is being prepared. To reason about exception handling in this context, explicit specification of persistent exceptions [10] becomes important. Such exceptions can be caused by faults in system configuration or data that exist independently of any executed program. They do not fit well into the traditional programming-language-centric view of exception handling.

Unfortunately, we cannot at present offer sound advice on techniques for removing faults. One step in the right direction would be constructing their taxonomy based on examination of real-world systems. After faults are classified into well-defined categories aligned with strategies applicable to their removal, design patterns and components might be conceived to not just handle resulting exceptions, but often to avoid the need for exception handling altogether. It is worth pointing out that even if this succeeds largely, exception handling mechanisms will still remain useful for addressing unpredictable, dormant faults.

4 Related Work

In this section, we briefly point out related work from the research community and articles published by software developers.

4.1 Academic Research

The concept of uniform programmatic exception handling was popularised by Goodenough [7]. Goodenough informally describes a syntax and a proposed notation for both the termination and the resumption model of exception handling. Our suggested way of handling exceptions in callbacks relies only on the more popular termination semantics, but it replaces the default exception propagation when necessary.

Cristian [5] provides a formal semantics for the termination model. He focuses on an examplary direct invocation scenario and does not specifically consider callbacks, though he briefly discusses exception propagation from an indirectly invoked method.

Meyer [6] defines guidelines for using exceptions according to the termination model under the label “organised panic” in his textbook on object-orientation. Like Cristian, he does not address propagation of exceptions from callbacks, nor does he elaborate on how (and when) contracts should be specified for them in order to successfully apply his proposed basic rules of exception handling.

Our described problem can be viewed as a specific instance of the general conflict between exception handling and object-oriented abstraction discussed by Miller and Tripathi [9]. However, their paper does not provide guidance with respect to balancing information hiding for modularity and exposure for exception handling.

Buhr [11] points out the problems resulting from premature specification of exception lists for routines invoked through pointers (i.e., callbacks) and polymorphic methods. His observations support our view that specification of exceptions is not practical at the level of callback interfaces and must be delayed until implementation.

Awareness of system invariants aids determining code affected by a callback's failure. Ernst [12] proposes a dynamic approach for reconstructing invariants from executed code and provides a related tool, Daikon. Only relatively low-level invariants are recognised automatically, however, they may provide valuable hints for the programmer about more abstract invariants.

Barnett and Naumann [13] propose a methodology for specifying invariants across object ownership boundaries, which could be potentially adopted as a formalism for expressing specifications suggested in our paper.

Borgida [10] proposes a language for describing and handling persistent exceptions in object-oriented databases and points out that a relationship exists between run-time exceptions and flaws in data maintained by information systems. Our argument that exception handling in a broad sense should ultimately aim at removing the causes of exceptions is based on similar observations.

4.2 Software Industry

The Observer pattern is one of the very basic patterns commonly taught to software engineering students and widely applied in practice. Accordingly, one would assume that the problems described above were already addressed by researchers and practitioners. A survey of articles on "best practices" in exception handling reveals the opposite.

Doshi [14] does not address the issue at all, while he provides the following advice: "When a method from an API throws a checked exception, it is trying to tell you that you should take some counter action. If the checked exception does not make sense to you, do not hesitate to convert it into an unchecked exception and throw it again". This is wishful thinking, as should be obvious from our previous example.

Shenoy [15] follows the EJB specification [16] by distinguishing between application and system exceptions. The latter are handled by the EJB container in a default way (logging, aborting the current transaction). Shenoy suggests that application exceptions which have been propagated to unaware components should be caught and converted to system exceptions. He does not address the inappropriateness of propagation of such exceptions, but rather treats them as a "real-world complexity".

5 Conclusions and Future Work

Important practical problems remain that make the proper application of established exception handling concepts difficult for programmers using today's object-oriented programming languages.

We have presented one such problem—exception handling in callbacks—and provided a set of relevant examples based on the Observer design pattern. We have also shown that existing simple guidelines available to programmers for solving this problem are either overly restrictive or leave the most challenging decisions open. Finally, we have described a more sophisticated alternative approach based on the early specification of callback effects combined with the well-established concept of exception propagation.

The remaining issues to be addressed by future research include:

1. Syntactic and semantic description of our suggested additional specifications in the context of a chosen object-oriented programming language: An approach similar to [13] could be taken for specifying system invariants which cross object boundaries. We also plan to evaluate the support offered by existing tools for static and dynamic control and data flow analysis [17].
2. Empirical evaluation of the proposed additional specifications' usefulness: Metrics have to be defined that measure the amount of work expended on exception handling and its quality in software products. Such metrics should build upon prior research on misuse of exception handling.

In a broader view, we feel that exception handling needs tools and techniques beyond simple syntactic checks such as determining missing or empty handlers. In addition, discovery of failure modes, intent, and stakeholders of a software application should be made possible. Offered solutions should fit into a bigger framework for building dependable software by recognising the role of other quality-of-service attributes and multiple programming languages.

The problem described in this paper is not at all restricted to small object-oriented programs employing specific design patterns, as could be subsumed from our simple examples and definitions. It prominently figures in component-based system development (CBSD), where it is a basic assumption that independently developed software artefacts are combined and configured at a later time in flexible ways to obtain desired functionality.

Deployed components may invoke each other either as services or to deliver notifications. All invocation relationships cannot be determined until integration time. For these reasons, approaches which position exception handling in the programming domain or request that exceptions are explicitly specified by developers as part of component interfaces [18] appear too limited.

Instead, we feel that the specification of exceptions and implementation of handlers need to be delayed until all required information about possible exception sources and propagation paths becomes available. It could be shortly before deployment or (re-)configuration of a component-based system. Developing methods and tools for realising this vision represents our long-term research goal.

References

1. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, Massachusetts (1995)
2. Hapner, M., Burridge, R., Sharma, R., Fialli, J., Stout, K.: Java Message Service. `http://java.sun.com/products/jms/docs.html` (April 2002)
3. Stroustrup, B.: Exception safety: concepts and techniques. In Romanovsky, A., Dony, C., Knudsen, J.L., Tripathi, A., eds.: Advances in exception handling techniques. Volume 2022 of LNCS., New York, NY, USA, Springer-Verlag New York, Inc. (2001) 60–76
4. Arthorne, J.: Project builders and natures. `http://www.eclipse.org/articles/Article-Builders/builders.html` (January 2003)
5. Cristian, F.: Exception handling. Technical Report RJ5724 (57703), IBM Almaden Research Center (1987)
6. Meyer, B.: Object-Oriented Software Construction. second edn. Prentice-Hall, Inc., Upper Saddle River, NJ, USA (1997)
7. Goodenough, J.B.: Exception handling: issues and a proposed notation. Commun. ACM **18** (1975) 683–696
8. Liskov, B.H., Wing, J.M.: Behavioural subtyping using invariants and constraints. In: Formal methods for distributed processing: a survey of object-oriented approaches. Cambridge University Press, Cambridge, UK (2001) 254–280
9. Miller, R., Tripathi, A.: Issues with exception handling in object-oriented systems. In Aksit, M., Matsuoka, S., eds.: ECOOP '97 - Object-Oriented Programming. Proceedings of the 11th European Conference. Volume 1241 of LNCS., Springer-Verlag New York, Inc. (1997) 85–103
10. Borgida, A.: Language features for flexible handling of exceptions in information systems. ACM Trans. Database Syst. **10** (1985) 565–603
11. Buhr, P.A., Mok, W.Y.R.: Advanced exception handling mechanisms. IEEE Trans. Softw. Eng. **26** (2000) 820–836
12. Ernst, M.D.: Dynamically Discovering Likely Program Invariants. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington (2000)
13. Barnett, M., Naumann, D.A.: Friends need a bit more: Maintaining invariants over shared state. In: MPC '04: Proceedings of the Conference on Mathematics of Program Construction. (2004)
14. Doshi, G.: Best practices for exception handling. `http://www.onjava.com/pub/a/onjava/2003/11/19/exceptions.html` (2003)
15. Shenoy, S.: Best practices in EJB exception handling. `http://www-128.ibm.com/developerworks/linux/library/j-ejbexcept.html` (May 2002)
16. DeMichiel, L.G., et al.: Enterprise JavaBeans specification, version 2.1. `http://java.sun.com/products/ejb/docs.html` (November 2003)
17. Detlefs, D.L., Leino, K.R.M., Nelson, G., Saxe, J.B.: Extended static checking. Research Report 159, Compaq Systems Research Center, Palo Alto, USA (1998)
18. Romanovsky, A.: Exception handling in component-based system development. In: COMPSAC '01: Proceedings of the 25th International Computer Software and Applications Conference, Los Alamitos, CA, USA, IEEE Computer Society Press (2001) 580–588

Handling multiple concurrent exceptions in C++ using futures

Matti Rintala, matti.rintala@tut.fi

Tampere University of Technology

Abstract. Exception handling is a well-established mechanism in sequential programming. Concurrency and asynchronous calls introduce the possibility for multiple simultaneous exceptions, which complicates exception handling, especially for languages whose support for exceptions has not originally been designed for concurrency. This article discusses these problems and presents a mechanism for concurrent exception handling in C++.

1 Introduction

Exceptions have become more and more common as a means of handling error situations during the last decade, especially because commonly used programming languages now support exception handling as a language feature. However, the basic ideas behind exception handling are far older [1]. Exceptions are now considered a "standard" way of signalling about exceptional situations and they have widely replaced the use of specially coded return values, additional boolean state flags etc.

Basically exception handling is about separating exception handling code from the "normal" code. This improves readability of the code by structuring it logically. Exception handling also introduces its own error handling flow-of-control to the program, so that the normal code does not have to explicitly prepare for every error situation and divert the program to appropriate error handling code.

Because exception handling is essentially about flow-of-control, concurrency cannot be added to a programming language without affecting exceptions. Concurrency introduces several (usually independent) threads of execution, each of which has its own flow-of-control. This causes several problems, some of which are analysed in [2]. Asynchronous calls and futures (delayed return values) complicate the problem even further.

There are several ways to add exception handling to a concurrent object-oriented language, all of which have their benefits and drawbacks [3]. However, when adding concurrency to an *existing* programming language (like C++), the exception handling mechanism has already been fixed, and the added concurrency features should be designed to be as compatible with the existing mechanism as possible. This unavoidably means some compromises to the elegance of the system.

This article shows how to add support for multiple concurrent exceptions to C++. The solution is library-based and does not extend the normal C++ syntax. The mechanisms described in this article are as compatible with normal C++ exceptions as possible, and allow the programmers to choose the exception handling strategy best suited for their programs. The solution described in this article is part of KC++ [4], a concurrent C++ system based on active objects.

2 Asynchronous calls and futures

Futures are a mechanism to achieve concurrent asynchronous calls which return a value. They were originally introduced in Multilisp [5], but have since been used in many other languages, like wait-by-necessities in C++// [6] or IOUs in the Threads.h++ library [7]. The current Java 5 also provides futures.

Futures are placeholders for an eventual return value from an asynchronous call. In C++ they can be implemented as templates parametrized with the type of the return value. When an asynchronous call is made, the caller immediately gets an "empty" future object representing the return value, while the call continues to execute concurrently. When the call completes, its return value is transferred to the future, from where the caller can fetch it. If the value of the future is asked for before the call is completed, the future suspends the execution of asking thread until the value is available.

In KC++ futures can also be copied and assigned without waiting for their values, and they can be passed to other execution threads. When the call is completed, the return value is automatically propagated to all futures copied or assigned from the original future.

Normally futures get their value when an asynchronous call completes and returns. However, sometimes its practical to delay the future even further, until the completion of *another* call for example. For this purpose KC++ has *future sources*. An indefinite number of "empty" futures may be created from a future source, and these futures may be returned as return values. These futures receive their value when the `bind` method of the future source is called. This allows manual and explicit synchronisation using futures.

3 Issues caused by asynchrony

Introducing asynchronous calls (concurrency) affects also exception handling. It changes the way control-flows work in a program, and this has to be taken into account when designing exception handling in such a system. This section describes some of the main issues caused by asynchrony.

If exceptions are propagated from an asynchronously call back to the caller, it is not self-evident where they should be handled. The caller has already continued after making the call, and may in fact have terminated its execution. Usual exception handling mechanisms in sequential languages bind exception handlers to a specific part of the code (`try`-blocks etc.). An exception caused by an

asynchronous call may occur at any part of the calling code, so it is not obvious which exception handlers should be considered.

Several asynchronously executing threads also introduce a possibility for several exceptions raised simultaneously in different threads. If these exceptions are sent to other threads, this can result in more than one exception being active in a *single* thread — a situation which is problematic and forbidden in many programming languages (for example C++).

If return values of asynchronous calls are handled with futures, that also affects exception handling. In a sense, an exception *is* a return value. In synchronous calls return values and exceptions do not conflict, because if an exception is thrown from a call, a normal return value is not created. However, futures are created in advance, and the represent a return value of an asynchronous call.

If an exception is thrown from an asynchronous call, a future for the return value already exists and may in fact have been copied and assigned to other futures as well, maybe even sent to other threads. If exceptions are regarded as another form of return value, they have to also affect the way futures work.

Finally exceptions from asynchronous calls become a synchronisation issue. An exception also represents the termination of a call, so all other threads waiting for the call to complete (through futures or some other mechanism) should be signalled. The question then becomes whether these threads should also automatically receive the thrown exception. This question becomes especially interesting if there are more than one thread waiting, because normal exceptions are not duplicated, but propagating an exception to several threads would unavoidably mean duplication of the exception.

4 Exceptions and futures

If an exception occurs during the execution of an asynchronous call, it must be propagated to the caller. Since the call is asynchronous, the only way for an exception to propagate is through futures.

The simplest situation arises when an exception is thrown from an asynchronous call. When this happens, the KC++ library code responsible for handling the asynchronous call catches the exception. It marshals the exception object and embeds the data to the call's reply message which is sent back to the caller.

When the caller side KC++ code receives a message containing an exception, an appropriate future object is first created. Then a copy of the exception object is created from the message data and a pointer to this exception object is stored inside the future.

If the future object is assigned to other futures or copied, resulting local copies of the future share the future's data structure and therefore also the exception object. If the future is sent to another process, the future's marshalling code also marshals the exception object using the same mechanisms as before.

Whenever the value of a future is needed, the future checks whether it contains an exception object instead of a value. If this is the case, the future's `value`

method throws the stored exception, which is then handled using normal C++ exception handling mechanisms.

Although this mechanism makes it possible to propagate exceptions from asynchronous calls, it has some differences to normal C++ exception semantics. Normally a thrown exception can only be caught once (unless an error handler re-throws an exception). However, in KC++ the value of the same future may be requested several times (either through the same future or other futures copied from it) and in several concurrent execution threads. Although this behaviour is logical and practically the only option, programmers must consider its implications in the program logic — the same exception may get thrown many times and in several execution threads.

Even though future sources do not represent actual return points from calls, it is still possible that an error situation occurs and the holder of the future source wants to inform others of this situation. For this reason future sources provide a way to propagate exception objects manually (i.e. without initially throwing the exception object) to futures generated from the future source.

Usually future sources are given their value using the `bind` method, which stores the value in the future source and also sends it to all generated futures. In addition to this, future sources contain a method called `bindThrow`. This method takes an KC++ exception object as its parameter. The method copies and stores the exception object inside the future source and also sends to it all generated futures as an exception. When the value of these futures is needed, the futures throw the exception as described previously. The same applies also to all futures that are generated from the future source after `bindThrow` is called.

5 Multiple simultaneous exceptions

Asynchronous calls make it possible to end up in a situation where several exceptions are active simultaneously. If the caller continues its execution while a call is active, both execution threads may end up throwing an exception, and these both these exceptions may end up in the calling thread. The C++ language cannot handle more than one exception in the same try-block (although "nested" exceptions are possible) [8, § 15.5.1], which forces some compromises to be made in KC++.

One problem is caused by the fact that the C++ exception model makes it impossible to return to the try-block after an exception is handled. When the C++ exception handling mechanism searches for a correct exception handler, it permanently exits from try-blocks, destroying their local variables. This includes the try-block which contains the correct handler. After the exception is handled program execution continues from the point after the try-catch-compound containing the chosen exception handler.

If several exceptions occur simultaneously, only one of them can be handled normally. However, since handling of even one exception means abandoning try blocks (and their respective catch-handlers), it would be impossible to search

handlers for the rest of the exceptions, because the same catch-handlers are no longer available (the program execution has already left the try-block).

Another problem with handling concurrent exceptions one at a time would be to choose the order in which the exceptions should be handled. Exceptions may be thrown from several (mutually independent) locations, so deciding the correct handling order would either require a global priority scheme for exceptions, or mean that there would have to be a mechanism for informing the relative priority of a thrown exception (and maybe change it during exception handling when the exception is propagated to other parts of the program). Unfortunately this kind of simple priority scheme is not enough in every situation [2].

The third and maybe most important aspect in concurrent exception handling is the fact that several concurrently thrown exceptions may in fact be caused by the same abnormal situation. If the exception objects would just be copies of each other, they could be easily reduced to one exception. However, sometimes the nature of the actual exceptional situation may be only understood by analysing *all* of the component exceptions it causes, in which case it is important that the exception handling mechanism can cope with several exceptions at the same time. Sometimes it would also be beneficial to be able to *replace* a set of exceptions (caused by the same exceptional situation) with a new exception which represents the abnormal event that actually happened.

Writing exception handling becomes easier if these reductions of exceptions can be performed *before* an exception handler is chosen, because then exception handlers may still catch single exceptions whose static type represents the exception type. Since it is impossible to give a global rule for reducing multiple exceptions to one, it is important that the program can provide its own algorithms for the reduction (and have different reduction algorithms for different exception contexts).

Figure 1 show the structure of KC++ exception handling mechanism. It is based on futures and future sources, as well as *future groups* for collective synchronisation, *compound exceptions* for handling multiple simultaneous exceptions, and *reduction functions* for analysis and reduction of exceptions. The behaviour of the mechanism is described in the following sections.

5.1 Compound exceptions

The C++ language can only handle one exception at a time on one level. Another exception may be raised during stack unwinding triggered by the first exception, but these additional exceptions must be handled to conclusion before stack unwinding proceeds further. In a concurrent program this limitation is problematic, because several exceptions may be propagated to a try block from asynchronous calls.

KC++ provides a *compound exception* class to represent a set of simultaneous exceptions. It is a normal KC++ exception class, but it may contain an unlimited number of other KC++ exceptions. This approach is somewhat similar to Java's chained exceptions, where each exception object may also contain a reference to its *cause*, which is another (single) exception object.

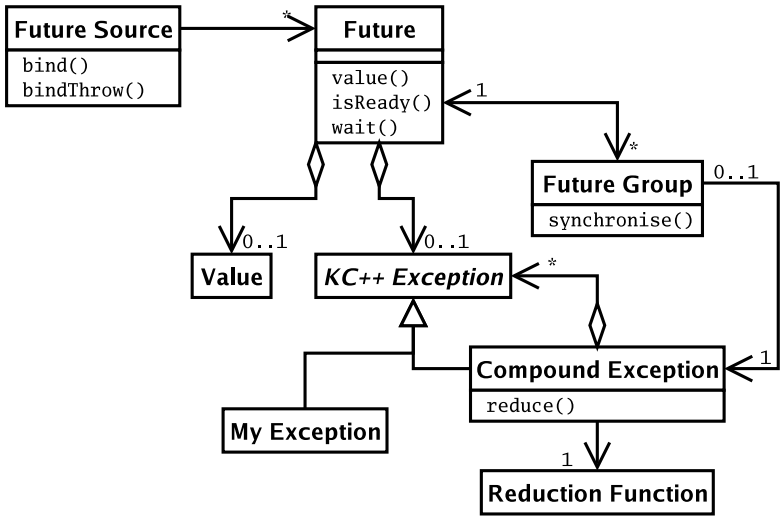


Fig. 1. Structure of KC++ exception handling

As compound exceptions are handled as part of normal KC++ exceptions, they can be passed between execution threads. The interface of compound exceptions allows addition of new exceptions and removal of existing ones. In addition to this, all exceptions may be moved from one compound exception to another. This makes it possible to combine compound exceptions in reduction functions or error handlers. KC++ also provides iterators which can be used to iterate through all exceptions in a compound exception.

Compound exceptions are mostly used to store exceptions from asynchronous calls and collect them for further analysis and reduction. *Future groups* (explained in Section 5.2) are used to mark a set of futures whose exception status must be analysed together. These future groups collect exceptions to a compound exception, where the set of exceptions can be reduced using reduction functions explained in Section 5.3.

Usually exceptions in a compound exception are also stored in futures, whose lifetime may or may not have been ended when the exceptions are added to the compound exception. Sometimes the futures may be destroyed as part of stack unwinding during exception handling, sometimes they may still exist after exception handling has been completed. For this reason sharing is used between futures and compound exceptions. They both share the same exception object and use reference counting to find out when it is safe to destroy the object. The sharing means however, that even after exception handling has been completed using the compound exception, the futures will throw the same exception object again, if their value is asked for.

In addition to actual compound exceptions, KC++ allows any KC++ exception object to contain a set of other exceptions. This makes it possible to select the most suitable exception from multiple exceptions (using reduction functions)

and then embed the rest of the exceptions in the selected one. This way upper level error handlers can also have access to all necessary exceptions.

5.2 Future groups

Normal C++ exception handling is based on try blocks surrounding the code in which exceptions may occur. Asynchronous calls make this approach problematic. As the calling code does not wait for the call to complete, it is quite possible and even likely that it will exit the try block before all possible exceptions have occurred. This makes it hard to combine conventional-looking error handling code with asynchronous calls. Exceptions from the asynchronous calls are stored in futures, which may be copied and sent to entirely other parts of code before values of the futures are asked and the stored exceptions are thrown.

Sometimes these "delayed" exceptions are exactly what the programmer wants, but in many cases it is necessary to write one or more asynchronous calls *within* a try block and not leave the block until possible exceptions have occurred. Checking each future separately would be awkward and not even straightforward in case of multiple exceptions. For these reasons KC++ offers *future groups* to help with synchronisation and exception handling.

Future groups are objects to which futures may be registered. They have an operation `synchronise`, which waits for the futures to receive a value (or exception). Normally future groups wait for all their futures to become ready, making future groups similar to barrier synchronisation [9, Ch. 4] or future sets in ES-Kit [10].

If exceptions are found in the futures during synchronisation, the future group collects the exceptions in a compound exception. This compound exception is either The future group's internal object or an external object provided by the programmer. The exceptions in the compound exception can then be reduced using reduction functions (Section 5.3), and an exception suitable for the situation can be thrown.

When the program asks for a value of a future belonging to a future group, the future first waits for the asynchronous call to complete, if necessary. Then, if the call ended with an exception, the future asks for its future group to perform synchronisation. This makes sure that all necessary exception information is available before exception handling is started. Finally, necessary exception reduction is performed and an appropriate exception is thrown. If the asynchronous call did not end with an exception, the future directly returns the call's return value, since synchronisation or reduction is not needed. An alternative strategy would be to perform synchronisation in this case also.

If the future group is destroyed before its synchronisation method is called, the destructor of the group performs synchronisation automatically. This makes future groups very close to the useful RAII (Resource Acquisition Is Initialisation) design idiom [11, Ch. 14] which is very common in programs using C++ exceptions. In RAII each resource is wrapped inside an object whose destructor releases the resource, preventing the possibility of resource leaks even in case of exceptions. Future groups represent a need to synchronise with certain futures

(asynchronous calls). Future groups make sure that exceptions propagating from futures are not silently ignored if the future is not used elsewhere.

The following code demonstrates the use of future groups:

```
Future<int> f1;
try {
  FutureGroup group;
  f1 = aobj.call1(); // An asynchronous call
  group.add(f1);    // Register f1 to the group
  // Request synchronisation without storing future:
  group.add(aobj.call2());
  int i = f1.value(); // Synchronises with group first
} // Destruction of group synchronises with all futures
catch (const E1P& e1)
// ...
```

5.3 Reduction functions

As mentioned previously, especially in concurrent programs several exceptions may be thrown in several execution threads by a single exceptional situation. In that case it would be beneficial if these exceptions could be reduced to a single exception object representing the complete error situation. Even if there are several *independent* active exceptions simultaneously, finding "the most important" of these exceptions depends very much on the program in question.

Because there is no single all-purpose way to reduce a set of exceptions to a single exception or a smaller set of exceptions, KC++ does not force programs to adapt to any single behaviour. Rather it allows programs to register *reduction functions* to compound exceptions. Reduction functions can analyse the current set of exceptions, alter it, and select an appropriate exception to be thrown. KC++ provides several ready-made reduction functions to cover most common reduction strategies. Reduction functions can also be composed from ready-made parts using template metaprogramming.

Reduction functions are function objects which can be registered to a compound exception. After a future group synchronises with its futures, it collects possible exceptions from those futures to a compound exception object. Then it calls the reduction function. The compound exception and information about the current exception handling status is passed to the reduction function as parameters.

The reduction function may use these parameters to decide how to simplify or modify the set of exceptions given to it. It may add or remove exceptions to or from its parameters. If no prior exceptions are currently in effect, the reduction function may return an exception, which it regards the "most important" of the given exceptions. Alternatively it may create a completely new exception object representing the current exception situation and return it. The possibilities for reduction functions have been kept as open as possible in order not to restrict different reduction needs found in different programs.

There is one limitation — a reduction function may not throw an exception if another exception has already been thrown. This is dictated by the C++ language itself. It is impossible to replace an exception object with another before it has been caught in a catch-clause. However, since KC++ futures automatically perform synchronisation and reduction, usually reduction is performed *before* any exceptions have been thrown. KC++ also offers a macro which can be added to the end of a try-block in order to catch KC++ exceptions and call a reduction function afterwards to handle already thrown exceptions.

The following code shows a simplified way to handle multiple exceptions in a loop, using a reduction function to decide on what exceptions to throw:

```
CompoundExcp ce(myReductionf); // Compound with custom reduction
while (!ce.reduced()) // While something to reduce
try {
  FutureGroup fg(ce); // Future group using ce as compound
  // Use futures and register them to group
  fg.synchronise(); // Excplicit synchronisation
  ce.reduce(); // Reduces and (possibly) throws
}
catch (MyException1& e) { // Error handling
  continue; // Deal with next exception, if any
  // or: break; // Continue, abandon the rest of exceptions
  // or: throw; // Abandon rest of exceptions, rethrow
}
```

6 Restrictions in the KC++ exception model

KC++ exception handling is implemented using standard C++ and without modifying the C++ exception model (and compiler) in any way. This means that KC++ exceptions have to cope with the restrictions caused by the C++ exception model, which has been designed without thinking about issues caused by concurrency.

One problem with reduction functions and multiple exceptions is that C++ does not allow replacing a thrown exception before it has been caught in a catch clause. This means that reduction functions can freely choose a suitable exception only if reduction happens prior to throwing any exceptions. KC++ futures do this automatically by synchronising with their future group before throwing exceptions. For other situations KC++ provides a macro mechanism to catch an already thrown exception and perform necessary reduction.

Standard C++ has a library function `std::uncaught_exception` [8, § 15.5.3] which is intended to be used in destructors to find out whether exception handling is in progress. However, it is widely known that this function is not as useful as it could be, and many books recommend to avoid it in most cases [12, 13]. KC++ exception handling uses this function in certain rare cases to find out whether exception handling is already in progress. Therefore special care should be taken if KC++ exceptions are used in destructors or code called from destructors.

7 Related work

Problems with asynchrony described in this article have been solved differently in different programming languages. This section lists some languages and how the combination of exception handling and concurrency has been implemented in these languages.

The strategy used in the Ada language [14] is quite common in other languages too. In Ada new tasks (threads) can be created, and the new tasks start executing asynchronously with their creator. Problems with asynchronous exceptions have been avoided by declaring that if an exception tries to leave the task body (i.e. if the task does not handle the exception locally), the exception is not propagated further and the task in question is simply terminated. Communication between tasks happens using the *rendez-vous* mechanism, in which one tasks calls a service on another task, which explicitly accepts the call. If an exception occurs during the rendez-vous, the exception is propagated from the accepting task to the calling task. However, this causes no problems because normally rendez-vous is a synchronous operation, so the calling task is waiting for the call to complete.

Ada-95 also defines an *asynchronous* accept statement. However, this is not an asynchronous call, but rather a way to execute a sequence of statements while waiting for a rendez-vous call to be accepted. When acceptance happens, the normal execution of the statements is immediately aborted. This mechanism does not suffer from the exception problems of asynchronous calls, but introduces its own problems, for example because the abortion just mentioned can occur in the middle of exception handling.

The Argus language implements asynchronous calls using *call-streams* and uses a very future-like construct called a *promise* to handle return values from asynchronous calls [15]. In Argus promises are strongly typed and represent the result of the asynchronous computation including possible exceptions. The type of the promise identifies the type of the return value and lists all possible exceptions which the promise may contain. Every asynchronous call returns a promise, which the caller can either poll periodically or start waiting for the call to complete ("*claim*" the promise). Waiting for the result of a promise either returns the normal return value of the call, or raises the exception the call has raised. If the same promise is claimed again, it re-returns the return value or re-raises the exception.

The Java language [16] has built-in concurrency. The exception handling strategy in the language is very close to the Ada approach. All exceptions are handled locally inside a thread, and they are lost if the thread does not contain an appropriate exception handler. All Java inter-thread communication happens either through shared objects or synchronous method calls (like RMI, Remote Method Invocation), so it involves no asynchrony.

Even though Java does not provide asynchronous method calls by default, there are several extensions to the language for this purpose. One of these is Java ARMI (Asynchronous Remote Method Invocation) [17] which is built on normal synchronous Java RMI. ARMI uses futures to represent results of asyn-

chronous calls. For exception handling, ARMI provides two alternative methods. The Delayed Delivery Mechanisms (DDM) embeds the possible exceptions inside futures like in Argus. The exceptions are thrown from the future when the return value of the asynchronous call is requested. The second choice called the Callback Mechanism (CM) allows the programmer to attach special exception handlers to the future. Each exception handler is capable of handling a specific type of exception. If that exception occurs, ARMI automatically calls the appropriate attached exception handler.

Another asynchronous extension to Java is JR [18]. JR implements asynchronous calls via *send* and *forward* statements. Exceptions are handled by requiring that each *send* and *forward* statement also specifies a *handler object*. Handler objects must implement the `Handler` interface and provide a method for each possible exception type, with the exception object as a parameter. When an exception occurs in an asynchronous call, an appropriate method in the handler object is called. The handler methods cannot throw any additional exceptions and they have to be able to completely handle the exceptional situation. The JR compiler statically checks that the handler object is able to handle all possible exceptions the call may throw. [19]

In the programming language Arche [20] asynchrony and synchronisation have been implemented quite differently. Every object in the language has its own thread of control, and every method call is synchronous. However, concurrent objects can communicate and co-operate asynchronously using *multimethods* (invocation of a method in a group of objects). Exception handling problems are solved by attaching to each multimethod a *coordinator*, which controls the overall action. The coordinator may have a *resolution function* which receives exceptions from all participants and computes a *concerted exception* representing the resulting "total" exception. [21]

8 Conclusion and future work

This article has shown how support for multiple exceptions under asynchronous concurrent calls can be added to C++. This is down with futures, future groups, compound exceptions and reduction functions. All this can be achieved using a library-based approach without changing the C++ syntax or the underlying compiler.

The mechanisms described in this article have been implemented in KC++, a concurrent active object based system implemented on top of C++. Optimisations and performance measurements of reduction function based exception handling are still in progress.

References

1. Goodenough, J.B.: Exception handling: issues and a proposed notation. Communications of the ACM **18** (1975) 683–696

2. Buhr, P.A., Mok, W.Y.R.: Advanced exception handling mechanisms. IEEE Transactions on Software Engineering **26** (2000) 820–836
3. Romanovsky, A., Kienzle, J.: Action-oriented exception handling in cooperative and competitive object-oriented systems. [22] pp. 147–164
4. Rintala, M.: KC++ — a concurrent C++ programming system. Licentiate thesis, Tampere University of Technology (2000)
5. Halstead, R.H.: Multilisp: A language for concurrent symbolic computation. ACM Transactions on Programming Languages and Systems **7** (1985) 501–538
6. Caromel, D., Belloncle, F., Roudier, Y.: C++//. In Wilson, G., Lu, P., eds.: Parallel Programming Using C++. MIT Press, Cambridge (MA), USA (1996) 257–296
7. Thompson, P.: IOUs: A future implementation. C++ Report (1998) 29–32
8. ISO/IEC: International Standard 14882 – Programming Languages – C++, Second Edition. (2003)
9. Andrews, G.R.: Concurrent Programming — Principles and Practice. Addison-Wesley, Reading, Massachusetts (1991)
10. Chatterjee, A.: Futures: a mechanism for concurrency among objects. In: Proceedings of the 1989 ACM/IEEE conference on Supercomputing, ACM/IEEE, ACM Press (1989) 562–567
11. Stroustrup, B.: The C++ Programming Language (Special Edition). Addison-Wesley, Reading, Massachusetts (2000)
12. Sutter, H.: More Exceptional C++. C++ In-Depth Series. Addison-Wesley (2001)
13. Henney, K.: From mechanism to method: The safe stacking of cats. C/C++ Users Journal Experts Forum (2002) `http://www.cuj.com/documents/s=7986/cujcexp2002henney/`.
14. Intermetrics, Inc.: Ada 95 Reference Manual. (1995)
15. Liskov, B., Shrira, L.: Promises: Linguistic support for efficient asynchronous procedure calls in distributed systems. In: Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation. (1988) 260–267
16. Arnold, K., Gosling, J.: The Java Programming Language. Addison-Wesley, Reading, Massachusetts (1998)
17. Raje, R.R., Williams, J., Boyles, M.: An asynchronous remote method invocation (ARMI) mechanism for Java. Concurrency: Practice and Experience **9** (1997) 1207–1211
18. Keen, A.W., Ge, T., Maris, J.T., Olsson, R.A.: JR: flexible distributed programming in an extended Java. In: Proceedings of the 21st IEEE International Conference on Distributed Computing Systems, IEEE (2001) 575–584
19. Keen, A.W., Olsson, R.A.: Exception during asynchronous method invocation. In: Euro-Par 2002: Parallel Processing. Volume 2400 of LNCS (Lecture Notes in Computer Science)., Springer-Verlag (2002) 656–660
20. Benveniste, M., Issarny, V.: Concurrent programming notations in the object-oriented language Arche. Research Report 1822, INRIA, Rennes, France (1992)
21. Issarny, V.: Concurrent exception handling. [22] pp. 111–127
22. Romanovsky, A., Dony, C., Knudsen, J.L., Tripathi, A., eds.: Advances in Exception Handling Techniques. 2022 of LNCS (Lecture Notes in Computer Science). Springer-Verlag (2001)

Two Techniques for Improving the Performance of Exception Handling

Michael J. Zastre and R. Nigel Horspool
{zastre,nigelh}@cs.uvic.ca

Department of Computer Science, University of Victoria, P.O. Box 3055 STN CSC, Victoria, B.C., V8W 3P6, Canada

Abstract. Two new optimizations using whole-program analysis are presented for languages using `try-catch` exceptions clauses (such as Java and C#) – *farhandlers* reduce the time needed to locate the handler for a thrown exception, and *throwsite reduction* eliminates unnecessary run-time overhead at throw instructions. Also presented are experimental results from a Java VM modified to implement these optimizations.

1 Introduction

Exceptions and their handling occupy an odd place amongst the set of programming-language constructs in languages such as Java. Exceptions are absolutely necessary in that Java's API simply requires their use in many cases, the most notable being for I/O. At the same time, however, exceptions are deemed to be rather expensive, and so in dispensing practical advice regarding performance Java programmers are encouraged to avoid them if at all possible in their own code [Shir00]. The advice seems reasonable especially given the run-time expense of exceptions – for instance, with many implementations of the Java VM the time taken to throw and catch a `NullPointerException` when dereferencing a null pointer is thousands of times more expensive than explicitly comparing an object reference with the value `null` [Zast05]. Nor can programming-language implementors be blamed for this run-time expense as they have been led to believe that exceptions are rare, and given the complexity of compilers, interpreters and VMs, they (rightly) choose to concentrate on what they consider to be more profitable optimizations. Indeed they often follow the advice given by designers of the programming language, a rare exemplar of which is shown here in text taken from the Modula-3 report [Card89, p. 17]:

> Implementations [of Modula-3] should speed up normal outcomes at the expense of exceptions (except for the return-exception and exit-exception). Expending ten thousand instructions per exception raised to save one instruction per procedural call would be defensible.

The result for most languages appears to be something of a classic "catch-22": Before exception implementations become faster there needs to be more programmers using exceptions, yet programmers tend to avoid exceptions because

of their negative impact on program performance. Therefore there exists a danger that the innovative uses of exceptions in the structuring of programs may not get much purchase amongst practitioners if the run-time costs are deemed to negate the gains to program clarity and expressibility.

Our contribution to breaking out of the "catch-22" is to propose two optimizations that reduce the run-time cost of exceptions. These optimizations do not require programmers to write their code any differently than they would when using exceptions, nor are programmers expected to supply special annotations or pragmas. The optimizations are also applicable when throw and handler sites are located within different methods. We introduce the first of these, called *farhandlers* in the next section. After that we introduce the second technique, called *throwsite reduction*. Next appears some experimental results obtained from a modified Java VM, and this is followed by a brief description of related work.

2 Farhandlers

Consider the callgraph in Figure 1. If the site calling a method is enclosed within an exception handler, then the edge corresponding to that method call is labelled. For example, there is a call to method `b` within method `a`, and this call occurs within a handler for exceptions of class `E`. In this particular callgraph there appear four different handlers for exceptions `E`; handlers are numbered within parentheses. (Note: This numbering is not programmatic – that is, a programmer does not provide this numbering.) The question posed by the diagram is: If an exception of class `E` is thrown in method `m`, which of the four handlers will catch it?

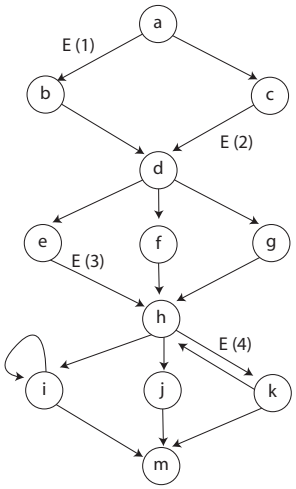


Fig. 1. Callgraph example

The answer depends, of course, on the path through the graph from node `a` to node `m`, and here we assume that there is no handler for `E` local to `m`. In practice most exception-handling mechanisms *unwind the stack*: a local handler for `E` in `m` is sought at run-time, and if one is not found, the search continues within the method that called `m`, with the search proceeding through the callstack until a local handler is found.

The goal of a *farhandler table* is to reduce both the number of handler-table lookups and the number of times stack-unwinding occurs by taking into account information about the callpath available on the runtime stack. For example, if an exception of class `E` is thrown on an invocation of `m`, then we can examine the return address stored for that invocation. If this return address indicates a call from either `i` or `j`, then unwinding to the first invocation of `h` is needed to help determine the location of a handler. However, if the return address indicates a call from `k`, then the handler location is known (that is, we unwind to the most recent invocation of `h` and transfer control to the first instruction in the handler `E(4)`). Therefore even if an exception-handler table exists in nodes `i` or `j` (that is, for other exception classes), we do not examine the tables; as we need not check for such tables, we can unwind the stack faster than if we had to examine the tables.

Extending the example somewhat, consider the callpath `a, b, d, f, h, j, m`. The callgraph indicates that if such a path has been followed at runtime, handler `E(1)` would be the appropriate handler when an instance of `E` is thrown in `m`. At runtime, however, we would normally be able to piece together the callpath only by examining the call stack one frame at a time. (The implication here is that we choose not to keep a copy of the callpath separate from the call stack itself as that would introduce possibly unnecessary work at each method invocation.) That being the case, what is the smallest number of distinct unwind/lookup steps needed to locate the handler? Here the answer is "3":

1. At the invocation of `m`, the return address stored indicates that `m` was called from `j`; therefore we unwind to `h`.
2. Now that we are in `h`'s context, the return address stored here indicates that this invocation of `h` is the result of a call from `f`; therefore we can unwind to `d`.
3. In `d`'s context, we discover from the return address stored here that `d` was called from `b`, and that `b` itself was called from `a` and this from within a handler to `E` (handler 1 from the callgraph). Therefore we can unwind to `a` and then transfer control to the first instruction in `E(1)`.

To implement this new form of handler lookup we introduce a *farhandler table*; the name is meant to suggest an aid for finding non-local handlers. Such a table for the example callgraph appears in Table 1. Here are a few items to observe about this table:

– Each row in this table has an edge in the callgraph (*i.e.,* one-to-one mapping from edges to row).

– At exception-throw time, the current stackframe context (*i.e.,* the node) and the return address stored in the context are used to find a single row in the table.
– Each row in the table has either an entry in the *Dispatch Address* column or in the *Handler Info* column, but not both.

Entries in *Dispatch Address* indicate both an unwind point *and* an address to which control flow is transferred. For example, `PC(E1)` denotes the first location in handler `E(1)`. No further lookups or unwinding are necessary after using information in *Dispatch Address*. If the location of a handler is not yet known, then *Handler Info* is used to indicate the point to which the stack must be unwound. For example, the first step given the program path above would result in a lookup of the second-to-last row, and this contains $\mathcal{H}(\mathtt{h})$ – therefore the runtime stack is unwound to the earliest instance of `h` and handler lookup continues in `h`'s context.

Node	Return Address	Dispatch Address	Handler Info
a	—	—	$\mathcal{H}$(a)
b	PC(a.b())+4	a:PC(E1)	—
c	PC(a.c())+4	—	$\mathcal{H}$(c)
d	PC(b.d())+4	a:PC(E1)	—
d	PC(c.d())+4	c:PC(E2)	—
e	PC(d.e())+4	—	$\mathcal{H}$(d)
f	PC(d.f())+4	—	$\mathcal{H}$(d)
g	PC(d.g())+4	—	$\mathcal{H}$(d)
h	PC(e.h())+4	e:PC(E3)	—
h	PC(f.h())+4	—	$\mathcal{H}$(d)
h	PC(g.h())+4	—	$\mathcal{H}$(d)
h	PC(k.h())+4	h:PC(E4)	—
i	PC(i.i())+4	—	$\mathcal{H}$(h)
i	PC(i.h())+4	—	$\mathcal{H}$(h)
j	PC(h.j())+4	—	$\mathcal{H}$(h)
k	PC(h.k())+4	h:PC(E4)	—
m	PC(i.m())+4	—	$\mathcal{H}$(h)
m	PC(j.m())+4	—	$\mathcal{H}$(h)
m	PC(k.m())+4	—	$\mathcal{H}$(k)

Table 1. Program-wide handler table for callgraph example

Computing these tables is relatively straightforward and consists of three separate steps:

1. For each node, a set of reachable handlers for a given exception class is computed; set items are denoted by a triple of the form $\langle n_s, \Sigma, n_d \rangle$, where n_s is a calling (or *source*) node, n_d the called (or *destination*) node, and Σ the exception handler enclosing the call of m to n. (*i.e.,* these are representations of labelled edges in our callgraph). The sets of reachable handlers from a node such as `m`, *i.e.,* those later in the call sequence, are larger than a node such as `d`, *i.e.,* those earlier in the call sequence.

2. We then identify those nodes where control-flow paths merge together. These nodes are called *mergehandlers*. In our callgraph example, such nodes are `d`, `h` and `m`; the root node of a callgraph is considered to be a trivial merge node (*i.e.,* `a` in our example). All other nodes can be reached from only one other node. Each node in the callgraph will have associated with it a mergehandler, which is either the node itself (if the node is a mergehandler) or the first mergehandler encountered when traversing up the callgraph (*i.e.,* towards the root node).
3. With the previous two items we can now construct the farhandler table row by row. Each callgraph edge corresponds to a table row. The first field in the row is the edge's destination node. The second field is the return address corresponding to the callsite (*i.e.,* next instruction following the callsite in the edge's source node). If there is exactly one item in set computed for the destination node in (1) above, then the corresponding handler is used to fill in the third field. Otherwise the third field remains blank and the fourth field is filled with the mergehandler for that node.

Farhandler tables can be extended to deal with more than one exception type by adding extra columns to the table and repeating the analysis for each additional exception type.

There has been no mention of `finally` clauses so far. Previous work has shown that these clauses are rare in practice [Ryde00]. We nevertheless have described one way of dealing with such clauses in [Zast05].

3 Throwsite Reduction

The actual run-time cost of throwing and catching an exception can be broken down into approximately four separate categories:

- the effort required to allocate an exception object on the heap;
- the construction of a stack trace;
- the cost of unwinding the stack; and
- other activities such as handler table lookup, actions of the VM, garbage collection during unwinding, etc.

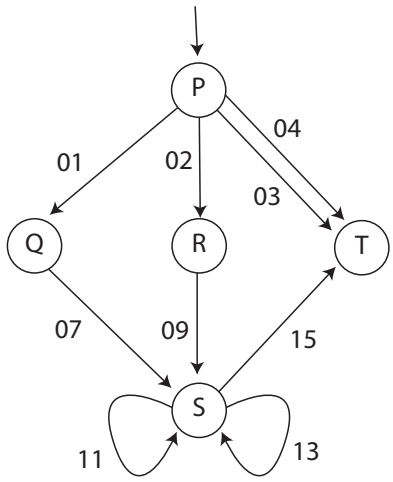
These activities can often take surprisingly long periods of time, and one or two of these consume large proportions of the overall effort needed for throwing and catching [Zast05]. For example, stack-trace construction on the Java Classic VM (version 1.4.2) makes up at least 50% of the effort, and the deeper the callstack the longer the time required (*i.e.,* the complete stacktrace comprises information starting at program's `main` entry point and leading down to the throwing method itself).

What is somewhat more shocking is that there is no requirement stating a handler must use the exception object, nor need the handler even reference the stacktrace itself. Many handlers in fact do not use the exception object to transfer information from the throwsite to the handler, and instead depend

upon exceptions as a form of control-flow transfer. Therefore we propose an optimization called *throwsite reduction*, implying that the work performed at the throwsite (such as exception-object creation and stacktrace construction) is eliminated if certain facts are true about handlers reachable from the throwsite.

```
   P() {
01   try { Q();
02         try { R(); }
03         catch (F f) { use; T(); }
04         T();
     }
05    catch(E e) { discard }
06    catch(F f) { discard }
   }

   Q() {
07   try { S(); }
08   catch (F f) { discard }
   }

   R() {
09   S();
   }

   S() {
10   switch (condition) {
11     case 1: try { S(); }
12             catch(F f) { use }
               break;
13     case 2: try { S(); }
14             catch(F f) { discard }
               break;
15     case 3: T();
               break;
16     default: throw new E();
     }
   }

   T() {
17   throw new F();
   }
```

(a) code



(b) call graph

Fig. 2. Code and callgraph

Consider the code appearing in Figure 2 and the corresponding callgraph appearing to the right of the code. The labels on callgraph edges now correspond to line numbers in the code example (*i.e.,* the edge from `R` to `S` corresponds to line `09` where `R()` makes a call to `S()`). Nearly every call is enclosed by some `try` block. The start of handler blocks are notated with either the word `use` or `discard`; this indicates whether or not the exception object is used by the handler. The questions raised by this code are: Does an exception object need to be created at the throwsite on line 16? or on line 17?

If all handlers that can catch an exception thrown at line 16 do not use the object, then we can annotate the throwsite at line 16 as being eligible for *tsr* (throwsite reduction). When the bytecode corresponding to line 16 is evaluated at runtime, a virtual machine could check if a *tsr* annotation exists for this `athrow` bytecode, and if this does exist then a dummy placeholder object is pushed onto the virtual machine's operand stack.

Code inspection in this case reveals there exists *no* handler in the example which would use an exception objected created at line 16. The same cannot be said, however, for the instance of `F` thrown at line 17: As `T` is reachable from `S` – via a call to `S` in line 11 followed by a call to `T` in line 15 – and as the handler at line 12 uses the exception object, we cannot mark line 17 as *tsr*.

The computations for determining *tsr* annotations uses some of the intermediate values prepared for farhandler tables (*i.e.,* the set of reachable handlers for each node in step 1). We need add only one extra bit of information about each handler – that is, whether or not the exception object is used by the handler – and use the following for each throwsite:

1. If a local handler around the throwsite for the exception class exists, then check if the handler uses the exception object. If so, then the throwsite cannot be annotated *tsr* and we proceed to examine the next throwsite.
2. Otherwise we examine all of the handlers for the exception class reachable from the throwsite. If no handler uses the exception object, then the throwsite is annotated as *tsr* and we proceed to the next throwsite.
3. Otherwise the throwsite is left unannotated.

The usefulness of the *tsr* annotations, not to mention that of handler information in farhandlers, is directly affected by the analysis used to build the callgraph. In the following section we present experimental results where callgraphs were built using a *flow-insensitive* analysis, and we can do better than this but only if we are willing to pay the extra cost at compile time to perform a flow-sensitive analysis. The analyses now available for object-oriented programming languages (such as Class Hierarachy Analysis [Dean95]) can provide an even greater levels of precision (*i.e.,* fewer callgraph edges).

4 Experimental Results

We implemented both farhandlers and throwsite reduction using an analyzer based on the `Soot` bytecode-manipulation framework [Vale99] and modified an existing Java virtual machine called `SableVM` [Gagn01]. Timings provided in the next two subsections were produced on a Pentium 3 running at 750 MHz and 128 MB RAM running RedHat Linux 7.2. Our experiments were in two groups: the first was our *validation* group in which we used the SPECjvm98 benchmarks, and the other *exception-idiom usage* group in which a standard algorithm was converted into an exception-handling style of code.

4.1 Validation group

As with medical doctors who must administer treatment to patients, our modification of the VM should follow the rule from medicine of *primum non nocere*—"first of all, do no harm." Therefore our modified VM should not produce poorer performance for programs, regardless of whether or not they use exceptions. The SPECjvm8 benchmark suite allows us to check for this; the version of we used is maintenance release 1.04, but with two omissions:

- `_227_mtrt` (a ray-tracing program) raises a `ClassCastException` that causes program failure when run with either the unmodified VM or the modified VM. (The same error occurs when using the HotSpot VM from Sun.)
- `_213_javac` (Sun's Java compiler from JDK 1.0.2) causes our `Soot`-based analyzer to fail from an `OutOfMemoryException` (one of life's little ironies!) and therefore no farhandler tables or *tsr* annotations can be generated.

Of the SPECjvm98 benchmark programs tested here, only `_228_jack` makes significant use of exceptions, and even then its programmers appear to have taken special care to eliminate a lot of exception-handling overhead (*i.e.,* the thrown exceptions are previously created objects, with the object creation cost amortized over the many throws which use it).

Each benchmark was run on four different VM configurations:

- original: This is the unmodified SableVM;
- fh: modified VM using only the farhandler table;
- tsr: modified VM using only the *tsr* annotations;
- fh+tsr: modified VM using both the farhandler table and the *tsr* annotations.

The timings are shown in Table 2. Only `_200_check` and `_228_jack` throw any exceptions at all; the former throws 104 exceptions caught by local handlers, while the latter throws 241,876 exceptions caught by non-local handlers. What the results show is that the benchmarks run as fast—if not faster—under the modified VM as they do under the unmodified VM.

We make two general observations about this data:

1. Only `_228_jack` throws a significant number of exceptions—all of them to handlers outside of the throwing method—and the benchmark's speed is improved (about 1% on average, with 0.7% in the worst case and 1.2% in the best case). This gain is significant considering that much other computation is being performed by the benchmark program.
2. For all of the other benchmark programs, there is no observable difference (*i.e.,* "no harm").

4.2 Exception-Idiom Usage group

As a test of the effect of our two optimizations on a practical problem, we have chosen one for which a file of words must be examined, and a histogram

Benchmark	original	fh	tsr	fh+tsr
_200_check	49	49	50	49
_201_compress	111635	111418	111475	111564
_202_jess	122402	122380	121380	122362
_209_db	212402	210672	209706	211114
_222_mpegaudio	505410	504821	504667	504856
_228_jack	68933	68131	68270	68432

Table 2. SPECjvm98 benchmark timings (milliseconds)

of those words produced (as might be needed by a compression algorithm, for instance). As each word is input, a binary tree is searched. If the word is found, the corresponding tree node's `frequency` field is incremented. If the word is not found, then a new node must be created and linked into the existing tree.

Several versions of the program were written:

- `SearchLocal` uses exceptions to transfer control to node-creation code when a word is first encountered; all searching of the tree and node creation occurs within the same method.
- `SearchNonLocal` also uses exceptions as mentioned above, but now the tree is searched recursively. Control-transfer for new words now entails unwinding the stack.
- `SearchLocalX` and `SearchNonLocalX` do not use exceptions and perform searching local and via recursive calls, respectively. These should be the fastest versions of the programs.

Text files from the *Calgary Compression Corpus* provided the workload for various programs [Bell90]. Timing results for for `SearchNonLocal` are in Table 4. Each of the individual tests in the corpus corresponds to a table row. The column labelled "w/o exceptions" is the time (in milliseconds) taken by the unmodified VM to process the test file with an algorithm that *does not* use exceptions. There follow two pairs of columns: the first pair is for a version of the VM not using the optimizations described here, while the VM of the second pair *does* support both farhandlers and throwsite reduction. Each of the "exception cost" columns represents the contribution made by exceptions to processing a file (*e.g.,* when computing the histogram for words in `bib`, exception-handling in the original VM results in a program running 353% longer than the version of the program without exceptions, while in the modified VM exception-handling the program runs only 0.3% longer).

The modified VM is clearly a win. The overhead of using exceptions (*i.e.,* the time difference between an exception-free program and exception-rich one) is low, ranging from .3% to 1.1% for `SearchNonLocal`. A pleasant surprise from `SearchLocal` (results not shown here) is that in some cases there is a *speedup* as in that for `book1` of about 0.6% [Zast05].

file	w/o exceptions	*Using original VM*		*Using modified VM, fh + tsr*	
		w/ exceptions	exception cost	w/ exceptions	exception cost
bib	2,445	11,081	353%	2,453	0.3%
book1	20,661	73,255	254%	20,731	0.3%
book2	15,240	48,962	218%	15,338	0.6%
geo	819	2,279	178%	821	0.2%
news	8,645	42,262	388%	8,725	0.9%
obj1	223	1,095	391%	226	1.3%
obj2	2,737	14,949	447%	2,762	0.9%
paper1	1,198	5,895	392%	1,210	1.0%
paper2	1,941	8,269	326%	1,958	0.9%
paper3	1,028	5,922	476%	1,049	2.0%
paper4	276	1,596	478%	279	1.1%
paper5	266	1,573	491%	269	1.1%
paper6	916	4,167	354%	921	0.5%
progc	823	4,160	405%	828	0.6%
progl	1,426	5,281	270%	1,434	0.6%
progp	805	3,644	353%	809	0.5%
trans	1,647	6,713	307%	1,655	0.5%

Table 3. SearchNonLocal timings (milliseconds, 10,000 iterations)

5 Related Work

Some optimizations for improving exception-handling performance are applied to cases where the throwsite and its handler are within the same method, as in the `LaTTe` system [Lee99]. If some method inlining is acceptable, then *Exception-Directed Optimization* may be suitable; paths through a callgraph are profiled, and those paths with a high execution frequency are inlined into one large method and local optimizations then applied [Ogas01].

A stack-unwinding optimization was proposed by Drew *et al.* [Drew95] in which as little state as possible is restored when moving from a frame to its calling procedure's frame. State is instead restored incrementally, *i.e.,* only when a handler is found is the complete state of a procedure's context restored.

6 A word about Just-In-Time compilation

The results described in the previous section were obtained using a VM supporting only interpreted bytecode. Much recent work on improving the run-time performance of languages such as Java and C# have focused on *Just-In-Time* compilers, *i.e.,* where individual methods are compiled such that they run at the speed of the underlying machine's native code (or more precisely, the interpreted VM may invoke both bytecode and native-code version of methods). One assumption therefore may be that the cost of exceptions can be eliminated by a JITter without any extra analysis or algorithms.

Unfortunately this assumption is woefully inaccurate. Without extra analysis, a JIT may lead a programmer to believe that exception-handling has an even *higher* cost relative to code written without exceptions. This is due to the way in which a language's runtime deals with exceptions, *i.e.,* non-local exceptions

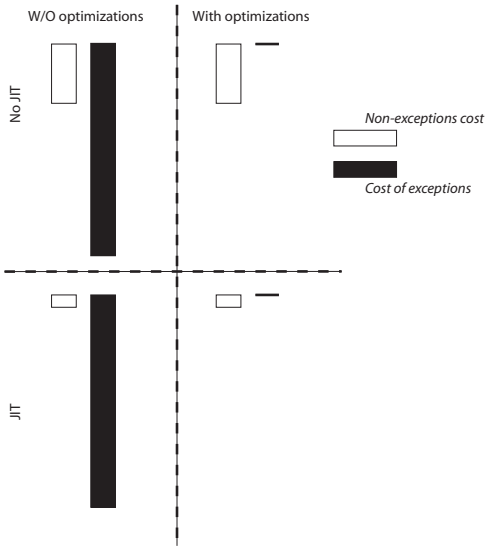


Fig. 3. Interaction of JIT-compilation with optimizations

invoke code within the interpreted VM for transferring control from the throwsite to the handler. One way of visualizing this is shown in Figure 3; each of the quadrants represents the contribution to the overall cost of the execution (in this case the processing of `bib` from the Calgary Compression Corpus in the experiments just described). If a JIT produces a modest five-fold increase in execution speed for JIT-compiled methods and exceptions are still dealt with as previously, then what the programmer experiences appears in the left side of the diagram – the relative contribution of exceptions relative to the "unexceptional" code – appears much larger even though the absolute contribution is unchanged.

We argue that our optimizations are even *more* important for a JIT compiler than for a purely interpreted VM. This can be seen in the right-hand side of Figure 3 where the exception cost is now very small, such that the five-fold increase in performance offered by the JIT is – for all intents and purposes – achieved in the presence of exception handling.

7 Conclusion

We have presented two new optimizations which are designed to improve the performance of exception handling. Our main goal, however, is to make the use exceptions more attractive to practioners such that they need not be concerned about the runtime cost (or at least not unduly concerned). We have shown that in cases where exceptions are used to express control flow, nearly all the overhead of exceptions can be eliminated. Extensions to these techniques to cover a larger set of cases, such as those where some handlers use the stack trace but not all do (*lazy*

stacktrace construction), or where some handlers use the exception object but not all do (*lazy exception-object creation*). Our current analyses require access to the whole program, but a more incremental approach towards farhandler-table construction (and *tsr* annotations) would combine well with JIT technology (*i.e.,* perform analysis as classes are loaded into the VM). However, much work is still needed to improve the performance of code in the presence of exceptions, specifically analyses to mitigate the negative impact of exceptions on traditional compiler optimizations (for example, Factored Control-Flow Graphs [Choi00]. There also remains the hard work of convincing programmers that exceptions can improve the readability and maintainability of programs, and perhaps this can be achieved via the identification of useful *exception idioms* or *exception patterns*.

References

[Card89] Cardelli, L., Donahue, J., Glassman, L., Jordan, M., Kaslow, B. and Nelson, G.: Modula-3 Report (revised), Digital Systems Research Centre. 1989.

[Choi00] Choi, J., Grove, D., Hind, M., and Sarkar, V.: Efficient and Precise Modelling of Exceptions for the Analysis of Java Programs, in *Proceedings of the SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '99)*, pp. 21–31.

[Dean95] Dean, J., Grove, D. and Chambers, C.: Optimization of Object-Oriented Programs using Static Class Hierarchy Analysis, in *Proceedings of the European Conference on Object-Oriented Programming '95*, pp. 77–101, Springer-Verlag. 1995.

[Drew95] Drew, S., Gough, K. and Ledermann, J.: Implementing Zero Overhead Exception Handling. Technical Report FIT 95-12, Queensland University of Technology. 1995.

[Gagn01] Gagnon, E. and Hendren, L.: SableVM: A Research Framework for the Efficient Execution of Java Bytecode, in *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '01')*, pp. 27–40, ACM Press. April 2001.

[Lee99] Lee, S., Yang, B., Kim, S., Park, S., Moon, S., Ebcioğlu, K., and Altman, E.: On-Demand Translation of Java Exception Handlers in the LaTTe JVM Just-In-Time Compiler. In *Proceedings of the Workshop on Binary Translation*. October 1999.

[Ogas01] Ogasawara, T., Komatsu, H., and Nakatani, T.: A study of exception handling and its dynamic optimzation in Java. In *Proceedings of the OOPSLA '01 Conference on Object-Oriented Programming Systems, Languages and Applications*, pp. 83–95, ACM Press. October 2001.

[Ryde00] Ryder, B., Smith, D., Kremer, U., Gordon, M., and Shah., N.: A Static Study of Exceptions Using JESP. In *Proceedings of Compiler Construction 2000*, pp. 67–81. April 2000.

[Shir00] Shirazi, J.: Java Performance Tuning, O'Reilly and Associates, Inc. 2000.

[Vale99] Vallée-Rai, R., Hendren, L., Sundaresan, V., Lam, P., Gagon, E., and Co, P.: Soot – A Java Optimization Framework, in *CASCON 1999*, pp. 125–135. September 1999.

[Bell90] Bell, T., Cleary, J., and Witten, I.: Text Compression, Prentice-Hall, Inc. 1990.

[Zast05] Zastre, M.: The Case for Exception Handling. PhD Thesis, University of Victoria (2004).

CAMA: Structured Coordination Space and Exception Propagation Mechanism for Mobile Agents

Alexei Iliasov, Alexander Romanovsky

School of Computing Science, University of Newcastle
Newcastle upon Tyne, NE1 7RU, United Kingdom

Abstract. Exception handling has been proven to be the most general fault tolerance technique as it allows effective application-specific recovery. If exception handling is to make programmer's work more productive and less error-prone, however, it requires adequate support from the programming and execution environments. Scoping is a dynamic structuring technique which makes it easier for the developers to deal with the complexity of system execution by narrowing down the context visible for the individual system components. In this work we are specifically interested in scoping that supports error confinement and allows system error recovery to be limited to the area surrounding the error. The approach we propose aims at assisting in rigorous development of structured multi-level fault tolerant agent systems.

1 Introduction

Intrinsic virtues of mobile agents such as mobility, loose coupling and ability to deal with disconnections can make them look promising for structuring large-scale distributed systems. Yet agents have to face all kinds of communication media failures as well as failures of software in their fellow agents and, of course, internal failures. System openness brings even more concerns, such as interoperability, security and trustworthiness. The types of mobile agent system failure can be roughly grouped into the following categories:

1. *failure to deliver service by the hosting environment;*
2. *failure in one of the collaborating agents;*
3. *internal agent failure;*
4. *an environment failure.*

While a similar classification is discussed in [2], in this paper we are introducing different categories of faults in order to focus on the interoperability issues more and to capture in a more practical and detailed way all possible kinds of the environment failure.

Failures of the first category include all types of transient failures, such as disconnection, migration, spawning, inability to deliver messages, etc. Such failures may be caused by changes in the environment, for example those due to

migration, and are better handled by application logic. It has often been said that recovery actions for such situations must be developed at the initial stages of agent design. And, unlike traditional software, mobile agents have, thanks to mobility and code migration, a whole new kind of recovery possibilities.

The second category consists of failures of a very interesting kind. One of the appealing features of mobile agents is dynamic composition. Agents do not have to know what other agents they will cooperate with, and this allows extreme flexibility in agent system design. In open systems, where agents discover their partners dynamically and where each agent has its own interest in cooperation, there must be some mechanism to encourage communication among matching agents and prevent it among incompatible ones. In addition to the means of communication among agents, we also need means for inter-agent exception propagation and cooperative exception handling. We believe that this is essential for a disciplined and fault-tolerant composition of mobile agent systems.

The abnormal situations of the third category are detected inside an agent. All the traditional recovery techniques developed for sequential programming can be used to deal with them. If an agent fails to recover from a failure individually, then there is a need for cooperative exception handling by all the involved agents.

The last category of failures corresponds to exceptional situations in the environment that are beyond the control of a mobile agent. Examples of this are failures of hardware components, administrative restrictions, software bugs in the underlying middleware and in the core components of the environment.

All these failures are typical of the domain of mobile agent software and mobile agents usually cannot anticipate or avoid this kind of malfunctions. In this paper we are focusing on the second category of failures and propose two fault tolerance solutions. The first one is an exception handling technique for coordination space-based mobile agents. The second solution is a scoping mechanism for mobile coordination spaces. In our approach we combine these two solutions in one fault tolerance development method.

Exception handling has been proven [1] to be the most general fault tolerance technique as it allows effective application-specific recovery. If exception handling is to make programmer's work more productive and less error-prone, however, it requires adequate support from the programming and execution environments. Scoping is a dynamic structuring technique which makes it easier for the developers to deal with complexity of system execution by narrowing down the context visible for the individual system components. In this work we are specifically interested in scoping that supports error confinement and allows error recovery to be limited to the area surrounding the error.

2 Related Work

MobileSpaces [12] is a middleware with a hierarchical organisation of agents. The notion of agent nesting and the approach proposed to migration is similar to those used in the Ambient Calculus [13] algebra. An agent in MobileSpaces communicates only with its parents or descendants (nested agents). Whole branches

of the agent tree can migrate, changing their parent nodes. This approach presents quite a flexible form of isolation.

In Mole [9] inter-agent communication is based on the publish/subscribe model called Object Management Group (OMG). OMG introduces channels through which events can be propagated among agents. Channels can be created during run-time and it is the creating agent who decides to whom pass the channel reference. This encourages closed group work among several agents. However event channel is not an isolation mechanism since it can pass events to other channels and also receive external events interesting to the channel subscribers.

Paper [8] discusses an extension of the publish/subscribe scheme with the scope concept. Scopes can be nested and they regulate event propagation and publishing. Agents can create, join and leave scopes dynamically. The purpose of scopes in this model is to limit visibility of published events to a subset of agents from the same tree of scopes (all scopes in the system form a forest). Another important implication of the scope notion is the introduction of the administrator role. Administrator is a utility agent that controls event flow inside a scope and across its boundaries according to the rules statically defined for the scope.

A different approach is taken in ActorSpace [10, 11], where communication space is structured using actorSpace - an abstract agent container. Special entities called managers may control visibility of agents and actorSpaces with respect to some other actorSpace. Each agent has a set of patterns describing its interests. There are three basic ways of sending a message: using a pattern to non-deterministically describe a destination agent, using a unique agent name and a pattern-based broadcast which delivers messages to all the agents satisfying the specified pattern. In addition it possible to create arbitrary complex visibility structures by placing a reference to an actorSpace in another actorSpace.

Coordination with Scopes [14] discusses a scoping mechanism for Linda tuple space built in way similar to ActorSpace. However scope here is not a container but a viewpoint of an agent on otherwise flat tuple space. The most interesting aspect is possibility of dynamically create new scopes by using several predefined operations on already known scopes forming a kind of scope algebra. In addition to the obvious joining and nesting operations, scopes can be also intersected and subtracted. This gives extreme flexibility in structuring tuple space and adapting it to an agent needs. A dedicated scope initially known to all agent is used to exchange scope names.

3 Coordination with Scopes

3.1 CAMA Model

The CAMA (context-aware mobile agents) system consists of a set of *locations*. Active entities of the system are *agents*. CAMA agent (further agent) is a piece of software that conforms to some formal *specification*. Each agent is executed on its own *platform*. Platform provides execution environment and interface to

the location middleware. Agents communicate through the special construct of coordination space called *scope*. An agent can cooperate only with agents participating in the same set of scopes. Agents can logically and physically migrate from a location to a location. Migration from a platform to a platform is also possible using logical mobility. An agent is built on the base of one or more *roles*. Role is a formal functionality specification and composition of specifications of all the roles forms the specification of an agent. A role is the result of the decomposition of an abstract *scope model* and a *run-time scope* is an instantiation of such abstract model. After this point we will use term *scope* to refer to a runtime scope in coordination space. More details on building formal specification of roles using the B Method and general description of the CAMA system can be found in [3].

3.2 Scoping mechanism

After analysing a number of existing approaches to introducing structuring of mobile agent communication (see Section 2.1) we have found that the best way to do it for the purpose of dealing with complexity of the system behaviour during rigorous system development, and, in particular, with supporting behaviour and information hiding for fault tolerance is to *structure agent activity* (dynamic behaviour). This automatically introduces communication structuring however with a much cleaner semantics and a number of other benefits discussed below.

Structuring activity means arranging agents in groups according to their intentions and *afterwards* configuring the means of their communication to adapt to the requirements of the agent group. Reconfigurations happen automatically thus allowing agents (and developers) to focus solely on collaboration with other agents. The distinctive features of this approach are

- *higher-level abstraction of communication structuring;*
- *impossibility to create incorrect, malfunctioning or cyclic structures;*
- *strong relationship with interoperability and exception handling;*
- *simple semantics facilitating formal development.*

In a very basic view scope is a dynamic data container. It provides an *isolated* coordination space for *compatible* agents by restricting visibility of tuples contained in a scope to the participants of the scope. Concept of compatibility is based on the concepts of role and scope. A set of agent is compatible if there is a composition of their roles that forms an instance of an abstract scope model.

Agents may issue a request for a scope creation and, at some point, when all the precondition are satisfied, the scope is atomically instantiated by a hosting location. Scope has a number of attributes divided into categories of scope *requirements* and scope *state*. Scope requirements essentially define type of a scope, or, in other words, kind of activity supported by the scope. Scope requirements are derived from a formal model of a scope activity an together with agent roles form an instance of the abstract scope model. State attributes characterise a unique scope instance.

Requirements	State
- *list of roles*	- *currently enrolled roles*
- *restriction on roles*	- *owner*
	- *name*

In addition to the attributes, scope contains *data*, that in case of coordination space are *tuples*. Along with data there may be *subscopes* to match *nested activities* that may happen inside of a scope.

Restrictions on roles dictate how many agent roles there can be for any given role of a scope. Requirements are defined by two numbers - a minimum required number of agents for a given role and a maximum allowed number of agents for a given role. A scope state tracks the number of currently taken roles and determines whether the scope can be used for agent collaboration or not.

R_1	R_1^{min}	R_1^{max}
R_2	R_2^{min}	R_2^{max}
...		
R_k	R_k^{min}	R_k^{max}

$R_i^{min} \leq N_{R_i} \leq R_i^{max}$	(taken roles)
n	(scope name)
A	(owner)

Fig. 1. Scope requirements (left). Scope state (right)

In addition to obvious $R_i^{min} \leq R_i^{max}$ we also require that $R_i^{max} > 0$.

There are three important states of a scope. Their summary is given on Table 2. A scope in the *pending* state does not allow agents to communicate because

State name	Definition
pending	$\exists r \cdot (r \in R \wedge N_r < r^{min})$
expanding	$\forall r \cdot (r \in R \Rightarrow N_r \geq r^{min}) \wedge \exists r \cdot (r \in R \wedge N_r < r^{max})$
closed	$\forall r \cdot (r \in R \Rightarrow N_r = r^{max})$

Fig. 2. Three important states of a scope

there are some essential roles missing. When all the required roles are taken the scope becomes *expanding* or *closed*. In the expanding state a scope supports communication among agents while still allowing other agents to join the scope. In the closed state there are no free roles and no additional agent may join the scope.

Some scope configurations present interesting cases. A scope with zero required number of agents for all the roles is called *blackboard*. It persists even

without any participating agents and all the contained data also persist. With this scope type agents do not have to wait for any other agents to communicate, they may put some information into a blackboard scope and leave. Note, that there is an important difference between a blackboard scope and a generic tuple space. For a blackboard scope only agents implementing the roles specified by the blackboard scope requirements may enter and put or read any data whilst in a tuple space anyone can always put and read any tuples. *Container* is a scope

Scope class	Definition
blackboard	$\forall r \cdot (r \in R \wedge r^{min} = 0)$
container	$card(R) = 1 \wedge (r \in R \wedge r^{min} = r^{max} = 1)$
bag	$card(R) = 1 \wedge (r \in R \wedge r^{min} = 0 \wedge r^{max} = 1)$
unrestricted	$\exists r \cdot (r \in R \Rightarrow r^{max} = \infty)$

Fig. 3. Some interesting classes of scopes

with a single role for which only single agent is allowed and required. This is an important case since such kind of a scope can act as a private and protected data container of an agent. A variant of container scope that can survive change of owners without losing all the contents is called *bag*. Bags can be used to privately pass some bulk data between two agents.

Unrestricted scope permits an unlimited number of agents for one or more of its roles. It can be used for client-server models when there are no restrictions on number of clients.

In global view scopes form a tree. Due to the specifics of our approach the tree is mostly shallow and wide since the depth is determined by the nesting level of actions that is usually not large. All the high-level scopes are united by the dedicated scope λ. Any scopes other than λ are subscopes of λ. Scope λ has two predefined roles: role λ_A of agent requesting services from location and and role λ_L of location. Functionality of role λ_A is arbitrary and defined by agent developers. However there is a fixed set of operations called λ_L^0 that must be included into implementation of each λ_L. λ_L^0 operations are described below.

Since the CAMA system allows agent to communicate in several locations at the same time we include location name to disambiguate reference to a scope. Moreover scope names have to be unique only inside the containing scope. Thus the full name of a scope includes the names of all the containing scopes. We are omitting name of λ scope for convenience of notation. Sometimes we have to explicitly state in what location a scope is contained and what are its parents (the containing scopes). In this case location name is the initial part of a name, then follows the outermost containing scope and so on up to the name of the scope we deal with.

λ_L^0 operations:

- `engage(id)` - issue a new location-wide unique and unforgeable name for agent `id`. This name as agent identifier in all other role operations.
- `disengage(a)` - issued name `a` becomes invalid.
- `create(a, n, R)@s` (`n` $\notin$ `l.s`) - creates a new sub-scope of scope `s` with the name `n` and given scope requirements `R`. The created scope becomes a private scope of agent `n`.
- `destroy(a, n)@l.s` (`n` $\in$ `l.s` $\wedge$ `a` *is owner of* `l.s.n`) - destroys sub-scope with the name `n` contained in the scope `s`. This operation always succeeds if the requesting agent is the owner of the scope. If the scope is not in the pending state then all the scope participants receive `EDestroy` exception as the notification of the scope closure. This procedure is executed recursively for all the subscopes contained in the scope.
- `join(a, n, r)@s` (`n` $\in$ `l.s` $\wedge$ `r` $\in$ `n` $\wedge$ `n` *is pending or expanding*) - adds agent `a` into scope `n` contained in `l.s` with roles `r`. Succeeds if scope `l.s.n` exists and it is possible to take the specified roles in the scope. This operation may cause state change of the scope.
- `leave(a, n, r)@s` (`a` *is in* `l.s.n` *with role(s)* `r`) - removes agent roles `r` from scope `l.s.n`. The calling agent must be already participating in the scope. This operation may also change the state of the scope.
- `put(a, n)@s`: advertises scope `n` contained in scope `s` thus making it a public scope. A public scope is visible and accessible by other agents. contained in scope `l.s` and supporting role(s) `r`.
- `get(a, r)@s`: enquires names of the scopes contained in scope `l.s` and supporting role(s) `r`.
- `handshake(a, t)@s`: allows agents to safely exchange their names.

3.3 Naming issues

In the following discussion we discuss certain assumptions on how names of various resources are used and passed between agents. One essential requirement is that a scope name can be known to an agent only if the agent joins the parent of the scope. A scope name passed as a message between two agents may violate this rule so we have to take special care of the names used by agents. However we still allow agents to learn names of other agents, scopes, locations and traps (discussed further below) through communication with other agents. To make it impossible for an agent to use incorrect names and pass names without permission to do so to third parties we do not use any absolute names or references. Instead the naming mechanism is based on *tickets* issued by location. Whenever an agent needs to have a name for some resource (e.g. a new scope created by its request) a location generates a new structure consisting of the agent name and the resource reference. This structure is associated with a random ticket number which is passed to the agent. The ticket issuing location is the only entity that can decode ticket numbers into resource references. Location has the full control over name passing and prevents issuing and usage of incorrect names. Note that a legitimately owned name may become invalid if a resource is destroyed or the agent no longer has rights to access it. Assuming that agent

names are unforgeable (we need some additional scheme to ensure agent names validity) tickets numbers can be exposed without any risk for the owner. When an agent wants to share or pass a resource to another agent it requests the location to issue additional ticket for another agent. The major advantage of this scheme is that the agent requesting a new ticket still has, being the owner of the ticket, the full control over the resource access which this new ticket allows. At any moment it can send a request to the location to invalidate the ticket which will have an immediate effect on the ticket user. In addition an owner of a resource can add other agents to the list of owners and remove itself from the list. When an agent becomes a resource owner it can control resource usage and issue new tickets.

This procedure cannot be used to exchange agent names since to do that agents would have to already know the names of each other. For this purpose we introduce the `handshake` operation which implements secure and atomic exchange of names within a group of agents. Each agent receives names of all its peers made specifically for the agent. Each name is a ticket number usable only by the agent and referring to a name of another agent. The second argument of `handshake` operation is the list of tuples received from fellow agents within one scope. Location knows how to identify agents from tuples they produce (see [5]). `handshake` operation must be executed symmetrically by all the agents for the same list of agents (although tuples in the list may be different for each agent). The operation fails for *all* agents if there is at least one not executing the handshake at all or executing it with a different set of agents. Unsuccessful handshakes are unblocked after a timeout determined by the location.

4 Exception Propagation via Coordination Space

Previously we have developed a mechanism for propagating exceptions among independent, anonymous and asynchronously-communicating agents. In this section we give only a general and brief overview of the work to allow us to discuss problems of exception propagation between scopes, introduced in section 5. The detailed discussion of the mechanism and its experimental implementation for the Lime mobile agents system [7] can be found in [5].

The mechanism of the exception propagation is complimentary to the application-level exception handling. All the recovery actions are implemented by application-specific handlers. The ultimate task of the mechanism is to transfer exceptions between agents in reliable and secure way. However the enormous freedom of behaviour in agent-based systems makes it impossible to guarantee reliable exception propagation in a general case. Although we can clearly identify the situations when exceptions may be lost or not delivered within a predictable time period. If an application requires cooperative exception handling at certain moments then for that time, agents behaviour must be constrained in some way to disallow any unexpected migrations or disconnections.

There are three basic operations available to agents to receive and send inter-agent exceptions. They are supplementary to the application-level mechanism and their functionality do not overlap.

throw	**wait**	**check**

The first operation, `throw`, propagates an exception to an agent or a scope. Important requirements is that the sending agent prior to sending an exception must have got a message from the destination agent and they both must be in the same scope. These two variants of the operation has the following form:

- `throw(m, e)` - throws exception `e` as reaction to message `m`. The message is used to trace the producer and to deliver the exception to it. The operation fails if the destination agent has already left the scope in which the message was produced.
- `throw(s, e)` - throws exception `e` to all the participants of scope `s`.

The crucial requirement to the propagation mechanism is to preserve all the essential properties of agent systems such as agents anonymity, dynamicity and openness. The exception propagation mechanism does not violate the concept of anonymity since we prevent disclosure of agent names at any stage of the propagation process. Note that the `throw` operation does not deal with names or addresses of agents. Moreover we guarantee that our propagation method cannot be used to learn names of other agents.

Also the mechanism itself does not restrict agent activities in any way. Though agent dynamicity and reliability of exception propagation are conflicting concepts we believe that it is the developers who must take the final decision to favour either of them. Notion of openness is the key for building large-scale agent systems. Proper exception handling was proved to be crucial for consistent and reliable component composition. It is even more so for mobile agent systems where composition is dynamic and parts of the system are developed independently. To support large-scale composition of exceptions-enhanced agents we are going to elaborate a formal step-wise development procedure.

Two other operations, `check` and `wait` are used to explicitly poll and wait for inter-agent exceptions.

- `check` - if there are any exception pending for the calling agents raises exception `E(e)` which is a local envelop for the oldest pending exception.
- `wait` - waits until any inter-agent exception appears for the agent and raises it in the same way as the previous operation.

We also redefine semantics of blocking Linda operations so that unblock whenever a coordination space exception appears and throw this exception inside of the agent.

4.1 Traps Mechanism

The propagation procedure expressed only with the primitives above would be too restrictive and inflexible for mobile agent systems. To control the propagation process in a way that accounts for various agent behaviour scenarios we

introduce a notion of *trap*. A trap is a set of rules created by agents that controls exception propagation and exists independently of the creating agent. Traps are stored and manipulated by a location that provides the coordination space. A trap is essentially a list of rules that chose reaction for a coordination space exception. It can be represented as a CASE construct where rules are associated with exceptions (see Figure 4). Exception matching and comparison are non-trivial issues usually dictated by the language of choice.

```
case e is
   when E1 => op1
   when E2 => op2
   ...
   when En => opn
   when others => abort
end case
```

Fig. 4. Trap is a CASE-style construct

A trap can be enabled when there is an incoming message for the agent that created the trap. Agent may have several traps and traps are automatically organised into an hierarchical structure. When an exception appears, the most recently added trap is activated. If the trap fails to find a matching rule for the exception, the exception is propagated to the second most-recent trap and so on. Agents can dynamically create, add and remove traps. The following operations are used to express trap structure:

- `deliver` - delivers the exception to the destination agent. The exception is stored until the destination agent is ready to react to it or the containing scope is destroyed;
- `relay(t)` - propagates the exception to a trap `t` which may be a trap of another agent. Name of a trap can be only learnt through negotiations with the trap owner. Owner of the trap becomes the destination the propagated exception;
- `abort` - leaves the current trap and transfers control to the next trap in the hierarchy.
- `if` *(condition)* `then` *ac* - action *ac* is applied conditionally;
- . *(concatenation)* - forms a new action by concatenation of two other actions.

The `deliver` operation was designed to be able to tolerate agent migration and connectivity fluctuations. It introduces some level of asynchrony and makes the whole exception propagation scheme more suitable to the asynchronous communication style of coordination space. The `relay` operation is a tool for building linked trap structures supporting a disciplined cooperative exception handling. Discussions and examples related to this approach can be found in [5].

Preconditions for the `if` operation are formed from the following primitives:

- `local` - holds if the owner of the trap is joined to the current scope
- `local(a)` - holds if agent `a` is joined to the current scope
- `tuple(t)` - holds if there is a tuple matching template `t`
- $\neg$, $\vee$, $\wedge$ - logical operations that can be used on the predicates above

Rule preconditions and concatenation provide a very expressive mechanism that may form traps for many interesting and useful scenarios. For example, a rule in a trap could make multiple deliveries to involve several agents, or, depending on the locality of the trap owner, another agent or even a trap in a different location.

5 Exception propagation through scope boundaries

The exception propagation mechanism described above works well within the boundaries of a scope. However in many cases a scope corresponds not to a completely isolated activity but rather forms a part of a more general activity of the containing scope. In such a case a failure of a scope may disrupt activity of the containing scope and require agents of the containing scope to execute some recovery actions. In this work our intention is to introduce exception propagation through scope boundaries in a way that is smoothly integrated with the concepts of the scoping and exception propagation mechanisms discussed above.

It is very natural to try and take the advantage of the scope nesting mechanism to build a scope-based exception recovery. However the specifics of the mobile agents and the scoping mechanism bring unexpected complications. First of all, scope nesting does not necessarily correspond to a logical nesting of scope activities and this presents problems in interpreting exceptions propagated from inner scopes. Another complication is that different kind of scopes - nested, at the same level or even unrelated and from different locations, can be linked by a common global state of an agent. Hence, in addition to the hierarchy of scopes introduced by the scoping mechanism, we have to take into consideration relations between scopes introduced by agents.

Currently we are looking into possible realisations of the propagation mechanism for the CAMA system. Below we briefly present several promising approaches. They can be classified into the two categories - the first one looks at the problem from the viewpoint of a failed scope and the second one discusses recovery schemes for a containing scope.

Case 1. Throwing an external exception. One possible solution is to allow agents to throw an external exception which results in the immediate termination of the failed scope. When this happens the scope closes and all other scope participants get a notification of the scope failure.

Case 2. Common root trap tree. Agents create a common root trap tree to propagate exceptions in a disciplined manner outside a scope. Through a negotiation process they exchange trap names and put them together in a common trap tree structure. For example, this tree may be initially created by one of the agents and then updated by others. An exception indicating the scope failure (as classified by the traps) is propagated through the chain of traps and finally arrives to some or all of the agents. If it during this propagation it arrives to the root of the common tree, the scope is terminated with an external exception. This solution is more general and flexible than the first one because different recovery scenarios can be built for different types of exceptions and agent groups.

Case 3. Internal propagation. Taking into account the fact that each agent participating in a nested scope is also present (though does not necessarily active) in the containing scope we can propagate exceptions through the internal state of an agent and, if required, trigger recovery actions in the containing scope. In this case after an unsuccessful cooperative recovery in the failed (nested) scope each participant throws an exception in some of its active scopes. Exception propagation here is fully controlled by an agent and does not necessarily relate to any existing scopes structure. We believe that offering such flexibility may be dangerous and, besides, it is becoming very hard to analyse systems formally. However some situations may require such propagation style. For instance, when an agent acts concurrently in two scopes and these activities are interrelated (say it buys something in one scope and sells it in another). An exception thrown in the scope where the agent sells may require actions in the scope where it buys (but never vice versa as there is one-direction information flow between these scopes).

Another part of the problem is developing a recovery scheme for a containing scope. An important point to note here is that the activities of a containing and a nested scopes in a general case are asynchronous. Some effort must be taken to put agents in the containing scope into some correct state suitable for recovery actions caused by the failed sub-scope. We discuss here two possible solutions.

Case 1. Throwing to everyone. Whenever an exception from a nested scope appears it is thrown to each participant of the containing scope. This triggers normal trap mechanism which involves all the usual recovery procedures that would take place in a case of a local exception. In other words an external exception from a sub-scope appears as a new local exception for all the agents although it may be distinguished as an sub-scope exception by its type.

Case 2. Throwing to the failed scope participants. According to the previous approach, whenever a sub-scope exception is thrown each agent of the containing scope may be interrupted and involved in handling the exception. However we can exclude those agents that are not associated with the failed contained scope from the recovery initial recovery actions because their involvement may be superfluous in the situations when recovery can be done by the nested scope participants. In this case we still must ensure that the exception is propagated

to all the agents of the containing scope if that group of agents fails to recover themselves.

6 Acknowledgements

This research has been supported by the EC IST grant 511599 (RODIN).

References

1. F. Cristian. Exception handling and tolerance of software faults. In M. Lyu, editor, Software Fault Tolerance, pages 81107. Wiley, 1995.
2. Di Marzo, G. and Romanovsky, A. *Designing Fault-Tolerant Mobile Systems*. In Proceedings of the International Workshop on Scientific Engineering for Distributed Java Applications (FIDJI 2002), Luxembourg-Kirchberg, Luxembourg, 28-29 November 2002 Guelfi, N., Astesiano, E. and Reggio, G. (eds.). LNCS 2604 pp.185-201. Springer-Verlag 2003.
3. A. Iliasov, L. Laibinis, A. Romanovsky, E. Troubitsyna. *Towards Formal Development of Mobile Location-based Systems*. To be presented at Workshop on Rigorous Engineering of Fault Tolerant Systems, 19th July 2005, at FME 2005. Newcastle upon Tyne, UK.
4. J.-R. Abrial. The B-Book. Cambridge Univ. Press, 1996.
5. A. Iliasov, A. Romanovsky. *Exception Handling in Coordination-based Mobile Environments*. In Proc of the 29th Annual International Computer Software and Applications Conference Edinburgh, Scotland, July 26-28, 2005. IEEE CS Press. 2005.
6. *The Mobile Agent List*. reinsburgstrasse.dyndns.org/mal/preview/preview.html.
7. G. P. Picco, A. L. Murphy, G.-C. Roman. *Lime: Linda Meets Mobility*. Proc of the 21st Int. Conference on Software Engineering (ICSE'99), Los Angeles (USA), May 1999.
8. L. Fiege, M. Mezini, G. Muhl, A. P. Buchmann. *Engineering Event-Based Systems with Scopes*. In Proc. of ECOOP 2002, pp.309-333. 2002.
9. J. Baumann, F. Hohl, K. Rothermel, M. Straßer. *Mole - Concepts of a Mobile Agent System*. World Wide Web Journal. 1(3), pp.123-137. 1998.
10. G. Agha and C. J. Callsen. *ActorSpace: An Open Distributed Programming Paradigm*.Proceedings 4th ACM Conference on Principles and Practice of Parallel Programming, ACM SIGPLAN Notices, pp.23-323, 1993.
11. C. J. Callsen and G. Agha. *Open Heterogeneous Computing in ActorSpace*. Journal of Parallel and Distributed Computing, vol. 21,3, pp.289-300, 1994.
12. I. Satoh. *Mobile agent-based compound documents*. Proc. of the ACM Symposium on Document Engineering 2001, pp.76-84. 2001. I. Satoh. MobileSpaces: A Framework for Building Adaptive Distributed Applications using a Hierarchical Mobile Agent System. Proc. of the ICDCS 2000, pp.161-168. 2000.
13. L. Cardelli and A. D. Gordon. Mobile ambients. In Maurice Nivat, editor, Proc. FOSSACS'98, volume 1378 of Lecture Notes in Computer Science, pages 140–155. Springer-Verlag, 1998. Also appear in Theoretical Computer Science, 240(1), pp.177-213, June 6, 2000.
14. I. Merrick, A. Wood. *Coordination with Scopes*. Proceedings of the 2000 ACM symposium on Applied Computing, Como, Italy, pp.210-217, 2000.

Modeling Exception Handling: a UML2.0 Platform Independent Profile for CAA

Alfredo Capozucca, Barbara Gallina, Nicolas Guelfi, and Patrizio Pelliccione

Software Engineering Competence Center
University of Luxembourg
6, rue Richard Coudenhove-Kalergi
Luxembourg, L-1359 -Luxembourg

Abstract. Complex fault-tolerant distributed systems have a growing need of new functional and quality requirements. An immediate consequence of this is an increasing need of new methods for developing complex fault-tolerant distributed applications.
Coordinated Atomic Actions (CAAs), making use of exception handling mechanism, offer an approach to ensure the needed requirements of reliability, availability and fault tolerance.
Unfortunately, there is currently no method for the high-level modeling of such systems. In this paper, in order to offer an instrument for modeling exception handling, we propose a UML2.0 Platform Independent Profile for CAAs that allows designers to describe complex systems separating the specification from the implementation on a specific technology platform.

1 Introduction

Implementing complex fault-tolerant distributed applications is labor intensive and error-prone. Such systems have increasingly new functional and quality requirements. An immediate consequence of such evolutions in the requirements of systems is an increasing need of new methods that assist in these systems development. Furthermore, there is a growing interest in the area of application-level fault tolerance and cooperative exception handling as the main paradigm for developing structured fault-tolerant architectures. Unfortunately, there is currently no method that assists the software engineer in developing such systems.

This situation has motivated the investigation of a UML-based, generative and architecture-centric method to support cost-effective, disciplined and high-level development of complex fault-tolerant distributed systems families. This effort is conducted by the Software Engineering Competence Center of the University of Luxembourg (SE2C) in the context of the CORRECT project [2]. The CORRECT project makes use of Coordinated Atomic Actions (CAAs) [9] that offer approaches for error recovery and of the framework called Dependable Remote Interacting Processes (DRIP) [10] that embodies CAAs in terms of a set of Java classes.

UML [6] provides a standard-based high-level specification language as well as the possibility to clearly separate concerns at different development phases.

Model-driven Engineering (MDE) advocates separating the specification of system functionality, the "what", from the implementation of that functionality on a specific technology platform, the "how" [5]. Platform Independent Models (PIMs) specify the structure and functionality of a system while abstracting away technical details. Platform Specific Models (PSMs) specify how the functionality has to be realized on a selected platform. MDE takes benefits of PIMs and PSMs for generating the executable code of a system by deriving PSM specifications from PIMs by (formal) refinement, and then transforming PSMs to code for a specific platform. This approach allows relating design to requirements and analysis, as well as verifying and testing the application, and generating code for different technologies by transforming PIMs.

In this paper we focus on the "what", i.e. on the specification of the system, by proposing a UML Platform Independent Profile for CAAs. This work refers to a previous work [3] where has been proposed the FTT-UML, a CAA-based UML profile for modeling fault-tolerant business processes. We fix some discovered problems and we propose a new UML2.0 profile for CAAs, able to model fault-tolerant systems on any domain. In other words, the proposed profile provides the necessary features for modeling "pure" CAAs.

After an introduction on CAA in Section 2, we present the FTT-UML profile in Section 3 highlighting the problems driving the definition of the proposal profile, that is explained in Section 4. We make use of a simple example to show how all aspects of a CAA can be modeled with the proposed profile. Conclusion and future works close the paper.

2 Coordinated Atomic Actions

Coordinated Atomic Actions (CAAs) is a fault-tolerant mechanism using exception handling to achieve dependability in distributed and concurrent systems [9]. Thus, using CAAs we can develop systems that comply with their specification in spite of faults having occurred or occurring.

This mechanism unifies the features of two complementary concepts: *conversation* and *transaction*. Conversation [7] is a fault-tolerant technique for performing coordinated recovery in a set of participants that have been designed to interact with each other to provide a specific service (cooperative concurrency). These participants are called **roles** in the context of CAAs. Transactions are used in order to deal with competitive concurrency on external objects, which have been designed and implemented separately from the applications that make use of them.

Every external object should be under the control of the transactional system to guarantee the ACID (atomicity, consistency, isolation and durability) properties. Anyway, there are some external objects that, because of their own nature, could not be rolled back. Therefore, they must be managed in a special way to guarantee the ACID properties on them. The external objects that can be rolled back are called *recoverable* and those that require explicit manipulation to be left in a consistent state are called *unrecoverable*. This categorization of external objects allows us to satisfy the ACID properties on them.

A CAA is the basic element of this technique, which characterizes an orchestration of actions executed by a group of roles that exchange information among them, and/or access to external objects (which at the same time are shared with other CAAs) to get a common goal. A CAA starts when all its roles have been activated and finishes when all of them have reached the CAA end. If for any reason an exception is raised in at least one of the roles belonging to the CAA, appropriate recovery measures have to be taken. Facing this situation, a CAA provides a quite general solution for fault-tolerance that consists of applying both forward and backward error recovery techniques in a complementary or combined manner.

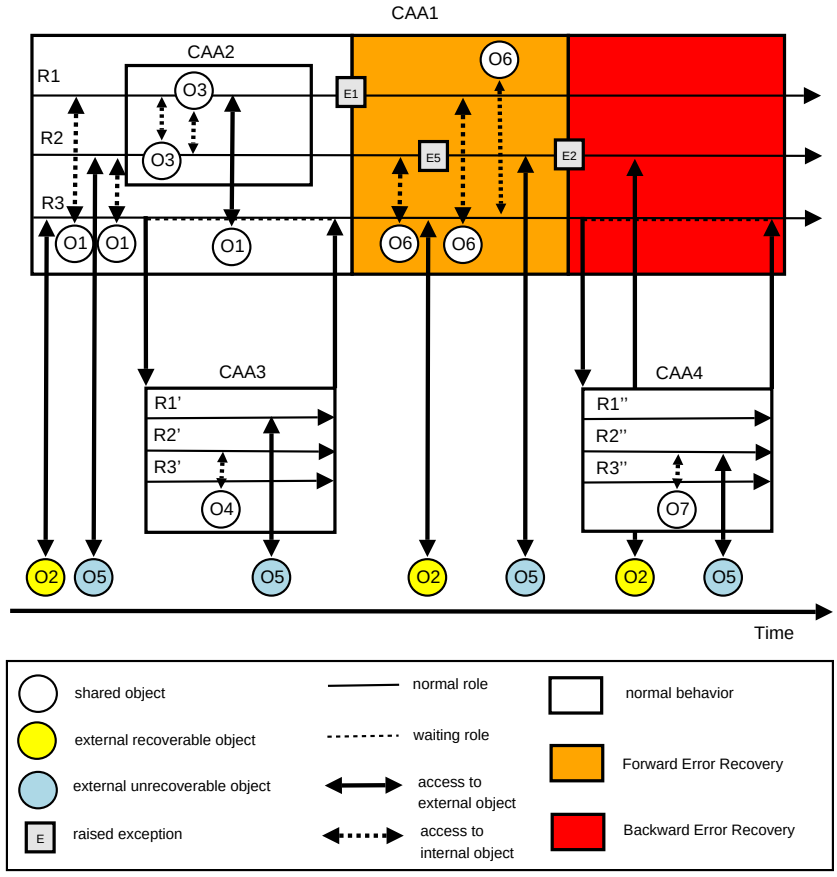


Fig. 1. Coordinated Atomic Actions.

Figure 1 shows how CAAs can be designed in a recursive way using **nesting** and/or **composing**. Nesting is defined as a subset of the roles of a CAA (CAA_1) defining a new CAA (CAA_2) inside the enclosing CAA (CAA_1). The roles of CAA_2 (R_1 and R_2) are the same roles that have been defined for CAA_1, but

the operations that they are doing inside CAA_2 are hidden for the other roles (R_3) (and other nested or composed CAAs) that belong to CAA_1. Accesses to external objects within a nested CAA are performed as nested transactions, so that, if CAA_1 terminates exceptionally, all sub-transactions that were committed by the nested (CAA_2) are aborted as well. Any role of a CAA can only enter one nested CAA at a time. Furthermore, a CAA terminates only when all its nested CAAs have terminated as well. Note that if the nested CAA_2 terminates exceptionally, an exception is signalled to the containing CAA_1.

Composed CAAs are different from the nested in the sense that we can reuse the CAAs designed in other contexts. Thus, composition allows us to develop open distributed systems. A composed CAA (CAA_3) is an autonomous entity with its own roles ($R_1{}', R_2{}'$ and $R_3{}'$) and external objects (unrecoverable O_5). The internal structure of the composed CAA_3 (i.e., set of roles, accessed external objects and behavior of roles) is hidden from the calling CAA_1. A role belonging to CAA_1 that calls the composed CAA_3 synchronously waits for the outcome. Then, the calling role resumes its execution according to the outcome of the composed CAA_3. If the composed CAA_3 terminates exceptionally, its calling role (which belongs to CAA_1) raises an internal exception which is, if possible, locally handled. If local handling is not possible, the exception is propagated to all the peer roles of CAA_1 for coordinated error recovery.

If the composed CAA_3 has terminated with a normal outcome, but the containing CAA_1 has to roll back its effects (abort operation), the tasks that were done in the composed CAA_3 are not automatically undone. Thus the CAA_1, in order to guarantee the ACID properties on the external object, needs to carry out a specific handling, which may have a composed CAA (CAA_4) to abort (or compensate, if there is at least one unrecoverable external object) the effects that have been done by the CAA_3.

In order to formally express the semantics of each possible kind of CAA outcome (**normal, exceptional, abort** and **failure**), we use statecharts [4](Figure 2). The specification is composed of two state machines, which are running in parallel. The machine on the left side represents the *System* which evolves according to the events that are sent (events with a line over them) by the state machine that is on the right side and represents a *CAA*.

A CAA is designed to provide a service, which is called by the users of the System where the CAA is embedded. The invocation of the service is represented by the event *Op*. We can assume that this event comes from the environment. The *W* state represents the execution of the service. If the service is able to satisfy its post-conditions, then the CAA terminates normally. Therefore, the CAA reaches the *OK* state, publishing at the same time the *normal* event, thus the System goes to the well defined *s1* state.

The state of the CAA is defined as the state of the System plus the state variables that are used to deliver the service. Thus, the CAA could be in an unspecified (not well defined or inconsistent) state, but the System is left in a specified (well defined or consistent) state. If the CAA does not normally finalize (because the post-conditions are not met), then each role must signal an

exception to indicate the outcome. The roles should agree about the outcome, and each role should signal the same exception. In this case, the CAA emits the event *exceptional* and it goes to an unspecified state. When the System receives the event *exceptional*, it goes to any well defined state, even if the specific service that is provided by the CAA could not be delivered satisfactorily.

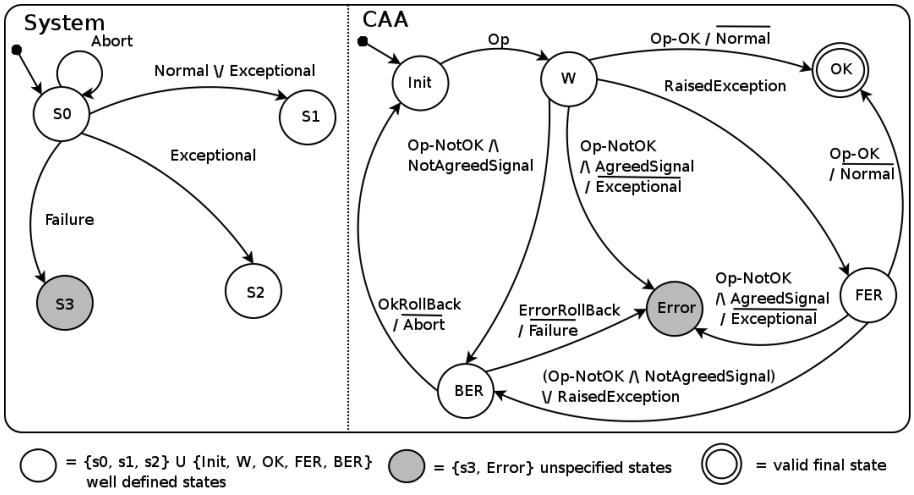


Fig. 2. Outcomes of a CAA and their semantics.

If an exception is raised during the normal execution of the CAA (*s0* state), then a process of exception handling is triggered. This exception handling process is defined as a combination of Forward (FER) and Backward Error Recovery (BER). The FER mechanism is represented by the *FER* state and, depending on how successfully it can recover from the exception, the CAA may still terminate normally. In any case (*W* or *FER* states), if the CAA finalizes not normally and the roles disagree about the outcome, then the CAA attempts to abort the action by undoing the effects through the BER mechanism.

If the post-conditions cannot be reached through the FER or an exception is raised again thought this recovery process, the CAA must attempt to roll back the state through BER (*BER* state). If BER is applied successfully, the CAA publishes the event *abort* and it goes to the initial state (*Init*). The System receives this event and moves to the same state (*s0*) where it was before calling the CAA.

If the BER is unsuccessful, then the CAA must publish the event *failure*. In this case, both the CAA and the System are left in a unspecified state.

3 FTT-UML: a UML-profile for Fault-Tolerant Transactions

The FTT-UML Profile has been presented at OTM Workshops in 2004 [3] and constitutes an important step towards modeling dependable complex business

processes since it provides means in order to manage exceptions and in order to describe nested transactions in e-business distributed contexts.

FTT-UML privileges Activity Diagrams and Class Diagram among all the UML 1.5 diagrams available: Activity Diagrams in order to model the behavior of the business process and Class Diagrams in order to model the business domain data structure. In this section we present a list of FTT-UML elements directly related to CAA concepts. The starting point for this list has been [1]. FTT-UML has been proposed in e-business context and therefore it provides also means in order to model typical e-business situations, such as filling a form, but these aspects are, in our context, not crucial.

- **Activity Diagram:** an Activity Diagram represents a CAA.
- **Partition:** a partition, which is a vertical solid line dividing an Activity Diagram, represents a CAA role. Each partition contains the actions performed by the corresponding role.
- **Object Flow State:** data structures are modeled using UML class diagrams. The corresponding data are used in activity diagrams exploiting object flow states represented by rectangles. Subsection 3.3 of [3] provides more detailed information about the data manipulation primitives provided by this notation.
- **Action state:** a role performs actions, represented by action states (corner-rounded rectangles). A state has pre- and post-conditions (first paragraph in section 3 of [3]). An action is executed when all its pre-conditions are satisfied, and then it offers a token to all its outgoing edges. Action states represent method calls on objects. The name of this state refers to the following topology: objectName.method() where objectName is an accessible object (from the role point of view) and method is a visible method of this object. Arguments of this call are represented by the name of the objects inside the parenthesis, or by linking objects to the action state (subsection 3.4 of [3]).
- **Transition:** a transition represents the execution flow (solid lines) or the data flow (dotted lines).
- **Final PseudoState:** represents both a normal outcome (if the activity does not produce outputs) and an implicit synchronization point among roles. In CAA, in fact, roles have to synchronize themselves in the exit and agree about the outcome.
- **Initial PseudoState:** represents the only starting point of the CAA in the case of synchronous roles. It means that CAA starts when all the role are activated. In cases in which asynchronous roles are present, more than one initial node has to bes used in order to model the asynchronous activation of the roles themselves.
- **ObjectFlowState with stereotype ≪output≫:** represents a normal outcome with an output.
- **Final PseudoState with stereotype ≪except≫ and labeled with the exception name:** represents an exceptional outcome without parameter.
- **ObjectFlowState with stereotype ≪except≫ and labeled with the exception name:** represents an exceptional outcome with a parameter.

- **Sub-activityState:** a sub-activity state models a nested CAA.
- **ActionState with the stereotype ≪invited≫:** represents a role in an enclosing CAA that does not contain the sub-activity node (representing a nested CAA), but participates in that nested CAA.
- **ObjectFlowState with an edge to the sub-activity node:** represents providing an input to a nested process.
- **Transition with the stereotype ≪compensate≫ going to the nested process sub-activity state:** represents undoing the effects of a nested process.
- **Activity diagram having the stereotype ≪compensate≫:** represents a compensation, it undoes the effects of a nested process already successfully terminated, whose effects are requested to be undone inside the running CAA.
- **Action state with stereotype ≪raise≫ and named by the exception to raise:** represents a role of a CAA that wants to raise an internal exception. Such a state must not have outgoing transitions.
- **Partition with the stereotype ≪ExceptionResolution≫:** represents a graph whose purpose is the resolution of concurrent exceptions into a single exception to raise. Such a partition contains an oriented graph, where action states represent exceptions.
- **Action state with the stereotype ≪handler≫ and named by the exception name:** indicates the starting point of a subset of a role activities that represents a handler for one of the exceptions of the exception resolution graph. (variants: for a parameterized exception, the action ≪raise≫ has an incoming edge from an object flow state and the action ≪handler≫ has an outgoing transition to an object flow state representing the parameter).

3.1 The Discovered Problems

Inside a CAA three phases may be distinguished: one in order to describe the normal behavior, one in order to capture the exception handling behavior and finally, one in order to capture the roll back behavior. It should, anyway, be clear that the last behavior is a special case of exception handling.

In FTT-UML these three behaviors are not well defined. Only the normal one is well defined and detailed, while the other two are only sketched and never illustrated. By reading the profile explanation, it comes out that there are two particular stereotypes, one called *compensate* which stereotypes an Activity Diagram and should be used in order to undo effects when roll back is needed, and one, called *handler*, in order to describe exception handling behavior.

These two stereotypes however are never illustrated and therefore it is not clear if they have to be used in a particular partition or not. Suppose, for example, to put the ≪handler≫ stereotype in all the partitions (roles) that may have a handler in order to manage an exception; the resulting diagram would be overloaded.

Moreover CAAs could be composed and nested and this difference does not appear in the FTT-UML profile.

Roles of an enclosing CAA may, in FTT-UML, change their name by entering in a nested process. This possibility in CAA theory is not present. In nested CAAs in fact still appear roles that were already part of the enclosing CAA. Different roles are only possible in the case of CAA composition.

Logically, a CAA starts when all roles have been activated (though it is an implementation decision to use either synchronous or asynchronous entry protocol) and finishes when all of them reach the action end (see [8]). In FTT-UML profile the user may also model implementation details by defining synchronous/asynchronous roles. A CAA model, however, in order to ensure a good separation of implementation and business concerns, should not contain implementation details.

4 Improving the FTT-UML profile

Taking the FTT-UML Profile as basic reference to represent CAA by UML 1.5, we have defined a new one (called CAA-UML), which has as main improvement the fact that it supports completely and purely CAA semantics. Supporting only the basic features of CAAs avoids embedding characteristics for any specific domain, which would be a bias of our target.

Using the last version of UML (2.0) as our notation language also allows us to find a better representation for some basic CAAs aspects, i.e. detecting and handling exceptions, as well as the definition of context where an exception may happen.

In the following, we give the detailed list of elements that compose the CAA-UML Profile explaining improvements and solutions with respect to the problems found on its predecessor profile. To better explain the new proposed profile we provide an example of its use (Figure 3), which corresponds to the CAAs described in Figure 1.

- **CAA:** this is represented by an *Activity Diagram*. In Figure 3, CAA_1 and the nested CAA_2 are examples of how an Activity Diagram describes a CAA.
- **Role:** this is represented by a *Activity Partition*. In Figure 3, R_1 and R_2 and R_3 are partitions that represent roles of CAA_1, *Nested* CAA_2 and *FER for exception E1*.
- **Nested CAA:** as shown in the Figure 1, several roles of CAA_1 can enter into a nested CAA_2, which defines an atomic operation inside the enclosed CAA_1. This is represented by one, and only one, *Call Behavior Action* with stereotype ≪Nesting≫. This stereotyped *Call Behavior Action* has an incoming control flow arrow for each role that is participating in the nested CAA. These incoming flow arrows do not need labels like in the previous profile, because it is not required to change the names of the participating roles in the nested CAA.

  The *Nesting Call Behavior Action* must be in one of the roles belonging to the nested CAA. Since there is no defined semantics in order to choose where to put the nested CAA, the designer must put it in one of the involved roles (it does not matter in which one).

The nesting of CAA_2 in CAA_1, represented in Figure 1, becomes, in the profile, *Call Behavior Action* with the stereotype ≪Nesting≫, called CAA_2, that is inside R_1 belonging to CAA_1 (see Figure 3).

- **Composed CAA:** composition is the other kind of relationship that we can use to design CAAs. This is denoted by a *Call Behavior Action* with the stereotype ≪Composition≫ and represents the creation of a completely independent CAA (CAA_3 in Figure 1) with its own roles and objects. The composed CAA is called by a role belonging to the enclosing CAA (CAA_1 in Figure 1). In Figure 3, inside R_3 of CAA_1, this representation can be found. Thus, according to the last two elements presented, there are two different well defined ways to describe nesting and composing of CAAs. This feature is not present in the previous profile.
- **Outcomes:** the profile must be able to represent the four kinds of outcome that a CAA can return. The **normal** outcome is represented by a *Final Activity Node*. The **exceptional** outcome is represented by an *Final Activity Node* with the stereotype ≪Exception≫. The name of the stereotyped *Final Activity Node* represents the exception that is signalled to the enclosing context. It can happen when the post-conditions of the CAA are not met and every role agrees with the exception to signal. The representation of these elements can be found in Figure 3 in each CAA and in the FER as well.

  The outcome **abort** and **failure** are used to notify how successfully the BER has been ("abort" if the effects can be undo, otherwise it must be "failure"). The BER is part of the Coordinated Error Recovery mechanism, so these outcomes are generate automatically according to how well this mechanism could be applied with respect to the handled exception. The enclosing context (CAA_1 in Figure 3), from where the nested (CAA_2) or composed (CAA_3) CAA has been called, receives the corresponding not normal outcome ("exception", "abort" or "failure") through an exception, which is represented by an *Accept Event Action Object Node* with an outgoing *Interrupting Edge*. The node must be stereotyped with ≪Abort≫ (AbortEx, in Figure 3) or ≪Failure≫ (FailureEx, in Figure 3) according to the outcome. The *Interrupting Edge* must be incoming to the associated handler (*Call Behavior Node* called CAA_3 with stereotype ≪BER≫) for the exception that has been detected (details about the handler can be found in the next point).
- **Coordinated Error Recovery mechanism:** this mechanism allows designers to describe the substitution of the normal behavior execution by an exceptional behavior. This exceptional behavior starts to execute when an exception is detected. Firstly, the designer has to declare the area in which an exception could be detected and then specify the exceptional behavior. This area is represented by an *Interruptible Activity Region*. This region must have associated a set of exception handlers, one of which is called when one of its related exception is raised.

  As explained in the representation of the outcomes, an exception is represented by an *Accept Event Action Object Node* with an outgoing *Interrupting*

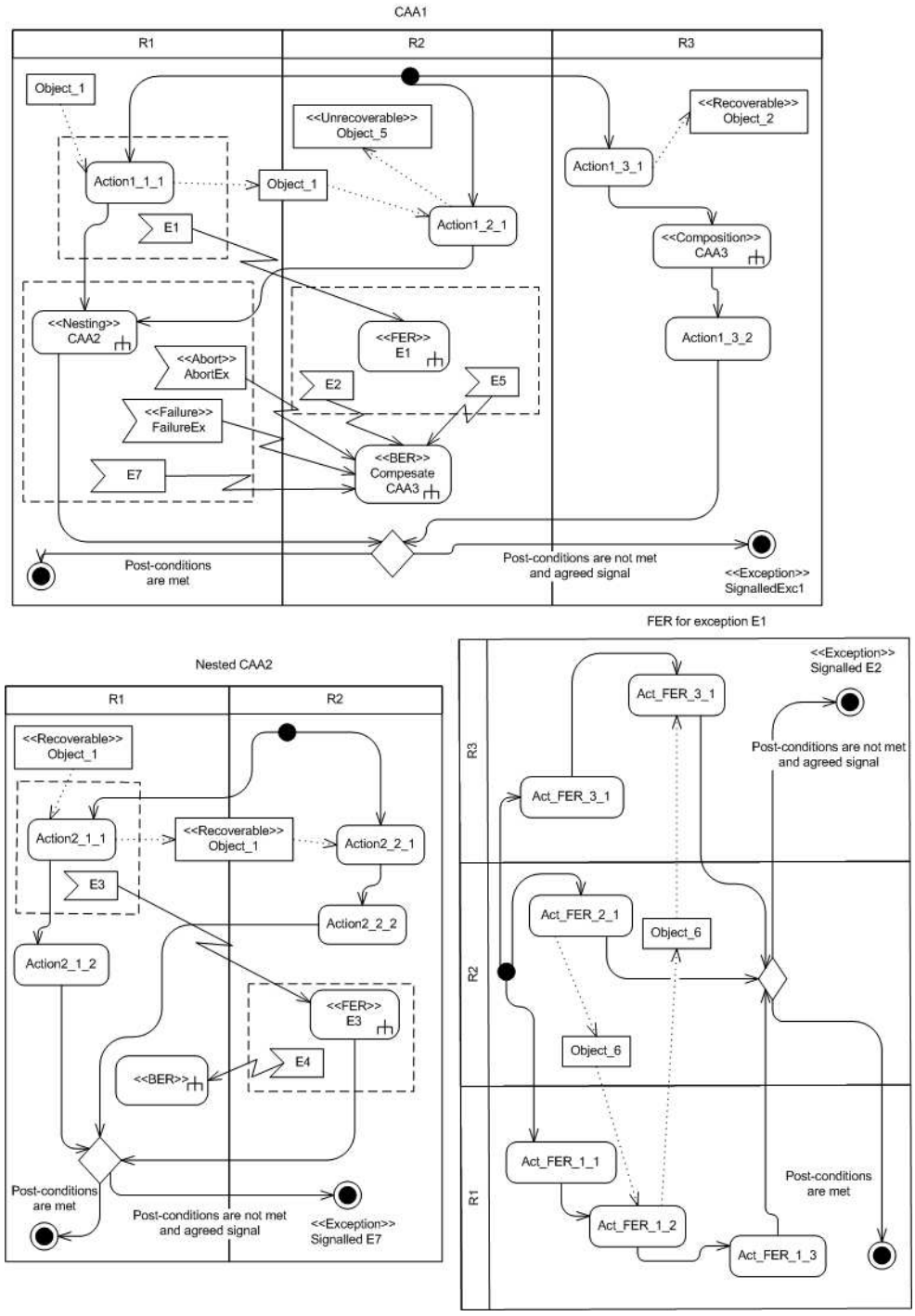


Fig. 3. UML representation of CAAs.

Edge. The exception handler, that is attached by the *Interrupting Edge*, is represented by a *Call Behavior Action*. The Coordinated Error Recovery mechanism allows us to combine FER and BER (*Call Behavior Action* with stereotype ≪FER≫ and ≪BER≫, respectively). Since there is no defined semantics in order to choose where to put the nested CAA, the designer puts it in one of the CAA roles (it does not matter in which one).

According to the exception semantics of CAA, if an exception is raised when the FER is executing (E_5, in Figure 3), the BER mechanism must be started.

A *Call Behavior Action* with stereotype ≪BER≫ and without name represents the classic roll back (the System is restored to its previous state). Otherwise, if the BER has a name, it means that a specific behavior must be executed to undo the effects that have been done by the CAA. In Figure 1, the BER uses the composed CAA_4 to leave the $Object_4$ in a consistent state. In Figure 3, it is represented by the *Call Behavior Action* with stereotype ≪BER≫ and name CAA_3. Since BER is a special case of FER the modeling way does not change therefore there is no need here to introduce another figure to represent it.

It is important to stress that an exception (represented by an *Accept Event Action Object Node*) with a stereotype ≪Abort≫ or ≪Failure≫ could only be found inside an *Interruptible Activity Region* where there is at least a nested or composed CAA (R_1 of CAA_1 has the nested CAA_2).

- **External unrecoverable object:** in this case we want to represent an external object that cannot be restored to its initial state after the operations applied inside the CAA. This kind of object is represented by an *Object Node* with the stereotype ≪Unrecoverable≫ ($Object_5$, in Figure 3). We must use a special handling to be sure that the ACID properties on this kind of objects are satisfied.
- **External recoverable object:** this is represented by an *Object Node* with the stereotype ≪Recoverable≫ (O_2, in Figure 3), and the meaning is the complement of the previous kind of object. Thus, the state for every object with this stereotype, can be restored to the last well defined state known just before the CAA starts. This kind of objects has full transactional support, therefore ACID properties are satisfied.
- **Local shared object:** this is represented by an *Object Node* ($O_{1,6}$, in Figure 3). This kind of object is used by the roles of a CAA to exchange information among them. A local shared object can become either a recoverable or unrecoverable external object depending on its recoverable or unrecoverable characteristics. In Figure 1, O_1 is local to CAA_1, but it becomes external recoverable for the nested CAA_2 (the fact that is recoverable could be seen in Figure 3).
- **Exception Resolution tree:** this is represented through a class diagram in which every class represents an exception and it is related to other exceptions by a generalisation relationship. By moving from the leaf to the root it is possible to find out how to manage exceptions in case of concurrency.

5 Conclusion

In this paper we presented a UML 2.0 profile for CAAs. The work is based on a previous work and, by putting in evidence the weak points of that approach, we have justified the introduction of a new one, based on UML 2.0. This profile is completely platform independent, separating the specification of system functionalities from the implementation of them on a specific technology platform. Thus, this paper represents an important contribution for high-level design of fault-tolerant complex distributed systems. Finally, through a theoretical example, we showed how a software engineer can model these systems detailing normal and exceptional scenarios.

On the future work side we are interested in integrating this profile in a development process from a high-level system description to the system implementation, through suitable refinements, able to preserve fault tolerance properties.

Acknowledgments This work has benefited from a funding by the Luxembourg Ministry of Higher Education and Research under the project number MEN/IST/04/04. The authors gratefully acknowledge help from A. Campéas, A. Romanovsky and R. Razavi.

References

1. A. Capozucca, N. Guelfi, and R. Razavi. Towards a UML-based Notation for CAAs. *TR-SE2C-04-05, SE2C-University of Luxembourg, Luxembourg*, 2004.
2. Correct Web Page. http://se2c.uni.lu/tiki/tiki-index.php?page=correctdoc.
3. N. Guelfi, G. L. Cousin, and B. Ries. Engineering of dependable complex business processes using uml and coordinated atomic actions. In *OTM Workshops*, pages 468–482, 2004.
4. D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
5. Object Management Group (OMG). OMG/Model Driven Architecture - A Technical Perspective, 2001. OMG Document: ormsc/01-07-01.
6. Object Management Group (OMG). Unified Modeling Language (UML): Superstructure version 2.0, final adopted specification (02/08/2003). http://www.omg.org/cgi-bin/doc?ptc/2003-08-02, 2003.
7. B. Randell. System structure for software fault tolerance. *IEEE Transactions on Software Engineering. IEEE Press*, SE-1(2):pp. 220–232, 1975.
8. A. B. Romanovsky and A. F. Zorzo. A distributed coordinated atomic action scheme. *Comput. Syst. Sci. Eng.*, 16(4):237–247, 2001.
9. J. Xu, B. Randell, A. B. Romanovsky, C. M. F. Rubira, R. J. Stroud, and Z. Wu. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In *Symposium on Fault-Tolerant Computing*, pages 499–508, 1995.
10. A. F. Zorzo and R. J. Stroud. A distributed object-oriented framework for dependable multiparty interactions. In *OOPSLA '99: Proceedings of the 14th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 435–446. ACM Press, 1999.

Practical Exception Specifications

Donna Malayeri[1] and Jonathan Aldrich[1]

Carnegie Mellon University, Pittsburgh, PA 15213, USA,
{donna+, aldrich+}@cs.cmu.edu

Abstract. Exception specifications can aid in the tasks of writing correct exception handlers and understanding exceptional control flow, but current exception specification systems are impractical in a number of ways. In particular, they are too low-level, too heavyweight, and do not provide adequate support for describing exception policies.

We propose a novel and lightweight exception specification system that provides integrated support for specifying, understanding, and evolving exception policies. Our tool, implemented as an Eclipse plugin for Java, combines user annotations, program analysis, refactorings, and GUI views that display analysis results. Using our tool, we analyzed six programs and observed a 50 to 93% reduction in programmer-supplied annotations.

1 Introduction

Exceptions can be very useful for separating normal code from error handling code, but they introduce implicit control flow, complicating the task of understanding, maintaining, and debugging programs. Additionally, testing is not always effective for finding bugs in exception handing code, and these bugs can be particularly problematic (for example, a program that crashes without saving the user's data).

For programmers to write correct exception handlers, they need precise information about all exceptions that may be raised at a particular program location. Documentation is inadequate—it is error prone and difficult to maintain. On the other hand, precise information can be obtained through a whole-program analysis of exception flow (including analysis of all libraries used), but this is not a scalable solution. Moreover, this would complicate team development; if one programmer changes exception-related code, the control flow in apparently unrelated parts of the program may change in surprising ways.

We believe that exception specifications are a useful tool for reasoning about exceptions (see, for example, [15, 13, 2, 7]). They serve to document and enforce a contract between abstraction boundaries, such as method calls. This facilitates modular software development, and can also provide information about exception flow in a scalable manner.

However, current exception specification systems are either impractical or flawed in one or more ways. In these solutions, specifications are either too

low-level, too heavyweight, or do not provide adequate support for describing a high-level exception policy. We believe that a good exception specification system should be lightweight while sufficiently expressive, and should facilitate creating, understanding, and evolving specifications.

We have implemented a tool, ExnJava, which provides practical and integrated support for exception policies. It combines user annotations, program analysis, refactorings, and GUI views that display analysis results. ExnJava raises the level of abstraction of exception specifications, making them more expressive, more lightweight, and easier to modify.

Note that we focus on the problem of *specifying* various properties of exception behavior, rather than a proposal for a new exception handling mechanism. The problem of specifying such properties exists independently of the exception mechanism, though of course some details of our solution would not apply directly to languages whose exception handling mechanism is significantly different than that of Java. The goal of our work is to shed light on properties that are important for an exception specification scheme, regardless of the handler mechanism.

In the next section, we describe the essential properties of a practical exception specification system; in Sect. 3 we describe how previous solutions have failed to meet one or more of these criteria. We describe the details of our system and how it meets these criteria in Sect. 4.

2 Practical Exception Specifications

If an exception specification system is to be practical, we believe that it must posses several essential properties; we enumerate these here. We use the general term "exception policy" to refer to programmers' design intent regarding how exceptions should be used and handled. An exception policy specifies the types of exceptions that may be thrown from a particular scope and the properties that exception handlers must satisfy.

In our view, a good exception specification system, which may include both language features and tools, should be lightweight while sufficiently expressive, and should facilitate creating, understanding, and evolving specifications.

Specification Overhead. The specification system must be lightweight. Programmers are not fond of writing specifications, so the benefits must clearly outweigh the costs. Additionally, incremental effort should, in general, yield incremental results. If a specification system requires that an entire program be annotated before producing any benefit, it is unlikely to be adopted.

Expressiveness. The system should allow specifying exception policies at an appropriate level of abstraction. It should support the common policy of limiting the exception types that may be thrown from some scope. Such scopes need not be limited to a method or a class. Rather, they could consist of a set of methods, a set of classes, or a module.

As a motivating example, suppose some module provides support for managing user preferences. Suppose also that its implementation should hide how the preferences are actually stored (e.g., files, a database, etc). Accordingly, its exception policy is that exceptions pertaining to these implementation details (e.g., `FileNotFoundException`, `SQLException`) should *not* be thrown by any of its interface methods. Rather, perhaps such exceptions would be wrapped by a higher-level exception type, such as `PreferenceStoreException`. If a low-level exception is erroneously thrown by an interface method, clients cannot write a meaningful exception handler without knowing the modules's implementation details.

Specifications could also include high-level properties of handlers while remaining lightweight.[1] Note that these policies need not be exposed to clients, as they may express implementation details. Such policies (for a particular scope) could include the following: handlers for exceptions of type E should be non-empty; thrown exceptions of type E should be logged; exceptions of type E should always be wrapped as type F before they escape the interface of this scope.[2]

Additionally, there should be a way to specify a policy independently of its implementation, though an implementation may perhaps be generated from a policy (e.g., code to log exceptions, or wrap some exception and rethrow). Solutions that make it easy to implement a policy are useful, but they do not obviate the need for one. Until it is possible to generate all desired implementations automatically—which may not ever be fully achievable—we believe that the distinction between specification and implementation is an important one.

Ease of Creating and Understanding Policies. The solution should provide tools that aid programmers in creating new exception policies and understanding existing policies. Without the aid of such tools, reasoning about exceptions is difficult due to their non-local nature. Such tools may, for example, include information on exception control flow.

Maintainability. The specification scheme should support evolving specifications as the code evolves, possibly through tool support. This differs from the property of being lightweight; a system may be lightweight but inflexible. The cost involved in changing specifications should generally be proportional to the magnitude of the code change.

In Java and in other commonly-used languages, exceptions automatically propagate up the call chain if there is no explicit handler. A specification system for these languages should take these semantics into account, so that small code changes do not require widespread specification changes.

[1] Supporting these high-level properties is the subject of our future work.
[2] A common practice recommended by Bloch [2], among others.

3 Related Work

Previous solutions have failed to meet one or more of the criteria described above; we describe each of these here.

Specifications. One well-known exception specification scheme is that of Java, which requires that all methods declare the checked exceptions that they directly or indirectly throw.[3]

Though we believe it is useful to separate exceptions into the categories of checked and unchecked (see, for example, [12, 2]), the Java design has a number of problems that make it impractical. Java `throws` declarations are too low-level; they allow specifying only limited exception policies at the method level. This leads, in part, to high specification overhead. It is notoriously bothersome to write and maintain `throws` declarations. Simple code modifications—a method throwing a new exception type; moving an handler from one method to another—can result in programmers having to update the declarations of an entire call chain.

There is anecdotal evidence that this overhead leads to bad programming behaviors [5, 23, 9]. Programmers may avoid annotations by using the declaration `throws Exception` or by using unchecked exceptions inappropriately. Worse, programmers may write code to "swallow" exceptions (i.e., catch and do nothing) to be spared the nuisance of the declarations [16, 11].

But even if programmers use checked exceptions as the language designers intended, exception declarations quickly become imprecise[4] as code evolves; statements that throw or handle exceptions will invariably be modified. In our study of several open-source Java programs (the subject programs are listed in Table 1), we found that between 16% and 81% of exception types within `throws` declarations were imprecise (with an average of 46%). Aside from illustrating the difficulty of maintaining `throws` declarations, this casts doubt on whether they are even a good tool for understanding exception flow and exception policies—though advocates often claim that this is one of their very benefits [8, 22, 2, 21].

Eclipse[5] provides a "Quick Fix" for updating a method's `throws` declaration if it throws an exception that is not in its declaration, but this can only be applied to a single method at a time. Consequently, programmers would have to

[3] In Java, the class `Exception` is the supertype of all exception types. One of its subtypes is `RuntimeException`, which represents *unchecked* exceptions. Exceptions that are a subtype of `Exception` but not a subtype of `RuntimeException` are *checked* exceptions; subtypes of `RuntimeException` are *unchecked exceptions*. A method must declare all checked exceptions that it throws (directly or transitively) in its `throws` declaration; unchecked exceptions may be omitted.

[4] For a method m, the declaration `throws` E is imprecise if m does not throw the exception E, though it may throw subtypes of E. The case where m throws neither E, nor its subtypes is an interesting one, as this is perhaps less likely to be an intentional design decision. However, due to space limitations, here we do not distinguish between these kinds of imprecision (though our tool does provide this capability).

[5] Available at `www.eclipse.org`.

iteratively update declarations until a fixpoint was reached. Eclipse also includes an optional warning that will list methods whose `throws` declaration is imprecise, but this too applies to a single method at a time.

There are several proposals for specifying method post-conditions on exceptional exit [6, 13, 1], but these are even more heavyweight than Java `throws` declarations. These solutions do, however, provide powerful verification capabilities. Whether such benefits will outweigh the significant cost of annotating an entire program, however, is unclear.

Other Work. There are several languages and language extensions which ease the task of implementing a policy, but provide no way to *specify* the policy. These include languages with first-class exception handlers [4] and languages that allow applying handlers to some set of methods or classes [10, 14]. However, unless new tools are created, these features will further complicate the task of reasoning about exceptions. It is also unclear how these schemes would work with programmer-supplied specifications; as far as we are aware, this problem has not been addressed.

Robillard and Murphy [18] provide a good methodology (though not a tool) for specifying exceptions at module boundaries; our tool builds on this work. A number of researchers have developed exception analysis tools [19, 20, 3], but they all perform a whole-program analysis, which does not scale [17]. For the task of understanding exception flow, Sinha et al. propose a set of views that display the results of their exception analysis, but for these they provide only a high-level design.

4 Features of ExnJava

We have designed and implemented an exception specification system for Java 1.4 that satisfies the initial criteria outlined in Sect. 2. Our design raises the level of abstraction of exception specifications, while remaining lightweight.

In developing our system, we found that Java classes and packages are not always appropriate units of abstraction; accordingly, we have designed a simple module system to be overlaid on standard Java code. A *module* consists of a set of Java classes or interfaces; each Java class or interface belongs to exactly one module (a default module is included for convenience). We add the following accessibility rule: methods may be accessed outside their module if and only if they are visible by standard Java accessibility rules *and* they are marked as module-public. Such methods are *interface methods*, as they effectively comprise the interface of the module; all other methods are *internal methods*.

We are in the process of implementing support for modules. Consequently, our current simplifying assumption is to equate modules with packages; each package is a separate module. Thus, methods with public or protected visibility are interface methods; private and package-private[6] methods are internal meth-

[6] Also known as "default" or "friendly" access.

ods. To emphasize our design goals, in the discussion below we use the term "module" rather than "package."

Our system, ExnJava, is implemented as an Eclipse plugin. It contains one language change: the Java rules for method `throws` declarations are relaxed such that only module *interface methods* require a `throws` declaration. ExnJava also includes module-level exception specifications, checked on every compilation. This is implemented as an extra-linguistic feature. Additionally, there is a Thrown Exceptions view to facilitate creating and understanding exception policies. Three refactorings help programmers evolve specifications: Propagate Throws Declarations, Convert to Checked Exception, and Fix Imprecise Declarations. In the subsections below, we describe each of these features, and our initial empirical results.

4.1 Specifying Exception Policies

In ExnJava, programmers specify exception policy at the module level. We believe this is a more appropriate level of abstraction than the low-level declarations of previous solutions, such as method-level declarations in Java. A module has two kinds of exception policy: one applies to each individual interface method, the other to the module as a whole.

Interface Method Policies. The exception policy of interface methods is specified using Java `throws` declarations. In contrast to Java however, the declarations for internal methods need not be specified—they are inferred by ExnJava.[7] Consequently, this design raises the level of abstraction of `throws` declarations.

To determine the checked exceptions thrown by internal methods, ExnJava performs an intra-module dataflow analysis. Within a module, the implementation of our analysis is similar to the whole-program analyses of previous systems [19, 20]. However, our analysis is scalable, as it depends on the size of each module rather than the size of the entire program. The results of this analysis, as well as additional information about exception control flow, are displayed in the Thrown Exceptions view, described below in Sect. 4.2.

There are several advantages to this scheme. First, annotations are more lightweight. As we describe in Sect. 4.4, in our subject programs we found that inference reduces the number of required declarations by a range of 50% to 93%. Also, inference gives programmers more precise information. Rather than examine Java `throws` declarations, programmers use the Thrown Exceptions view to determine the checked exceptions thrown by internal methods. And, in contrast to a pure exception inference tool, programmers can enforce exception policies by specifying `throws` declarations on interface methods.

[7] It may sometimes be useful to include `throws` declarations on internal methods; this is supported.

Module Policies. It is also useful to specify and enforce a policy that applies to all of the interface methods of a module. In ExnJava, for each module, programmers specify the set of exception types that may be thrown by its interface methods. This ensures that exceptions that are logically internal to a module are not leaked to its clients.

Module exception specifications thus ensure that the exception policy of each interface method (the types of exceptions that they throw) also conforms to the general exception policy of the module. Recall the example of Sect. 2 where the storage details of the user preferences module were to be hidden from clients. For such a module, its specification would include perhaps `PreferenceStoreException` but would *not* include `FileNotFoundException`.

4.2 Understanding Exception Policies

The Thrown Exceptions view (Fig. 1) displays the details of exception control flow, to help programmers understand the implemented exception policies. Without the information provided by this view, we believe that it would be difficult to correctly create and modify exception policies. Based on some anecdotal evidence, as well as our own programming experience, we believe that the general difficulty of programming with exceptions is partly due to lack of information on a program's exceptional control flow.

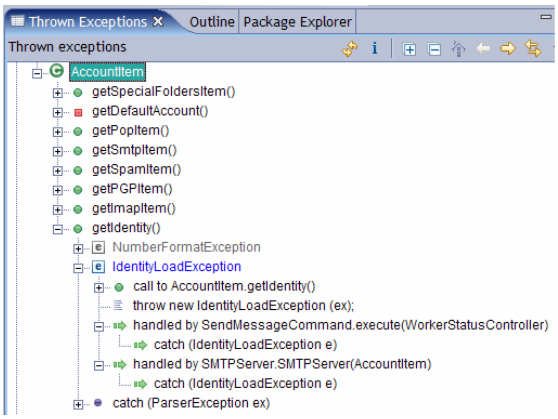


Fig. 1. The Thrown Exceptions view in method level mode.

The Thrown Exceptions view displays information computed by either a whole-project analysis or a per-module exception analysis; the former will provide more information, but the latter is more scalable. The view has two modes: method level and module level. The method level view, inspired by the work of Sinha et al [20], displays a tree view of the project's methods, grouped by pack-

age and class. For each method, the checked and unchecked exceptions[8] thrown by the method are listed, as well as the lines of code that cause the exception to be thrown. Using this view, the programmer can jump to method definitions that throw exceptions, and can also quickly jump to the ultimate sources of a particular exception (i.e., the original throw statements or library method calls that caused an exception to flow up to this part of the code.) Additionally, for each exception that a method throws, the view displays all catch blocks that may handle that exception. (This is limited, of course, to catch blocks in code available to the analysis.)

The module level view displays, for each module, the checked exceptions that are thrown by its interface methods. For each exception type, the methods that throw the exception are listed, as well as the detailed exception information described above. The module view can be useful for creating a module's exception policy and can also be used to discover possible errors in the exception policy. For example, if a particular exception type is only thrown by one or two methods, it is possible that the exception should have been handled internally or wrapped as a different exception type.

4.3 Evolving Exception Policies

Our system raises the unit of abstraction to which an exception specification applies; this alone makes it easier to evolve specifications. If the set of exceptions thrown by an internal method changes, no `throws` declarations need to be updated, unless one or more interface methods throw new exceptions. This often occurs when an exception handler is moved from one method to another in the same module. Though this is a conceptually simple modification, a number of internal methods may now throw a different set of exceptions. In standard Java, the `throws` declaration of each of these methods would have to be manually updated.

Propagating Declarations. Still, if a code change causes an *interface* method to throw new exceptions, the same "ripple effect" of Java `throws` declarations may result—requiring changes to the declarations of the transitive closure of method callers. To avoid this problem, ExnJava provides a Propagate Throws Declarations refactoring (accessible as an Eclipse "Quick Fix") that will propagate declarations up the call graph (see Fig. 2). The goal of this refactoring is to help programmers find the correct location for new exception handlers, rather than tempting them to carelessly propagate declarations to every method that requires them. To this end, the refactoring displays a checkbox tree view of the call graph (which includes only methods whose declarations need to be changed), which is initially collapsed to show only the original method whose declaration needs to be updated. The programmer then expands this one level to display

[8] Information on unchecked exceptions will not be complete, due to the fact that a whole-program analysis (including all libraries used) would be required. However, even partial information on unchecked exceptions can be useful.

the method's immediate callers (and callers of the methods that it overrides), and so on for each level in the tree. Checking a particular method in the tree will add the declaration to both that method and all the overridden superclass methods (so as not to violate substitutability).

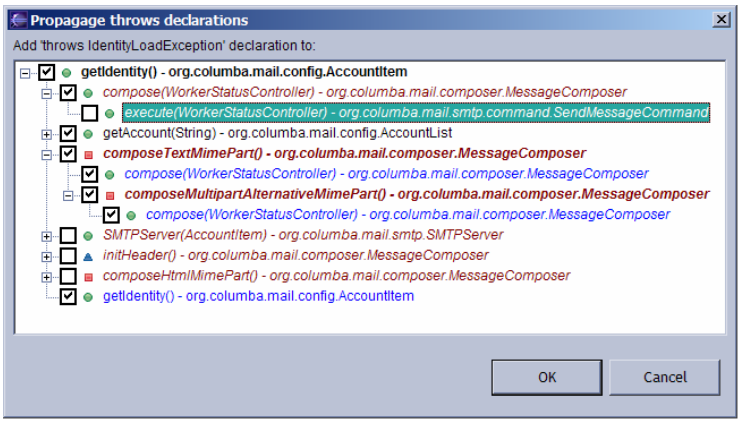


Fig. 2. The dialog for propagating throws declarations. Methods that are typeset in italics are those for which the module specification does not allow throwing this particular exception type.

The refactoring also incorporates the module exception specification; if updating the `throws` declaration of a particular method would violate the module specification, the method is displayed in a different color, with a tooltip describing the reason for the inconsistency. The declaration for the method can still be changed, but ExnJava will display an error until the package specification is modified.

Unchecked Exceptions. Sometimes, unchecked exceptions are used where checked exceptions are more appropriate. In fact, some programmers prefer to use unchecked exceptions during the prototyping phase, and then switch to checked exceptions later. ExnJava includes a Convert to Checked Exception refactoring which changes an exception's supertype to `Exception` and updates all `throws` declarations in the program accordingly.

Imprecise Exceptions. As previously noted, `throws` declarations can become unintentionally imprecise as code evolves: they may include exception types that are never thrown or types that are too general. (We realize, of course, that sometimes imprecise declarations are an intentional design choice, to provide for future code changes. Our tool allows programmers to retain such declarations.)

When a `catch` block is moved from one module to another, for example, a number of interface methods may include an exception type that they will consequently never throw. New callers of these methods will then have to include handlers for these exceptions—which would be dead code—or must themselves add superfluous exceptions to their `throws` declarations. Such problems do occur in actual code; for example, Robillard and Murphy found a number of unreachable catch blocks in their analysis of several Java programs [19].

To solve this problem, ExnJava includes an Fix Imprecise Declarations refactoring, which can be run on a module or set of modules. The refactoring first lists the exception types which appear in imprecise declarations; the programmer chooses an exception type from this list. The exception type is chosen first so that the view can show the propagation of this exception declaration. For this exception, the view displays all methods where that type appears in an imprecise declaration. The view displays a call graph tree (similar to that of the Propagate Throws Declarations refactoring) showing the propagation of imprecise declarations. This allows the programmer to determine the effect of fixing (or not fixing) a particular imprecise declaration. Initially all methods are checked, indicating that their declarations will be updated; the programmer can choose to not change the declarations for particular methods by unchecking them. (We chose this design as we hypothesize that most imprecise declarations are out-of-date rather than intentional design choices.) The view ensures that a consistent set of methods is chosen; if a method is unchecked, all of its transitive callers will also be unchecked.

Our tool could be extended to include a "Fix Imprecise" refactoring at the module specification level, to inform the programmer of specifications that may no longer be valid. Such a tool would display each module whose specification lists one or more exceptions that are not actually thrown in the implementation.

4.4 Empirical Results

We analyzed six open-source programs to determine the feasibility of interface method and module specifications; results are in Table 1. Even with the rough estimation of each package as its own module, we found that when the declarations of internal methods are inferred there are considerable annotation savings.

We first refactored the visibility modifiers of methods, making them as restrictive as possible. This was to simulate a good module design that hides as many implementation details as possible. (Of course, it is likely that some of the methods that were not currently used outside their package were intended to be accessible for future use, but we hope that this inference provides a reasonable estimate.) We found that after refactoring, the `throws` inference results in a 50% to 93% reduction in declarations. Also, since many imprecise declarations appeared on internal methods, inference reduces imprecision by 42% to 78%. (That is, internal methods contained 42% to 78% of all imprecise declarations.) Before refactoring, 12% to 56% of declarations were inferable.

Using modules would very likely increase these annotation savings, since we expect that most modules will consist of several packages. In such a case, there likely would be more internal methods (and therefore more inferred declarations).

We also computed the average number of exceptions thrown by the interface methods of packages in our subject programs. In most applications, packages generally throw few distinct exception types—fewer than 2 exceptions per package, on average. This strongly suggests that module exception specifications have a low annotation overhead.

Table 1. The subject programs studied, number of lines of code, percentage of declarations that could be inferred (i.e., appeared on internal methods) before and after refactoring to reduce visibility of methods, percent reduction in imprecise exceptions after refactoring (i.e., percentage of imprecise exceptions that appear on internal methods), and average number of exceptions types thrown by the interface methods of packages.

	LOC	Inferable decls		Imprecise reduction	Exceptions thrown per package
		Before refactoring	After refactoring		
LimeWire	61k	45%	72%	53%	2.1
Columba	40k	44%	50%	42%	1.3
Tapestry	20k	12%	75%	68%	0.45
JFtp	13k	44%	93%	59%	0.88
Lucene	10k	56%	81%	75%	1.9
Metrics	7k	23%	72%	78%	0.5

5 Acknowledgments

We would like to thank David Garlan and James Hendricks for their comments on an earlier version of this paper, and Bill Scherlis for his suggestions and discussions.

This work was supported in part by NASA cooperative agreements NCC-2-1298 and NNA05CS30A, NSF grant CCR-0204047, and the Army Research Office grant number DAAD19-02-1-0389 entitled "Perpetually Available and Secure Information Systems".

References

[1] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The Spec# programming system. In *Cassis International Workshop, Ed. Marieke Huisman*, 2004.

[2] Joshua Bloch. *Effective Java*. Addison-Wesley Professional, 2001.

[3] Byeong-Mo Chang, Jang-Wu Jo, Kwangkeun Yi, and Kwang-Moo Choe. Interprocedural exception analysis for Java. In *Proceedings of the 2001 ACM Symposium on Applied Computing (SAC '01)*, pages 620–625. ACM Press, 2001.

[4] Christophe Dony. A fully object-oriented exception handling system: rationale and Smalltalk implementation. In *Advances in exception handling techniques*, pages 18–38, New York, NY, USA, 2001. Springer-Verlag New York, Inc.

[5] Bruce Eckel. *Thinking in Java, 3rd edition*. Prentice-Hall PTR, December 2002.
[6] C. Flanagan, K. Leino, M. Lillibridge, C. Nelson, J. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of PLDI 2002*, 2002.
[7] Alessandro F. Garcia, Cecília M. F. Rubira, Alexander B. Romanovsky, and Jie Xu. A comparative study of exception handling mechanisms for building dependable object-oriented software. *Journal of Systems and Software*, 59(2):197–222, 2001.
[8] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) Language Specification, The (3rd Edition) (Java Series)*. Addison-Wesley Professional, July 2005.
[9] Anson Horton. Why doesn't C# have exception specifications? Available at http://msdn.microsoft.com/vcsharp/ team/language/ask/exceptionspecs.
[10] Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 327–353, London, UK, 2001. Springer-Verlag.
[11] Joseph R. Kiniry. Exceptions in Java and Eiffel: Two extremes in exception design and application. In *Proceedings of the ECOOP 2003 Workshop on Exception Handling in Object-Oriented Systems*, 2003.
[12] Jorgen Lindskov Knudsen. Fault tolerance and exception handling in BETA. In *Advances in exception handling techniques*, pages 1–17, New York, NY, USA, 2001. Springer-Verlag New York, Inc.
[13] K. Rustan M. Leino and Wolfram Schulte. Exception safety for C#. In *SEFM*, pages 218–227. IEEE Computer Society, 2004.
[14] Martin Lippert and Cristina Videira Lopes. A study on exception detecton and handling using aspect-oriented programming. In *Proceedings of the 22nd International Conference on Software Engineering (ICSE '00)*, pages 418–427. ACM Press, 2000.
[15] Robert Miller and Anand Tripathi. Issues with exception handling in object-oriented systems. In *ECOOP*, pages 85–103, 1997.
[16] Darell Reimer and Harini Srinivasan. Analyzing exception usage in large Java applications. In *Proceedings of the ECOOP 2003 Workshop on Exception Handling in Object-Oriented Systems*, 2003.
[17] Martin P. Robillard, May 2005. Personal communication.
[18] Martin P. Robillard and Gail C. Murphy. Designing robust Java programs with exceptions. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE '00)*, pages 2–10. ACM Press, 2000.
[19] Martin P. Robillard and Gail C. Murphy. Static analysis to support the evolution of exception structure in object-oriented systems. *ACM Trans. Softw. Eng. Methodol.*, 12(2):191–221, 2003.
[20] Saurabh Sinha, Alessandro Orso, and Mary Jean Harrold. Automated support for development, maintenance, and testing in the presence of implicit control flow. In *Proceedings of the 26th International Conference on Software Engineering (ICSE '04)*, pages 336–345. IEEE Computer Society, 2004.
[21] Bill Venners. *Interface Design: Best Practices in Object-Oriented API Design in Java*. Available at http://www.artima.com/interfacedesign, 2001.
[22] Bill Venners. Failure and exceptions: a conversation with James Gosling, Part II. Available at http://www.artima.com/intv/solid.html, September 2003.
[23] Bill Venners and Bruce Eckel. The trouble with checked exceptions: A conversation with Anders Hejlsberg, Part II.
Available at http://www.artima.com/intv/handcuffs.html, August 2003.

Invited Lecture

Exception Handling: The Case Against

Andrew P. Black
Portland State University, USA

Abstract

In the early 1980s, Andrew Black wrote his doctoral dissertation at Oxford University; the title was "Exception Handling: The Case Against". The thesis was in part a reaction to the growing complexity of language design, epitomized by the contemporary proposals for what became Ada. The case that the thesis made against exception handling was that (a) exceptions are not an abstract concept capable of rigorous definition, but a subjective classification of program behaviour, and that (b) such a classification could usually be carried out by general-purpose language constructs more effectively than by a special purpose exception handling mechanism.

In this talk, Professor Black, will re-examine this argument in the light of more than twenty years of experience, and attempt to convince you that the remainder of this workshop should be cancelled. Ample time will be allowed for questions and discussion

Exception Handling Issues in Context Aware Collaboration Systems for Pervasive Computing

Anand R. Tripathi* Devdatta Kulkarni, and Tanvir Ahmed

{tripathi, dkulk, tahmed}@cs.umn.edu
Department of Computer Science,
University of Minnesota, Minneapolis, MN 55455 U.S.A.

Abstract. The focus of this paper is on identification of failure cases in building specification driven context aware collaborative applications. We first present a model for building context-based collaborative systems in pervasive computing environments from high level specifications. We identify *User Participation Failures in Roles*, *Coordination Failures*, *Obligation Failures*, *Resource Discovery and Binding Failure*, and *Resource Access Failure* as the failure categories arising in this approach. Exception based approaches to handle these failures are also presented in the paper.

1 Introduction

There is a growing interest in building pervasive computing environments that allow mobile users to seamlessly access their computing resources to perform their activities while moving across different computing domains and physical spaces. A typical user is generally involved in many activities such as office workflow tasks, distributed meetings, collaborative tasks, personal activities such as shopping or entertainment. The resources required for it generally depend on the user's context [1] such as physical location, network domain, time of day, proximity to other users or some specific objects. Moreover, the resource access may be controlled by dynamic security policies that depend on the context.

The nature of collaborative interactions among users can range from ad hoc and unstructured to highly structured and coordinated. Moreover, many times a group of individuals or autonomous organizations may need to form virtual organizations for some common mission. User privileges to access shared resources and perform tasks generally depend on their roles [2] in the activity as well as their current context [1].

We have developed a programming framework for building context-aware applications in pervasive computing environments for supporting collaborative interactions among users and system-level ambient agents [3, 4]. In this framework, context-aware collaborative applications are built from their high level specifications expressed in XML and realized through a distributed middleware [5]. The

* This work was supported by NSF grant 0411961

specification model provides the abstraction of roles for users and environmental agents to participate in an activity. This specification model is essentially a composition framework for integrating users, application-defined components, and infrastructure services to build the runtime environment of a collaborative application. From the specification of an activity, the middleware derives policies for role-based user interactions, policies for dynamic resource binding and access control, and integrates these policies with the application level components and system level generic components to construct the runtime environment.

The focus of this paper is on the exposition of exception handling issues in such environments. We discuss here the nature of the various kinds of causes for exceptions that can arise in a collaboration activity. In the past, other researchers have discussed exception handling problems in collaboration and workflow system [6] and proposed modeling of dynamic structural changes in the workflow environments [7] . Most of those issues are also relevant in our environment. Moreover, we also outline here issues that arise due to security and coordination, and the issues that are related to dynamic discovery and binding of resources. We divide the causes of exceptions in the following categories: user participation failures in a role, failures in dynamic resource binding, resource access exceptions, and exceptions raised by application objects and services integrated through the collaboration framework. We provide an exposition of these issues in this paper. In our current research we are exploring solutions for these problems.

Section 2 describes our specification model along with a set of examples of context-based dynamic resource binding and access control. In Section 3 we discuss the exception handling issues in this model and propose extensions to the our current specification model in Section 4.

2 Specification Model

We present an overview of the collaboration specification model which we have developed [4, 3] . The exception handling issues are discussed in the context of this model. In our collaboration model, an *activity* defines how a group of users cooperate towards some common objectives by performing tasks involving shared resources and infrastructure services. A user joins one or more roles in the activity, and a role represents authorization of its members to invoke a set of operations representing tasks in the collaboration space. Therefore, admission of users into a role needs to be controlled according to the security policies.

The shared resources/services required by an activity may be discovered in the environment and bound to the objects defined in the activity. Moreover, resources may also be created within an activity or passed as parameters to it. Within an activity, the policies for coordination among participants as well as binding and access control of shared resources/services may depend on context-based requirements.

A *role* defines a set *operations* that are executed by role members. A role operation may involve execution of some actions on objects defined within the activity. A role operation can only be invoked by a member in the role. A role

Activity activityName
{**Parameter** objName}
{**Object** [**Collection**] *objName* **RDD** *rddSpec* }
{**Bind** *Binding-Definition*}
{**Reaction** *Reaction-Definition*}
{**Role** *Role-Definition*}

Fig. 1. Activity Syntax

operation can have precondition that must be satisfied before the operation is executed. Within an activity some operations may need to be automatically executed when certain conditions become true. Such operations are termed *reaction*. Context-based access control policies and coordination constraints are specified as operation preconditions.

We have developed an XML schema for pervasive activity specifications. Here, rather than using XML, we use a notation that is conceptually easy to follow. In Figure 1, the syntax for the XML schema for *activity* definition is shown, where `[ ]` represents optional terms, { } represents zero or more terms, | represents choice, and boldface **terms** represent tags for elements and attributes in XML schema. We have developed an XML schema, termed Resource Description Definition (RDD) which is similar to RDF (Resource Description Framework) and WSDL (Web Service Definition Language) to describe the resource and services required as part of the activity specifications. An RDD for a resource includes the attribute-value pairs describing the resource, the interfaces, and the events that are exported by the resource.

2.1 Specification of Context-Based Conditions

Preconditions are specified for role operations to enforce required coordination and security constraints. Such conditions are expressed in terms of predicates based on events occurring within the activity, role memberships of participants, and query methods of the environmental resources representing external context information.

Internal context-related events are generated by execution of operations and reactions within an activity. Internal events are generated as part of the execution of operations and reactions. These events are represented by the names of the corresponding operation or reaction. Each operation event has two predefined attributes: *invoker* and *time*. All the events related to previously executed operations and reactions represent an event list. The specification model supports various functions on event lists. The count-operator `#` returns the number of events of a given type that have occurred so far, and a sublist of these events can be obtained by applying a selector predicate. For example, the expression `#(opName(invoker=member(Chairperson)))` returns the number of times a member of the *Chairperson* role has invoked the operation called `opName`. Some preconditions may depend on the context and the previous operations executed by the

role member who is currently invoking an operation. We use the pseudo variable `thisUser` in the specification model to identify such a role member.

A boolean function `member(thisUser, roleId)` in the precondition of a role operation checks if the user invoking the operation is present in the specified role. The function `members(roleId)` returns the role member list. Set operations can be performed on role member lists. A *count* operator, `#`, can be applied on a member list. The count of the members in a role is `#(members(roleId))`. Within a role operation, a role member is referred to as `thisRole`.

External context events are generated by the objects in the environment. For example, an activity can specify execution of resource binding directives or reactions when certain events occur in the environment. Examples of such external events include user-presence detection, changes in the physical environment, and notification of resource utilization status. To express preconditions that depend on the context information represented by an object in the environment, a condition can include functions which query the object state. These query interfaces are also declared as part of the RDD. For example, the method *isPresent(userId)* supported by a room object can be invoked to check if a specific user is present in the room. Preconditions can also include functions that query a user's membership in a role.

2.2 Role Specification

The policies for coordination and dynamic security are specified in the form of preconditions associated with role operations and member admission. Figure 2 presents the syntax of a role definition. The objects declared within a role represent a separate namespace created for each member in the role, and binding of these names is performed independently for each member.

Role roleName
{**Object** [**Collection**] *objName* **RDD** *rddSpec* }
{**Bind** *Binding-Definition*}
[**Admission Constraints** *Condition*]
[**Activation Constraints** *Condition*]
{**Operation** *Operation-Definition*}
{**Reaction** *Reaction-Definition*}

Fig. 2. Syntax for role definition

Specification of Admission and Activation Constraints: Associated with each role, there are two types of constraints that are imposed on all role members: *Role admission constraints* must be satisfied when a user is to be admitted to a role. *Role activation constraints* must be satisfied for performing any role operations and reactions. Both these constraints can be based on the history of

the operations previously executed, the context of the user, such as the user's membership in other roles, or state of an object. Consider a *Meeting* activity consisting of a *Accountant* role and a *Chairperson* role. The role admission constraint for the *Chairperson* role in this activity are presented below. There are three constraints in this example: (1) there can be only one member in the *Chairperson* role, (2) at the time of joining this role, the user must be present in the meeting room, and (3) only a member of the *Manager* role can join this role.

Role Chairperson
  Admission Constraints
    #(members(thisRole)) = 0 & room.isPresent(thisUser)
    & member(thisUser,Manager)

Following is an example of role activation constraints in the *Secretary* role in the *Meeting* activity. Based on the specified constraints, a member of the *Secretary* role can perform operations only when at least one member of each of the *Manager* and the *Accountant* roles are present in the *room*.

Role Secretary
  Activation Constraints
    room.isAnyPresent(members(Manager))
    & room.isAnyPresent(members(Accountant))

Operation Specification: Members of a role can perform a role operation only when the operation precondition, if any, is satisfied. As part of a role operation methods are invoked on objects. The action part of an operation may include invocation of methods on shared or private objects in the collaboration space.

The *Accountant* role member can perform the *DisplayFinancialData* operation. In this activity a context-based security policy requires that the accountant is allowed to display financial data only when three constraints are satisfied: (1) the *Chairperson* role must have executed the operation *ApprovePresentation*, (2) the member of the *Accountant* role who performs the operation has to be present in the meeting room, and (3) the person in the *Chairperson* role must also be present in the room. The operation specification with these preconditions is shown below.

Role Accountant
  Operation DisplayFinancialData {
    Precondition
      #(Chairperson.ApprovePresentation) = 1
      & room.isPresent(thisUser)
      & room.isPresent(member(Chairperson))
    Action projector.display(data)
  }

In contrast to an operation, a reaction is not invoked by a user but is automatically executed when certain events occur. Similar to an operation, a reaction is executed only when its precondition is true.

2.3 Binding Specification

Resources and services required as part of the activity can be specified using different *Bind* primitives as shown below.
1. Binding to a new object: The binding primitive with `new` specifies that a new resource of the specified codebase type should be created. For example in a meeting activity, a whiteboard object is created and bound to the name *whiteboard* as:

Bind whiteboard **final With new**(//codeBase/WhiteBoard)

2. Binding to an existing resource through URL: This form of binding primitive with `direct` specifies that the resource identified by the given URL should be used in binding. For example, within an activity the URL of the location service might be well-known.

Bind locationService **With direct** *(//LocationServiceURL)*

3. Binding through discovery: This form of the binding primitive is useful when a resource with a particular set of attributes is needed to be discovered in the environment. In the example below, we present specification of a museum information desk activity. In this activity, the audio channel of user's device needs to be bound with the audio player based on the user's location and also taking into consideration the user's choice of the language. In this example, the *audioChannel* object is re-binded when there is a change in the user's location. Discover primitive used in binding of the *audioChannel* object specifies the location attribute and the preferred language in the *Audio-Channel-Description* to be used during resource discovery.

Activity Museum Infodesk
Object locationService **RDD** Location-Service-Description
Parameter userPreference
Bind locationService **With direct** *(//LocationServiceURL)*
Role Visitor
Object audioChannel **RDD** Audio-Channel-Description
Bind audioChannel **When** locationService.locationChange(thisUser)
With discover *(<location=locationService.getLocation(thisUser),*
language=userPreference.preferredLanguage>)

3 Exception Handling Issues

We identify here the causes of various kinds of exceptions that can arise in a pervasive activity. The broad categories of these causes include failures in user participation in roles, failures in resource discovery and access, and exceptions raised by application level objects and system level resources/services.

3.1 User Participation Failures in Roles

In our previous work we explored use of model checking techniques based on SPIN for static verification of security and coordination specifications of an activity [8]. The task-flow within an activity consisting of sequence of role operations is modeled using PROMELA (a Process Meta Language). The desired security and coordination policies specified as safety and liveness properties of role operations are expressed through LTL (Linear Temporal Logic). SPIN then verifies the required properties over the model. Model checking would detect if a role operation would be never executable. This verification is based on the assumption that role members do not abnormally leave their roles. However, in reality this possibility always exists, i.e. a user either departs or is removed from the role due to administrative actions. This can lead to a number of different kinds of failures within the activity as discussed below.

Coordination Failures: The departure of a member from a role can cause operations in the same or other roles to become inexecutable forever unless some corrective actions are taken. If the departed member was the only member to be ever present in the role, none of the operations of that role would be ever executed in the future. Consider the following workflow example, where a member of the *PurchaseOfficer* role submits an invoice, the *Manager* role executes *ApproveInvoice*, and finally the *Accountant* role executes *MakePayment*. Suppose that the *Manager* role has only one member and that person leaves the role prematurely before approving the invoice. Such a condition needs to be detected and it should then enable an alternate approver role, such as the *Director*, to approve the invoice. Another potential cause of failure in a workflow transaction can occur when security policies require that some specific set of actions in the transaction must be performed by the same person. For example, in a business contract transaction the person who prepares the bid must also be the one to sign the final agreement on the contract. In such cases, any premature departure of the person responsible for such critical set of operations in the transaction would leave that activity in incomplete state unless some corrective actions are taken.

Role Cardinality-Based Failures: Our model allows specification of role activation constraints and operation preconditions that are based on the number of members currently present in the role. For example, the operations of some role may be activated only when the number of users present in the role satisfy some cardinality constraint or when a specific operation has been executed by some given number of distinct members. Such conditions may become potentially unattainable if a member abnormally departs from the role. As in the previous example, here also we need to determine which operations' preconditions are affected by the abnormal departure of a role member.

Corresponding to the above two failure categories we define *RoleMemberException*. There are two central research issues that need further investigation.

First, how to determine which other roles' operations are affected because of the generation of the *RoleMemberException*. Second, what kind of corrective or compensating actions are needed and how should they be supported in the specification model.

Obligation Failures: If a critical operation which is necessary for the successful completion of a collaboration or workflow is never executed, we term it *obligation failure* and generate a corresponding *ObligationException*. An activity might require a guarantee that when the precondition of an operation becomes true, the role operation would eventually get executed. This requires a notion of role operation obligation, wherein a role operation is required (obliged) to be executed within some time interval after the precondition has become true. Specification model needs to be extended to support such obligation conditions.

3.2 Resource Discovery and Binding Failures

Dynamic discovery and binding of resources based on the user or activity context is one of the important aspects of context-aware systems. Failures may be encountered in an activity if the required type of resource cannot be found in the environment by the discovery service, or the specific resource needed by the activity is either unavailable or inaccessible due network failures, or access to it is denied by the security policies. In any such case, appropriate alternate resource binding directives are needed. Corresponding to the failures mentioned above, we identify the following two exceptions. *ObjectBindingException* indicates that the resource binding was not successful. Further exceptions are derived from this exception to indicate reasons of failure. *ResourceDiscoveryException* indicates the failure in finding the exact resource as required by the activity.

3.3 Application Objects and Resource Failures

A role operation may include invocation of a method on some object in the collaboration workspace. It is possible that this method invocation results in an exception. These are termed as *ApplicationExceptions*. *ObjectInvocationException* is a type of *ApplicationException* which occurs when invoking the application object method. The *ObjectAccessException* is an *ApplicationException* indicating access privilege violation. Each event is internally characterized by two sub events, viz. *Event.Start* and *Event.Finish*. Generation of an *ApplicationException* indicates that the role operation's *finish* event was not generated and operation's desired effect was not achieved. Therefore, any other operations whose preconditions depend on the *finish* event should not be executed.

4 Proposal for Exception Handling in Context-aware Collaborative Applications

In this section we propose extensions to the specification model to incorporate specification of exceptions and their handling.

4.1 RoleMember Exceptions

These exceptions convey a role member's abnormal departure from a role. There are two issues which need to be addressed in this regard. First is related to the monitoring of role membership and relaying the *RoleMember Exception* to other appropriate roles using *guardians* [9]. The monitoring of role membership is only required when a task-flow involving the role gets activated. Static analysis can be used to identify the conditions that would indicate starting of a task. The role member monitor can get activated based on such events. These monitors will generate *RoleMember Exception* and would send them to other role managers identified during policy derivation. Role managers can define handlers to handle this exception. An activity wide handler can terminate the activity if the role members critical for the activity's progress have left their roles.

4.2 Obligation Exceptions

An obligation exception occurs when a role operation is not executed within some time interval after the operation's precondition has become true. We specify this requirement by extending the operation specification as shown in Figure 3. The

---

Operation opName
[**Precondition** *Condition*]
[**Action** *objId methodSignature methodParameter*]
[**Obligation** {**Within** *Duration* **After** *Event-of-Interest*}]
[**Exception** {**Case** *ExceptionType*
(**Action** *objId methodSignature methodParameter* | **Retry** *Retry-Parameters*)}]

Fig. 3. Operation Syntax: Modified to incorporate Exception Specification

obligation specification requires that the operation *opName* must be executed within some specified time interval after the event *Event-Of-Interest* has occurred. The *ObligationException* is generated if the operation is not executed within the time interval *Duration*. The *Event-of-Interest* can be any application event or meta-event such as the precondition has become true. Multiple *Event-of-Interests* with specific *Durations* can be specified in the obligation specification. *ObligationException* is generated when any one of these events is not generated within the specified *Duration*. The exception specification consists of the exception name specified by the **Case** tag and the exception handling action specified by either **Action** tag or **Retry** tag.

Consider a *ExamSession* activity consisting of a single *Student* role and two role operations, viz. *StartExam* and *SubmitExam*. In the *ExamSession* activity there might be a requirement that the *SubmitExam* operation must be executed within three hours of executing the *StartExam* operation. This obligation will be specified as shown below.

```
Activity ExamSession
    Role Student
        Operation StartExam {
            Precondition currentTime >= 3.00 pm & currentTime <= 6.00 pm
            Action answerBook.writeAnswers()
        Operation SubmitExam {
            Precondition
                #StartExam == 1
            Action answerBook.submitExam()
            Obligation
                Within (3:00:00) After StartExam.Start
            Exception
                Case ObligationException
                Action answerBook.submitExam()
        }
```

The *ObligationException* is generated if the *SubmitExam* operation is not executed within three hours of executing the *StartExam* operation. The exception clause specifies that on the occurrence of *ObligationException* the answerBook will be submitted. It is also possible to specify an application event to be generated as part of the *Action* clause in the exception specification. This event can then be used to trigger a reaction to perform some recovery actions.

4.3 Object Binding Exceptions

Figure 4 shows the extension to the object binding primitive specification to incorporate specification of exception handling on binding failures. The exception

```
Bind objId [final]
   [When {Event}]
   With [new CodeBase | direct URL | objId | discover ({attribute=value}) ]
        [Exception {Case ExceptionType
        ( Action objId methodSignature methodParameter | Retry Retry-Parameters)}]
```

Fig. 4. Object Binding Syntax: Modified for Exception Specification

handling in case of binding failures depends upon the binding primitive being used and the nature of the binding exception that was generated. For example, in case where the object is to be bound with a *new* resource or where the object is to be bound with a resource whose location is already known (*direct*) and the exception generated is *ObjectBindingException* then there is a possibility that such an exception was generated due to a network failure. The exception handling in this case will be to retry the request for some number of times as shown in the following example.

Bind whiteboard **With new**(//codeBase/WhiteBoard)
Exception
Case ObjectBindingException
Retry *(limit=5)*

On the other hand if the binding is based on resource discovery and if the generated exception is *ResourceDiscoveryException* then the exception handling action may consist of retrying the discovery request by changing the discovery specification. For example, in the *Museum Infodesk* activity, if the audioChannel cannot be bound with English language resource an alternative could be to bind it with Spanish language resource as shown below.

Bind audioChannel **With discover**(language=English)
Exception
Case ResourceDiscoveryException
Retry *(language=Spanish)*

It might happen that the exception handling actions also generate exceptions. In that case the binding is aborted and the object remains unbound. This leads to the issue of how to handle role operations, operation preconditions and reactions that depend on the objects which are not bound. One option is to disable all the role operations and reactions that depend on unbound objects. This is similar to having an implicit precondition for each operation/reaction specifying that the operation/reaction will become active only when the objects are bound. Disabling of role operations might lead to other role operations not getting executed at all during the activity life time.

4.4 Application Exceptions

Application exceptions arise in the context of the role operations that are executing actions on the application objects. For example, an *ObjectInvocationException* can occur while the student is taking the exam. Such an exception can be handled by generating an application event such as *ActionFailedEvent* and enabling the *ResumeExam* operation. As part of *ResumeExam* operation the student can resume taking exam after rebinding with a new *answerBook* object. The obligation requirement for the *SubmitExam* operation is modified to take into account the delay caused by *ObjectInvocationException*. This is shown below.

Operation StartExam {
Precondition currentTime >= 3.00 pm & currentTime <= 6.00 pm
Action answerBook.writeAnswers()
Exception
Case ObjectInvocationException
Action NotifyEvent *ActionFailedEvent*
}
Operation ResumeExam {
Precondition #ActionFailedEvent > 0
Action Bind answerBook **With new** (//codeBase/AnswerBook)

```
        answerBook.writeAnswers()
Operation SubmitExam {
  Precondition
      #StartExam == 1
  Action answerBook.submitExam()
  Obligation
      Within (3:00:00) After StartExam.Start
      Within (3:00:00 - (ResumeExam.Start.Time - StartExam.Start.Time))
}
```

5 Conclusions

The primary contribution of this paper is identification of failure issues for collaborative activities that are immersed in pervasive computing environments. Exceptions related to *User Participation Failures in Roles*, *Obligation Failures*, *Resource Discovery and Binding Failure*, and *Resource Access Failure* are identified. Our earlier specification model [4] is extended to incorporate exception specifications and their handling.

References

1. Dey, A.K.: Understanding and Using Context. Journal Of Personal And Ubiquitous Computing **5** (2001) 4–7
2. Sandhu, R., Coyne, E., Feinstein, H., Youman, C.: Role-Based Access Control Models. IEEE Computer **29** (1996) 38–47
3. Tripathi, A., Kulkarni, D., Ahmed, T.: A Specification Model for Context-Based Collaborative Applications. Elsevier Journal on Pervasive and Mobile Computing **1** (2005) 21 – 42
4. Tripathi, A., Ahmed, T., Kumar, R.: Specification of Secure Distributed Collaboration Systems. In: IEEE International Symposium on Autonomous Distributed Systems (ISADS). (2003) 149–156
5. Tripathi, A., Ahmed, T., Kumar, R., Jaman, S.: Design of a Policy-Driven Middleware for Secure Distributed Collaboration. In: Proceedings of the 22nd International Conference on Distributed Computing Systems (ICDCS). (2002) 393 – 400
6. Hagen, C., Alonso, G.: Exception handling in workflow management systems. IEEE Transactions on Software Engineering **26** (2000) 943–958
7. Ellis, C., Keddara, K., Rozenberg, G.: Dynamic change within workflow systems. In: COCS '95: Proceedings of conference on Organizational computing systems, New York, NY, USA, ACM Press (1995) 10–21
8. Ahmed, T., Tripathi, A.R.: Static Verification of Security Requirements in Role Based CSCW Systems. In: Proceedings of 8th ACM Symposium on Access Control Models and Technologies (SACMAT 2003), New York, ACM (2003) 196–203
9. Miller, R., Tripathi, A.: The Guardian Model and Primitives for Exception Handling in Distributed Systems. IEEE Transactions on Software Engineering **30** (2004) 1008 – 1022

Aligning Exception Handling with Design-by-Contract in Embedded Real-Time Systems Development

Luis E. Leyva-del-Foyo[1], Pedro Mejia-Alvarez[1], and Dionisio de Niz[2]

luisleyva@acm.org, pmejia@cs.cinvestav.mx, dionisio@iteso.mx
[1]CINVESTAV-IPN, Seccion de Computacion,Av. I.P.N 2508,Zacatenco, 07300 Mexico,DF
[2]ITESO, DESI, Periferico Sur 8585, Tlaquepaque, Jal. 45090, Mexico

Abstract. In this paper we introduce an exception handling mechanism, which is part of the kernel of an operating system for embedded applications. Our approach is based on the theory of design by contract and is adapted for the development of embedded real-time systems.

1 Introduction

The cost of mishandling abnormal situations in general purpose computing (e.g. banking applications, ERPs, text editing, , etc.) can be high. For example, it could mean the improper decrement of the balance of an account upon the failure of an ATM, i.e., it could have decremented the balance and not delivered the money. Exception handling is one of the most important control structures of modern languages (e.g. Java, C++, Eiffel) to reduce this kind of errors. This structure has proven to be very useful in modern systems developed in languages such as Java and Eiffel for general purpose computing.

In embedded systems the cost of mishandled errors can be higher than in general purpose computing. For instance, the incorrect release of the brake of a wheel in an ABS system upon the failure of a sensor could put a car to spin with potentially deadly consequences. Even though embedded systems also benefit from exception handling, the criticality of errors in embedded software has motivated a trend in the use of more thorough techniques to ensure the correctness of embedded software. One of these techniques is the theory of design-by-contract [8]. In this theory, modules interactions are regulated by contracts. A contract defines pre and post conditions and invariants. A module then guarantees that the post conditions and invariants are honored if the preconditions and the same invariants hold when it is called. Even though the current implementation of design by contract include the implementation of exceptions such exceptions has not being made part of the contract. Such an omission diminishes the possibility of the automatic verification of such contracts.

Two additional factors further complicate exception handling in embedded systems. On the one hand, the interaction of the operating system is in general tighter than that of a general purpose application, even to the point of being in the same memory space and linked together (the OS and the application). On the other hand, it is common for an embedded system to interact with the environment in a timely fashion with strict requirements on the reaction time. For instance, in a front airbag system, the computer needs to ensure that it does not take more than 20 ms from the detection of a collision

to the triggering of the chemical reaction that inflates the bag [1]. This type of applications known as real-time applications needs to have a deterministic worst-case execution time. This determinism could be compromised by an exception handling mechanism for general purpose computing. This is due to the triggering of a potentially unbounded list of exception handling code when an exception occurs making the worst-case execution time difficult to measure.
In this paper we describe a novel design of a exception handling structure based on the theory of design by contract that: (a) aligns exceptions to module contracts, (b) enables recasting the interaction with the operating system as a contract, (c) provides constructs to interact with legacy code, (d) is implemented and available for use inside the kernel, and (e) provides mechanisms to bound the exception handling to simplify the measurement of the worst-case execution time of the application.

2 Exception Handling Problems in Embedded Real-Time Systems

As introduced in Section 1, embedded real-time systems can greatly benefit from exception handling. In this section we first discuss the shortcomings related to all types of applications and then those related to embedded real-time systems.

2.1 Exception Handling and Encapsulation Misalignment

Exception handling is a special purpose control structure that enables the separation of the normal code of the software from the abnormal one. By abnormal code we mean code that does not contribute to the behavior the software is required to exhibit but to corrections to deviations from such behavior. For instance, consider the example of a deposit operation to a checking account. The required behavior is to increment the balance of the account by a specified amount. An example deviation that must be controlled is the failure of the disk where the balance is stored. For such case, special code must be added to take care of such situation (e.g., with a retry). The deviating situations are known in programming languages as *exceptions* and the code that handles them as *exception handling code*.
Both, programming control structures for normal code (e.g. *if*s, *while*s, *switch*, etc) and exception handling structures define code blocks around which some control flow is defined. For instance, in an *if-else* structure two code blocks are defined, one is executed if the condition specified in the *if* sentence is true and the other if such a condition is false. Exception handling, on the other hand, defines two types of blocks we identify as: *exception guarded* and *exception handling* blocks. The exception guarded block is where an exception can occur. Such an exception is a condition that a line of code in this block could discover, e.g., a write to disk that failed. Upon the discovery of this condition a sentence to transfer the control to an exception handling block is used. This sentence is known as an *exception raising* sentence. The exception handling code can either finish its execution and continue the execution of the next

sentence after the end of this block or transfer the control to yet another exception handling block or even return to the original exception guarded block.

2.1.1 Encapsulation Corruption

Both control and exception handling structures can be recursively composed. For instance, we can define an *if* block inside a while, switch, or another if block. It is also possible to embed normal control structure blocks inside exception blocks and the other way around. In the end, a fundamental property of both of these types of blocks is encapsulation. By encapsulation we mean two things. On the one hand, that they have a single entry (where execution starts) and exit (where execution continues after it is done with the block) points. On the other hand, that the entry point has a description that enables a programmer to use such a block without knowing its internals. Such a description is both a syntactic description used by the compiler to generate code and a semantic description used by the programmer to understand how to use it[1]. Unfortunately, the encapsulation property gets compromised when normal and exception handling structures are mixed. This is because if a normal control block is encapsulated into an exception guarded block then the *if* control block would have two exit points. For instance, in our deposit example if the code to increment the balance is embedded in an *if* to verify the account number which in turn is embedded in a exception guarded block to correct failed writings to disk, then the if block would have a normal exit point at the end of its block and an exception exit point that transfers the control to the exception handling block. We identify this problem as an *encapsulation corruption* problem.

The encapsulation corruption is, in general, tolerated in control structures. This is because control structure blocks are not opaque, i.e., the detail code of the block is collocated with the code that uses it. In contrast, when modularity structures (procedures and functions) are used, the encapsulation corruption creates important problems. The reason is that the details of the code block is in a different place than the code that uses it (calls it). As a result, the embedding relationship of the modules and the exception structures can be hidden. In particular, determining where the module would return can be difficult.

2.1.2 Encapsulation Corruption in Design-by-Contract

The power of modular encapsulation has been used in a more formal manner in an approach known as design-by-contract [8]. In design-by-contract modules are used along with enforced use contracts. A use contract is composed of three parts: preconditions, invariants and postconditions. A precondition specifies a Boolean expression that must be true before calling the module. An invariant is a Boolean expression that must always be true (before, during, and after the execution of the module). Finally, a postcondition is a Boolean expression the module guarantees to be true after its execution provided that both the preconditions and the invariants were true before its execution. With these contract specifications it is possible to verify the integrations of modules into a system and the properties the system can guarantee.

[1] This semantic description is added as comments, separate documentation and sometime is not present at all.

Given that the design-by-contract methodology relies on the encapsulation to enable sound verification, the corruption of the encapsulation diminishes its utility.

2.2 Exception Handling Shortcomings for Embedded Real-Time Systems

Two shortcomings are specially related to embedded real-time systems: Exceptions across the application and OS boundary and the temporal predictability.

2.2.1 Exceptions Across the Application and OS Boundary

Operating systems provide a set of services to applications that simplifies the computer programming. To ensure the generality of such services (serving the purpose of a variety of applications), options should be given to the applications to customize both their execution and the interpretation of their results. An important part of this interpretation is, hence, the exceptions that can occur during the service execution. For instance, failure to write a file could be treated differently by unattended (e.g. overnight production plan calculations) and interactive (e.g. word processor) applications. In particular, the unattended application may only have the option of aborting the application. In contrast, the interactive application could ask the user for another file path that could include even a different drive. In addition, operating systems need to support multiple languages and hence cannot rely on a single programming language mechanism to handle exceptions. For this reason, it is still a common practice to transform exceptions into return codes that can be easily ignored by the programmer.

Embedded applications have a tighter relation with the operating system. On the one hand, given the limited resource of embedded processors, it is common to link together the application and the OS into a single program. On the other hand, embedded applications are, in general, devices with a single application. This implies that if the application crashes, the whole system crashes (as opposed to a desktop application where a word processor, for instance, could be restarted). This relationship implies that the cost of ignoring exceptions is higher and the consequence of an error in the application can jeopardize the whole system.

2.2.2 Temporal Predictability

Real-time systems must have predictable response times. This implies that their worst-case execution time must be bounded. The state-of-the-practice to find the worst-case execution time is to measure multiple times the execution of the tasks and get the worst measurement. Variations to the execution time are due in part to different paths modules take in the execution of the code. Exceptions can further complicate these paths, potentially leading the execution to a large chain of exception handling blocks that, instead of correcting an exception, can induce timing errors. As a result, it is important to limit the exception handling chaining to avoid creating a timing failure.

2.3 Current Solutions

Different positions have been taken to solve the encapsulation corruption problem related to modularity structures. On the one hand, some languages such as Java [4], enforces *checked exceptions* [5] at compile time. That is, modules (methods in Java) are forced to either capture all exceptions that can be produced by its code or declare the method as a thrower of such exceptions (in the method declaration). Any user of the method would then need to capture the exceptions or recursively declare itself as a thrower of them. Even though checked exceptions prevent ignoring exceptions, not everybody agrees on their benefits. The designers of C# [6] for instance, argue that checked exceptions produces the programming habit of capturing generic exceptions with empty exception handling blocks.
For the design-by-contract framework, Meyer [8] proposes an approach where a module can either finish its execution successfully or with a failure when its contract cannot be satisfied. Such a failure is communicated with an exception. However, the interpretation of exceptions exclusively as failures defies the purpose of the exception structure. That is, the exception handling block cannot serve the purpose of controlling the exception and continuing the execution.
To deal with exceptions within the operating system, some operating systems such as VMS [3], OS/2 [7], and Windows NT [10] have offered support for exceptions. However, no uniform interaction with different programming languages is provided.

3 Design of our Exception Mechanism

In this section we discuss the design of our exception handling mechanism. Our mechanism is designed to be included in the kernel (and was implemented in a kernel) enabling the development of multiple language interfaces. Our initial implementation was developed along with an interface for the C language given its popularity in embedded systems development.

3.1 Basic Design

The base of our exception handling mechanism is composed of an exception guarded block, an exception handling block, an exception raising function, and an exception propagation cancellation function. The construct in the C programming language has the following structure:

```
01:  TRY
02: { /* guarded code */
03:  if (<exception condition>)
04:   RAISE(code, parameter);
05: }
06:  UNLESS
07: {
08:  if (EXCEPTION == code)
09:   { /* exception handling code */
```

```
10:     …
11:     _ABORT(retcode);
12:   }
13: }
14: _END
```

In this construct, the guarded block is identified with a _TRY statement and delimited with braces. This block is followed by an exception handling block that is identified by an _UNLESS statement. Finally, the whole exception block is closed with an _END statement. The _RAISE() function allows a program to raise their own *exception codes* after detecting some abnormal conditions. When an exception is raised inside the _TRY section, the system invocate the code inside _UNLESS section to handle the exception. Then, the program's control flow continues after finishing the _END sentence. Besides this, if the code of the _TRY section is executed without producing any exception, the code of _UNLESS section is left unexecuted, to continue to the code after the _END sentence. The RAISE() sentence includes two arguments, the *code* of the exception and an argument, subject to the interpretation of the applications, that can be a pointer to any complex data type or object.

Contrary to other exception handling blocks, e.g. Java exceptions, the _UNLESS block does not specify the exception it is suppose to handle. Instead the exception code can be checked against the macro EXCEPTION to execute the proper code.
If the exception is properly handled an _ABORT function is used to cancel the exception, meaning that it was successfully handled. In such a case, the execution continues after the _END statement. If, on the other hand, no _ABORT statement is executed the exception is considered unhandled and continues its propagation to other embedding[2] exception handling blocks. We identify this semantics as *propagation-by-default* semantics. It is important to note that in this construct any exception can be identified by an integer code and may have a parameter which can be retrieved with the EXCEPARAM variable.

3.2 Preventing Modular Encapsulation Corruption

To prevent modular encapsulation corruption, we use a twofold scheme. On the one hand, we provide multiple control flow options to terminate the exception handling blocks to support multiple interpretations of exceptions. On the other hand, a mechanism to align the module and exception exit points is also provided.

3.2.1 Enabling Double Exception Semantics

The basic design described in Section 3.1, provides a control flow that facilitates the orderly termination of the guarded block. This semantics delegates the interpretation of the exception to the calling code enhancing the flexibility of the guarded code. However, when the code to be guarded include the full body of a module there could

[2] An exception guarded block can be embedded into other exception guarded blocks recursively as discussed in Section 2.

be exceptions that need to be resolved within the module while others would need to be propagated outside it. To facilitate the resolution of exceptions inside a module our design also provides retry semantics. The retry semantics is build by calling the _RETRY function with a retry code once the exception has been handled in the _UNLESS block. This statement transfers the control back to the beginning of the _TRY block. An example of the retry semantics follows.

```
01: _TRY
02: { /* guarded code */
03:  …
04:  if (_RETRYCODE)
05:   /* Code for retry */
06:  if (<exception condition>)
07:   RAISE(code, parameter);
08: }
09: _UNLESS
10: {
11:  if (EXCEPTION == code)
12:  { /* exception handling code */
13:    …
14:    _RETRY(1);
15:   }
16: }
17: _END
```

In this example, the _UNLESS block has a _RETRY statement with a retry code equal to one in line 14. This _RETRY sentence transfers the control back to the start of the block in line 2 and the retry code is checked with the variable _RETRYCODE to perform special code for the retry. The support for the double semantics – orderly failure and retry – promotes the separation of exception handling code internal to a module from the code that must be delegated to the module's caller.

3.2.2 Aligning Exception and Module Exit Points

The alignment of the exception and module exit points is handled by creating a code block that is always executed whether or not an exception occurs. This block is known as the _FINALLY block. This block is intended to encapsulate the finalization code of a module ensuring that such code will be executed when the control is transferred outside the module. In other words, the _FINALLY block represents a single exit point for both the module and the exception handling block**,** given that both exit paths, the normal termination and the exception termination executes it. An example of such construct follows:

```
01: _TRY
02: {
03:  /* guarded code */
04:  …
05:  if (_RETRYCODE)
06:   /* Code for retry */
07:  if (<exception condition>)
08:   RAISE(code, parameter);
09: }
10: _UNLESS
```

```
11: {
12:  if (EXCEPTION == code)
13:   {
14:    /* exception handling code */
15:    …
16:   }
17: }
18: _FINALLY
19: {
20:  /* Finalization code: free resource, etc.*/
21: }
22: _END
```

In this example, whether an exception occurs or not, the _FINALLY block from line 19 to 21 executes. In our design, the _FINALLY block must include the cleanup actions that must be executed regardless of the occurrence of exceptions. We do not keep a record of the locks retained by any section of code, and in consequence, the programmer is responsible for the release of any locks within the _FINALLY block.

Combining _FINALLY and _RETRY.

The _FINALLY block not only is executed with the orderly-failure semantics of exceptions but also with the retry semantics. This implies that every time we transfer the control back to the _TRY block with the _RETRY statement the _FINALLY block will be executed. This control flow is designed to keep a transactional model of the block where resources (e.g. locks) are acquired at the beginning of the _TRY block and release in the _FINALLY block. As a result, when using the _RETRY statement the resource would be released in the _FINALLY block and reacquired in the _TRY block.

3.3 Exceptions across the application and OS boundary

Given that our exception mechanism is implemented in the kernel, it can be used in the kernel itself. The propagation of an exception that occurs in the kernel toward the application depends on how the OS and the application are related. In embedded systems this relationship can be either tight or loose. By tight we mean that both the OS and the application are linked in the same memory space as a single program (e.g. μC/OS-II and OSEK/VDX). On the other hand, by loose we mean that the OS and the applications are linked as separate programs and the OS loads the application programs (the common model in desktop computers and larger systems).

When the OS and the application are tightly related, the exception handling mechanism can be used without modification and can be used even with contracts. However for legacy applications expecting return codes, the exceptions occurring in the OS must be translated into such codes. Our design includes a _TRYERROR block to wrap system calls to translate exception into return codes. This block is used as follows:

```
01:  int osService() /*Internal code of a system call*/
02: {
03:       _TRYERROR {
04:       osServiceCode();
```

```
05:         }
06:         _END
07:         return;
08: }
```

A layer of wrappers of this form is then used to support legacy applications that cannot handle exceptions. When the applications and the OS are loosely related, exceptions are not used. Instead such exceptions are translated into error codes with the _TRYERROR block. On the other hand, the termination of applications that uses our exception handling mechanism has to ensure that no exception is left unhandled. For this purpose a *default handler* is included in every task in the operating system. This handler can be used for the case when: 1) an exception is raised in some task (process) for which no block _TRY/_UNLESS/_FINALLY/_END has been established, or 2) none of the handlers from the nested blocks provides treatment to the exception. This default exception handler is declared using the same exception mechanism (_TRY/ _UNLESS/_END block). When this wrapping block traps an exception, it translates it into a return code to make it compatible with the legacy applications (that expect error codes).

3.4 Supporting Real-Time Applications

Real-time applications demand predictable timing behavior. In embedded applications with timing constraints not only reliability and safe operation is demanded but also the ability to provide guarantees for timing behavior. As explained before, this implies that the worst-case execution time of the exception handler must be bounded.
Exception handlers are not predictable because their propagation rules allows them to propagate exceptions as much as required throughout an unpredictable large chain of exception handling blocks. In our design, we included dynamic propagation of exceptions, but using the mechanism in a restricted form, for applications with timing requirements. We establish by means of configuration, a bound in the depth of nesting (*nesting limits*) and in the number of active handlers, which indirectly provides a bound on the worst-case propagation time of the exceptions. This scheme can be statically analyzed with techniques such as those presented in [11].

3.4.1 Exceptions and Design-by-Contract Alignment

In the original design-by-contract semantics, a contract is fulfilled if all the preconditions, postconditions, and invariants are honored through the module execution. This original semantics specifies that if an exception occurs then the contract is considered broken. In this case, two semantics are offered, either repair the broken conditions and retry or shutdown the system gracefully.
Our design supports both semantics but in addition, it enables exceptions to exit module boundaries within the contract agreement. For instance, this could be honoring preconditions and invariants and so the caller module could retry calling this same module (or calling another) or even doing reparable (by the caller module) damage to preconditions. In summary, our construct enables a triple semantics for

exceptions in the context of design-by-contract: failure organized failure, retry, and reparable failures. These semantics are explained next.

Organized Failure: In the organized failure semantics the preconditions, and invariants are guaranteed upon the occurrence of the exception. However, the selected course of action is the orderly termination of the program, sending the exception outside the module.

Retry: Upon the evaluation of the conditions in an _UNLESS block (and potential corrective actions) it can be determined that both the invariants and the preconditions are still valid and a retry is possible. In such a case the _RETRY sentence can be used to retry the execution of the _TRY block.

Reparable Failures: In this case the occurrence of the exception could still honor preconditions and invariants or cause the preconditions to be invalid. However, provided that the calling code knows how to restore broken conditions the exception can be propagated to this code for a potential corrective action to keep the system running. In this case the alignment of modules and exception allows the common _FINALLY block to clean up any acquired resources (e.g. mutexes) to be released.

These two last semantics align exceptions with contract by keeping corrective actions within the limits of the contract (retry) or making the exception propagation as a valid exit option within the contract (reparable failures).

4 Comparative Study of Exception Handling Mechanisms

The aim of this section is to compare our exception handling mechanism against other well known mechanisms. This comparison is based on a previous comparative work from Garcia et-al [2]. In this work, Garcia et al identify common design features of exception handling mechanisms and compares different design solutions against them. These features are presented next.

Exception representation: Different choices exist to encode an exception. Exception representations can be classified as (a) symbols, (b) data objects or (c) full objects. Our mechanism supports symbols with parameters.

Checked exceptions: Two approaches are taken related to exception checking enforcement: checked exceptions and runtime exceptions as discussed in Section 2.3. Our mechanism does not support checked exceptions.

Clean Separation of concerns based on design by contract: Only Eiffel and our mechanism support a clear separation of concerns based on design by contract. Specifically, our mechanism aligns exceptions to module contracts enable recasting the interaction with the operating system as a contract.

Attachment of handlers: Exception handlers can be attached to different guarded regions, such as a). a statement, b). a block, c). a method, d). an object, e). a class, or f). an exception. In our mechanism exception handlers are attached blocks.

Propagation of exceptions: If a handler does not handle a specific exception, then an external handler is searched to handle the exception. This mechanism is known as *exception propagation*. In our design, the exception propagation is controlled by dynamically chaining guarded blocks. This chain is bounded in depth with a nesting

limit parameter. This parameter limits the number of active handlers (to limit the execution time of real-time applications).

Continuation of the control flow: After an exception is raised and the corresponding handler is executed the control cannot resume at the point of the exception. As explained before, our mechanism supports the termination and retry semantics.

Resource cleanup is a requirement of atomic transactions, which are important in many concurrent and real-time systems. For example, if an exception occurs within the critical section of a routine, it should release the critical section. Otherwise other processes wishing to use the critical section will be blocked indefinitely. We extended the exception with the _FINALLY code block that executes whether an exception occurs or not. This allows us to not keep a record of the locks retained by any section of code providing the programmer with a block where the locks can be released appropriately.

Concurrent exception handling: When concurrent exception handling is supported one or more exceptions can be raised concurrently. Our mechanism provides limited support for concurrent execution by automatic signaling to a supervisor task the termination of tasks by exception.

Table 1 shows the design aspect supported by Ada 95, C++, Java, Eiffel and our mechanism and Fig. 1 illustrates the complete semantics of our mechanism.

Taxonomy Aspects	Design Decisions	Ada 95	C++	Java	Eiffel	Ours
Exception Representation	Only Symbols	x				
	Symbols with parameter				x	X
	Objects		x	x		
Checked Exceptions	Unsupported	x			x	X
	Optional		x	x		
	Hybrid			x		
Clean Separation of concerns based on design by contract	Unsupported	x	x	x		
	Supported				x	X
Attachment of Handlers	Block	x	x	x		X
	Methods				x	
	Class	x			x	
Propagation of Exceptions	Automatic				x	
	Configurable					X
	Explicit	x	x	x		
Continuation of Control Flow	Termination	x	x	x	x	X
	Retry				x	X
Clean-up Actions	Explicit Propagations	x	x		x	
	Semi Automatic Clean-up			x	x	
	Specific Construct			x		X
Concurrent Exception Handling	Unsupported		x		x	
	Limited	x		x		X

Table 1. Comparison of Exception Handling Mechanism.

```
Int myCode(){
   _TRY { /* Protected Code Section */
    <  Program code (protected section).
       T1.- raise exception [RAISE(code,parameter)]
       T2.- verify retry identifier [RETRYCODE] >
    }
    _UNLESS { /* Exception Handling Code */
    <  Exception handling code:
       U1.- identify the exception code [EXCEPTION]
       U2.- obtain exception parameter [EXCEPARAM]
       U3.- retry protected code[RETRY(code)]
       U4.- abort operation [ABORT(code)]
       U5.- propagate the exception [default option] >
    }
    _FINALLY { /* Termination Code Section */
     <Termination Code>
    }
    _END
    return;/* return the protected block exit code */
   }
```

Fig 1. Proposed Scheme for Exception Handling.

5 Concluding Remarks

In this paper we presented a novel exception handling mechanism implemented in the kernel that supports multiple languages interfaces. This mechanism was developed along with a *C* language interface. We also discussed how our mechanism prevents the corruption of modular language structures (e.g. procedures and functions) when used along with exception handling blocks. This feature is aligned with the theory of design-by-contract enabling new semantics to support including exceptions as a recoverable exit path within contracts. We finally compared the multiple features of our mechanism with other well-known exception implementations.

References

[1] "How Airbags Work". http://auto.howstuffworks.com/airbag1.htm, as of 05/12/2005.

[2] A. F. García, C. M. F. Rubira, A. Romanovsky, J. Xu, "A comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software". *Journal of Systems and Software*. 59, pp. 197-222, 2001.

[3] R. Goldenberg,S. Saravanan,"*OpenVMS AXP Internals and Data Structures*",Digital Press, 1994.

[4] J. Gosling, B. Joy, y G. Steele. *The Java Language Specification*, Addison- Wesley, 1996.

[5] "Failure and Exceptions: A Conversation with James Gosling, Part II". Artima Developer, September 2003.

[6] "The Trouble with Checked Exceptions: A Conversation with Anders Hejlsberg, Part II". Artima Developer 18 de August 2003.

[7] G. Letwin, "*Inside OS/2*". Microsoft, Press, Redmond, Wa, 1988.

[8] B. Meyer, "*Object-Oriented Software Construction*", Prentice-Hall, 2nd Edition, 1997.

[9] B. Meyer, "*Eiffel: the language*", Prentice Hall Object-Oriented Series, 1992.

[10] D. A. Solomon "*Inside Windows NT Second Edition*". Microsoft Corporation, 1998.

[11] R. Chapman, A. Burns and A. Wellings, "Worst-Case Timing Analysis of Exception Handling in Ada". *Proc. Ada UK. Conference*, London 1993.

A Quantitative Study on the Aspectization of Exception Handling

Fernando Castor Filho and Cecília Mary F. Rubira

Institute of Computing - State University of Campinas
P.O. Box 6176. CEP 13083-970, Campinas, SP, Brazil.
{fernando, cmrubira}@ic.unicamp.br

Alessandro Garcia

Computing Department - Lancaster University
South Drive, InfoLab 21, LA1 4WA, Lancaster, UK.
garciaa@comp.lancs.ac.uk

Abstract. It is usually assumed that the implementation of exception handling can be better modularized by the use of aspect-oriented programming (AOP). However, the trade-offs involved in using AOP with this goal are not yet well-understood. To the best of our knowledge, no work in the literature has attempted to assess whether AOP really promotes an enhancement in well-understood quality attributes other than separation of concerns, when used for modularizing non-trivial exception handling code. This paper presents a quantitative study of the adequacy of aspects for modularizing exception handling code. The study consisted of refactoring part of a real object-oriented system so that the code responsible for handling exceptions was moved to aspects. We employed a suite of metrics to measure quality attributes of the original and refactored systems, including coupling, cohesion, and conciseness. We found that AOP improved separation of concerns between exception handling code and normal application code. However, contradicting the general intuition, the aspect-oriented version of the system did not present significant gains for any of the four size metrics we employed.

1 Introduction

Aspect-oriented programming (AOP) [9] has been proposed recently as a means for modularizing systems that present crosscutting concerns. A crosscutting concern can affect several units of a software system and usually cannot be modularized by traditional object-oriented programming techniques. A typical example of crosscutting concern is logging. The implementation of this concern should be scattered across all the modules in a system, tangled with code related to other concerns, because some contextual information must be gathered in order for the recorded information to be useful. Other common examples of crosscutting concerns include profiling and authentication [11].

It is usually assumed that the exceptional behavior of a system is a crosscutting concern that can be better modularized by the use of AOP [9, 11, 12]. The most well-known study on the subject, performed by Lippert and Lopes [12], had the goal of evaluating

if AOP could be used to separate the code responsible for detecting and handling exceptions from the normal application code in a large object-oriented (OO) framework. The authors found that the use of AOP brought several benefits, such as less interference in the program texts and a drastic reduction in the number of lines of code (LOC). However, this first study has not investigated the ”aspectization” of application-specific error handling, which is often the case in large-scale software systems. Moreover, in spite of the assumption made by many authors that using AOP for separating exception handling code from the normal application code is beneficial, the trade-offs involved in using AOP with this goal are not yet well-understood. To the best of our knowledge, no work in the literature has attempted to assess whether AOP really promotes an enhancement in well-understood quality attributes such as separation of concerns, coupling, cohesion, and conciseness, when used for modularizing non-trivial exception handling code.

This paper presents a study performed to assess the adequacy of AspectJ [11], a general purpose aspect-oriented extension to Java, for modularizing exception handling code. The study consisted of refactoring part of a real OO system so that the code responsible for handling exceptions was moved to aspects. This study differed from the Lippert & Lopes study in the following points:

- The target of the study is part of a complete, deployable system, not a reusable infrastructure, like a framework. Hence, the exception handling code implements non-uniform, complex strategies, making it harder to move handlers to aspects.
- We employ the metrics suite proposed by Sant'Anna et al [15] to assess attributes such as coupling, conciseness, cohesion, and separation of concerns in both the original and the refactored system.
- We assess the overall quality of both the error handling aspects and the application classes affected by them.
- We have not attempted to move error detection code to aspects.

We have found that, in general, AOP improved separation of concerns between exception handling code and normal application code. Moreover, we noticed that aspects promote handler reuse, but reusing handlers requires careful design planning. Otherwise, the behavior of the system may be unintentionally altered when the handlers are extracted to aspects. Furthermore, contradicting the general intuition, we observed that, for systems with application-specific exception handling strategies, an aspect-oriented (AO) solution does not result in a reduced number of LOC. For the system we have refactored, the AO version had almost the same number of LOC as the OO version. Another consequence of using aspects was that, in many cases, it was necessary to refactor the application code to expose join points that AspectJ can capture. This produced code that did not appropriately express the intent of the programmer and had a negative impact in the overall cohesion of the system.

This paper is organized as follows. Section 2 describes the setting of our study, while providing very brief descriptions to the AspectJ language and to the Lippert & Lopes study. The results of the study are presented and analyzed in Sections 3 and Section 4, respectively. Section 5 discusses some limitations of our study and the last section points directions for future work.

2 Study Setting

This section describes the configuration of our study. Section 2.1 briefly describes the AO programming language we have used, AspectJ. Section 2.2 provides an overview of the Lippert & Lopes study. Section 2.3 describes the Telestrada system, the target of our study. Section 2.4 presents an example of how exception handling code was moved to aspects in our study. Section 2.5 presents the metrics we have used to evaluate the OO and AO versions of Telestrada.

2.1 AspectJ Overview

AspectJ [11] is a general purpose aspect-oriented extension to Java. It extends Java with constructs for picking specific points in the program flow, called join points, and executing pieces of code, called advice, when these points are reached. Join points are points of interest in the program execution through which crosscutting concerns are composed with other application concerns.

AspectJ adds a few new constructs to Java, in order to support the selection of join points and the execution of advice in these points. A *pointcut* picks out certain join points and contextual information at those join points. Join points selectable by pointcuts vary in nature and granularity. Examples include method call and class instantiation. Advice may be executed *before*, *after*, or *around* the selected join points. In the latter case, execution of the advice may potentially alter the flow of control of the application, and replace the code that would be otherwise executed in the selected join point. AspectJ also allows programmers to modify the static structure of a program by means of static crosscutting. With static crosscutting, one can introduce new members in a class or interface, or make a checked exception unchecked.

Aspects are units of modularity for crosscutting concerns. They are similar to classes, but may also include pointcuts, advice, and static crosscutting. Aspects are combined with Java code by means of a process called weaving. Therefore, the tool responsible for performing weaving is called *weaver*.

2.2 Lippert and Lopes' Study

The study of Lippert and Lopes used an old version of AspectJ to refactor exception handling code in a large OO framework, called JWAM, to aspects. The goal of this study was to assess the usefulness of aspects for separating exception handling code from the normal application code. The authors presented their findings in terms of a qualitative evaluation. Quantitative evaluation consisted solely of counting LOC. They found that the use of aspects for modularizing exception detection and handling in the aforementioned framework brought several benefits, for example, better reuse, less interference in the program texts, and a decrease in the number of lines of code. The Lippert & Lopes study was a important initial evaluation of the applicability of AspectJ and aspects in general for solving a real software development problem. However, it has some shortcomings that hinder its results to be extrapolated to the development of real-life software systems.

First, the target of the study was a system where exception handling is generic (not application-specific). However, it is well-known that exception handling is an inherently application-specific error recovery technique [1]. In other words, the real exception handling would be implemented by systems built using JWAM as an infrastructure and not by the framework itself. The authors report that most of the handlers in JWAM implemented policies such as "log and ignore the exception". This helps explaining the vast economy in LOC the authors achieved by using AOP.

Second, the qualitative assessment was performed in terms of quality attributes that are not well-understood, such as (un)pluggability and support for incremental development. The authors did not evaluate some attributes that are more fundamental and well-understood in the Software Engineering literature, such as coupling and cohesion.

Third, quantitative evaluation was performed only in terms of number of LOC. Although the number of LOC may be relevant if analyzed together with other metrics, its use in isolation is usually the target of severe criticisms [17]. In the context of the Lippert & Lopes study, the use of LOC as the sole metric provided a narrow view of the effects of the aspectization of exception handling on the program quality. It portrayed the AO solution as very superior to the OO solution even though, as described previously, this owed more to the nature of the target of the study than to the quality of the AO solution.

2.3 Telestrada: Our Case Study

Telestrada [4] is a large traveler information system being developed for a Brazilian national highway administrator. It comprises five subsystems: Central Database Subsystem, GIS (Geographic Information System) Subsystem, Call-Center Operations Subsystem, Roadside Operations Subsystem, and Complaint Management Subsystem.

For our study, we have selected some self-contained packages of the Complaint Management Subsystem (CMS). The implementation of the CMS comprises more than 12000 LOC and more than 300 classes. The packages we selected for the study comprise approximately 1600 LOC (excluding comments and blank lines) and more than 120 classes and interfaces.

The classes and interfaces of the selected portion of the CMS include more than 45 `try-catch` blocks of varied complexity. They implement diverse exception handling strategies that range from trivial to sophisticated, for example: (i) do nothing (empty catch block); (ii) log and close database connection; (iii) log the exception, perform a rollback, close the database connection, and raise a different exception; (iv) use Java's reflection API to create a new `Method` object and use it for logging.

2.4 Aspectizing Exception Handling

Our study focuses specifically on the handling of exceptions. We moved all the `try-catch`, `try-catch-finally`, and `try-finally` blocks in the selected portions of Telestrada to aspects. Method signatures (`throws` clauses) and the raising of exceptions (`throw` statements) were not affected because these elements are more related to exception detection than to exception handling.

Handlers moved to aspects were implemented by means of after and around advice, depending on whether or not the handler ended its execution by raising an exception, respectively. Whenever possible, we used after advice, since they are simpler. After advice are not appropriate, though, for implementing handlers that do not raise (or re-raise) an exception because these advice cannot alter the flow of control of a program. In cases where this was necessary, around advice were used.

New advice were created on a per-`try`-block basis, excluding cases where handlers could be reused. For each class in the original system, we defined an aspect to handle exceptions raised by the members of the class. In many cases, moving handlers to aspects required some refactoring of the original code. The following code snippet presents an example:

```
public class GenericOperations {
  public static boolean closeResultSet(ResultSet aResultSet) {
    boolean r = true;
    try { ...  // body of the "try" block.
    } catch (SQLException e) { System.out.println(e.toString());
      r = false;
    }
    return r;
  }   ... // implementation of the class
}
```

The procedure we used to move handlers to aspects is very similar to the *Extract Method* refactoring [5] and the same restrictions apply. After extracting all the handlers to aspects, we searched for reuse oportunities and eliminated identical handlers. For the example above, it was necessary to remove references to the local variable `r` from the `try-catch` block before moving it to an advice. Moving the `try-catch` block to an aspect named `GOHandler` produces the following code:

```
public class GenericOperations {
  public static boolean closeResultSet(ResultSet aResultSet) {
      ... // body of the original "try" block.
      return true;
  } ... // implementation of the class
}
public aspect GOHandler { // another source file
  pointcut crsHandler() :
      execution(public static boolean closeResultSet(..));
  boolean around(ResultSet rs) : crsHandler() && args(rs){
    try { return proceed(rs);
    } catch (SQLException e) { System.out.println(e.toString());
      return false;
    }
  }
  declare soft : SQLException : crsHandler();
}
```

Method `closeResultSet()` now consists of the body of the original `try` block, plus a return statement. In the `GOHandler` aspect, we defined a pointcut named `crsHandler` to select the execution of `closeResultSet()`. The around advice to which we extracted the `try-catch` block is executed at this join point. This advice calls `closeResultSet()` by means of the `proceed()` statement of AspectJ and, if no exceptions are raised, returns the result of the method execution. If an `SQLException` is raised, the exception handler is executed. In this example, `SQLException` was *softened*, meaning that Java's static checks are suppressed at

`crsHandler`. This is necessary because the body of `closeResultSet()` can still raise `SQLException` but, from the viewpoint of the Java compiler, the exception is not being handled since the handler is in an aspect.

2.5 Metrics Suite

In our study, we have selected a suite of metrics for separation of concerns, coupling, cohesion, and size [15] to evaluate both OO and AO implementations. These metrics have already been used in three different experimental studies [8, 7, 10] and have been effective to assess several internal quality attributes of Java and AspectJ programs. Some of them have been automated in the context of a measurement tool [16]. This metrics suite was defined based on the reuse and refinement of some classical OO metrics [2, 3]. The original definitions of the OO metrics [2] were extended to be applied in a paradigm-independent way, supporting the generation of comparable results.

The metrics suite also encompasses new metrics for measuring separation of concerns. They were used in our study to measure the degree to which the exception handling concern in Telestrada maps to the design components (classes and aspects), operations (methods and advice), and lines of code.

The employed metrics suite includes metrics related to four quality attributes: separation of concerns, coupling, cohesion, and size. Separation of concerns refers to the ability to identify, encapsulate, and manipulate those parts of software that are relevant to a particular concern. Coupling is an indication of the strength of interconnections between the components in a system. Highly coupled systems have strong interconnections, with program units dependent on each other. The cohesion of a component is a measure of the closeness of the relationship between its internal components. The software size measures the length of a software systems design and code.

Table 1 presents a brief definition of each metric, and associates them with the attributes measured by each one. In general, the higher the value of a measure, the worse the performance of the assessed system with respect to that metric. Detailed descriptions of the metrics appear elsewhere [15].

The use of the Concern Diffusion over LOC metric requires a shadowing process that partitions the code into shadowed areas and non-shadowed areas. The shadowed areas are lines of code that implement a given concern. Transition points are the points in the code where there is a transition from a non-shadowed area to a shadowed area and vice-versa. The intuition behind it is that they are points in the program text where there is a concern switch. For each concern, the program text is analyzed line by line in order to count transition points. The higher the CDLOC, the more intermingled is the concern code within the implementation of the components; the lower the CDLOC, the more localized is the concern code.

3 Study Results

This section presents the results of the measurement process. The data have been collected based on the set of defined metrics (Section 2.5). The presentation is broken in

Attributes	Metrics	Definitions
Separation of Concerns	Concern Diffusion over Components	Counts the number of classes and aspects whose main purpose is to contribute to the implementation of a concern plus the number of other classes and aspects that access them.
	Concern Diffusion over Operations	Counts the number of methods and advice whose main purpose is to contribute to the implementation of a concern plus the number of other methods and advice that access them.
	Concern Diffusion over LOC	Counts the number of transition points for each concern through the lines of code. Transition points are points in the code where there is a "concern switch".
Coupling	Coupling Between Components	Counts the number of components declaring methods or fields that may be called or accessed by other components.
	Depth Inheritance Tree	Counts how far down in the inheritance hierarchy a class or aspect is declared.
Cohesion	Lack of Cohesion in Operations	Measures the lack of cohesion of a class or an aspect in terms of the amount of method and advice pairs that do not access the same field.
Size	Lines of Code (LOC)	Counts the lines of code.
	Number of Attributes	Counts the number of fields of each class or aspect.
	Number of Operations	Counts the number of methods and advice of each class or aspect.
	Vocabulary Size	Counts the number of components (classes, interfaces, and aspects) of the system.

Table 1. The Metrics Suite

three parts. Section 3.1 presents the results for the separation of concerns metrics. Section 3.2 presents the results for the coupling and cohesion metrics. Section 3.3 presents the results for the size metrics.

We present the results by means of tables that put side-by-side the values of the metrics for the OO and AO version of Telestrada. Where relevant, results are broken in two parts, in order to make it clear the contribution of classes and aspects to the value of each metric. Hereafter, we use the term "class" to refer to both classes and interfaces.

3.1 Separation of Concerns Measures

Table 2 shows the obtained results for the three separation of concerns metrics. The AO version of Telestrada performed better for two of the three separation of concerns metrics, Concern Diffusion over Operations and Concern Diffusion over LOC. The two versions had the same value for Concern Diffusion over Components.

In the AO version of Telestrada, code related to the implementation of the exception handling concern was moved to aspects. Therefore, for all the metrics, the number of classes implementing exception handling was zero. The identical values for Concern Diffusion over Components in the OO and AO versions of Telestrada are due to the design choice of creating one "handler aspect" for each public class that implemented exception handling in the OO version. Other possible design choices would be to put the exception handling code in a single aspect or, for each exception, create

Metrics	# components	OO version	AO version
Concern Diffusion over Components	classes	8	0
	aspects	-	8
	total	**8**	**8**
Concern Diffusion over Operations	classes	25	0
	aspects	-	21
	total	**25**	**21**
Concern Diffusion over LOC	classes	131	0
	aspects	-	24
	total	**131**	**24**

Table 2. Separation of Concerns Metrics

an aspect that encapsulates the possible handling strategies for a given exception. The three approaches have pros and cons that revolve around the code size vs. modularity trade-off. This trade-off is also faced by developers applying design patterns [6] to unstructured OO systems. Our design choice was a middle-ground between a single, possibly bloated, aspect and more than twenty, possibly too fine-grained, aspects.

The AO version exhibited a better Concern Diffusion over Operations (16% lower than the OO version). For most components, the AO solution was either equivalent or superior to the OO one. Two exceptions were the AO versions of `db.ConnectionPool` and `system.modifyComplaint.Façade` components of Telestrada. The AO versions of these components had higher values in Concern Diffusion over Operations because the OO version had operations with more than one `try-catch` block. When these handlers were moved to aspects, each one had to be put in a separate advice. Moreover, handler reuse was low for these components, since they implement very context-specific exception handlers.

When moving handlers to aspects, we reused handler advice as much as possible. For example, even though `GenericOperations` had some methods that had more than one `try-catch` block, the AO version exhibited a lower value in Concern Diffusion over Operations, since some of the handler advice could be reused. However, we avoided situations where handler reuse could cause exceptions to be swallowed, since this could change the behavior of the system. For instance, we did not merge the following two advice, from the `ConnectionPoolHandler`[1] aspect, in a single one:

```
void around() : setPropertiesHandler() {
  try { proceed();
  } catch (MissingResourceException mre) { // do nothing
  } catch (NumberFormatException nfe) { // do nothing }
}
void around() : logHandler() {
  try { proceed();
  } catch (Exception e) {} // ignore exceptions when logging
}
```

Associating the `setPropertiesHandler` pointcut to the second advice would cause unchecked exceptions to be caught and ignored. However, the advice to which this pointcut is associated does not interfere with the propagation of these exceptions.

Concern Diffusion over LOC was the metric where aspects performed best. The AO version of Telestrada had less than 20% of the number of concern switches of the OO

[1] Handler aspects have the same name as their corresponding classes, plus the sufix “Handler”.

version. This finding confirms the results in the Lippert & Lopes study. The authors claim that the use of aspects decreases interference between concerns in the program texts.

3.2 Coupling and Cohesion Measures

Table 3 shows the obtained results for the two coupling metrics, Coupling between Components and Depth of Inheritance Tree, and the cohesion metric, Lack of Cohesion in Operations.

Metrics	**# components**	**OO version**	**AO version**
Coupling between Components	classes	73	58
	aspects	-	16
	total	**73**	**74**
Depth of Inheritance Tree	classes	73	73
	aspects	-	2
	total	**73**	**75**
Lack of Cohesion in Operations	classes	171	298
	aspects	-	-
	total	**171**	**298**

Table 3. Coupling and Cohesion Metrics

The OO and AO versions of Telestrada exhibited very similar measures for the coupling metrics. The Depth of the Inheritance Tree increased by less than 3% in the AO version. This was expected, since the use of aspects alone does not interfere with this metric. The increase of 2 in the value of the measure was due to the creation of a new aspect from which two handler aspects, `system.modifyComplaint.FaçadeHandler` and `system.registerComplaint.FaçadeHandler`, inherited. The super-aspect was created in order to avoid duplicated code.

Coupling between Components in the two versions was almost identical. New couplings were introduced only when aspects had to capture contextual information from classes. In these cases, at most one new coupling is created per aspect, due to a reference from the aspect to its corresponding class.

Among all the metrics, Lack of Cohesion in Operations was the one for which the AO version of Telestrada presented the worst results. Lack of cohesion in the operations of the AO version was more than 75% higher than in the OO version. This is due to the large number of operations that were created to expose join points that AspectJ can capture. These new operations are not part of the implementation of the exception handling concern (and therefore do not affect Concern Diffusion over Operations), but are a direct consequence of using aspects to modularize this concern. Refactoring to expose join points is a common activity in aspect-oriented software development [13], since current aspect languages do not provide means to precisely capture every join point of interest.

It is interesting to note that the goal of the Lack of Cohesion in Operations metric is to capture a partial view of cohesion: it considers only the explicit relationships between

the attributes and operations. It does not consider direct inter-operation relationships and the semantic closeness between elements of a component. Moreover, even though cohesion was worse in the AO version, the aspects had very good measures for Lack of Cohesion in Operations. This happened because none of the handler advice accesses fields of the classes they refer to and the aspects do not define new fields. Hence, there are no values of Lack of Cohesion in Operations for the aspects in Table 3.

3.3 Size Measures

Contradicting the general intuition that aspects make programs smaller [8, 11, 12], the OO and AO versions of Telestrada had very similar results in three of the four size metrics: LOC, Number of Attribues, and Vocabulary Size. Moreover, the number of operations of the AO version was 21% higher than the OO version. Table 4 summarizes the results for the size metrics.

Metrics	**# components**	**OO version**	**AO version**
Lines of Code (LOC)	classes	1594	1290
	aspects	-	285
	total	**1594**	**1575**
Number of Attributes	classes	50	50
	aspects	-	0
	total	**50**	**50**
Number of Operations	classes	166	180
	aspects	-	21
	total	**166**	**201**
Vocabulary Size	classes	113	113
	aspects	-	8
	total	**113**	**121**

Table 4. Coupling and Cohesion Metrics

The similar values for LOC were expected. As mentioned in Section 3.1, reusing handler aspects in Telestrada was much harder then we had originally predicted. Hence, although some reuse could be achieved, this was not anywhere near the results obtained by Lippert & Lopes in their study. Moreover, most handlers comprise just a few LOC and the use of AspectJ incurred in a slight implementation overhead because it was necessary to specify join points of interest and soften exceptions in order to associate handlers to pieces of code. In the end, the economy in LOC achieved due to handler reuse was compensated by the overhead of using AspectJ.

The 7% increase in the vocabulary size of the AO version was entirely due to the aspects. No new classes were introduced or removed. Similarly to Concern Diffusion over Components (Section 3.1), Vocabulary Size depends heavily on how the implementation of the exception handling concern is partitioned among the aspects.

The number of operations in the AO version of Telestrada was 17% bigger than in the OO version. The main reason for this increase was the creation of advice implementing handlers. Since there is a one-to-one correspondence between `try` blocks and advice, except for cases where handlers are reused, and handlers do not count as

methods in the OO version, this increase was expected. Another reason for the increase in the Number of Operations was the refactoring of methods to expose join points that AspectJ can capture.

4 Analysis of the Results

In general, we found that reusing handlers is much more difficult than is usually advertised [12]. Handler reuse depends directly on: (i) the type of exception being handled; (ii) what the handler does and whether it ends its execution by returning or raising an exception; (iii) the amount of contextual information required; and (iv) what the method that raises the exception returns and what exceptions appear in its `throws` clause. Factor (ii) above, whether a handler ends its execution by returning or raising an exception, is important because it restricts the types of advice that can be used.

When exception handlers are non-trivial, it may be difficult to fully understand the implications of moving a handler to an aspect. Hence, reusing handlers requires careful design, in order to avoid changing the exceptional behavior semantics of the system. The same issue applies for exception softening. Softening an exception that is a supertype of another exception raised within the same context causes the subtype to be softened as well, possibly with unexpected effects. We believe that developers should never soften exceptions that are supertypes of many other exceptions, such as `Exception` and `Throwable` in Java.

In spite of the better Concern Diffusion over LOC of the AO version of Telestrada, we expected the difference to be even bigger. This did not happen because, as discussed in Section 2.4, we used around advice whenever a handler did not end its execution by raising an exception. An around advice executes the code of its selected join points explicitly, by means of the `proceed()` statement. Since, in our case study, the code of the selected join points corresponds to the system's normal activity, occurrences of `proceed()` can be seen as concern switches.

Although the Coupling between Components in both versions of Telestrada was almost identical, this does not mean that components are as coupled to each other in the AO version as in the OO version. The measures of Coupling between Components for the classes in the AO version were lower than in the OO version. Moreover, the sums of the measures of Coupling between Components for the classes and their corresponding aspects in the AO version were similar to the measures for the respective classes in the OO version. Therefore, we can say that the AO version has more components but they are, in general, less strongly coupled to one another.

As seen in Section 3.3, handler advice accounted for a 10% increase in the number of operations. As with all size metrics, this value cannot be evaluated in isolation. Although a developer getting acquainted to the AO version will have to understand more operations, these operations are simpler and do not mix the system's normal activity with the code that handles exceptions. Therefore, the increase in the Number of Operations caused by the handler advice can be seen as a positive factor.

The number of operations refactored to expose join points that AspectJ could capture corresponded to 7% of the total Number of Operations measure. Unlike the increase caused by handler advice, the increase caused by refactored operations is definitely neg-

ative. These new operations are not part of the original design of the system and possibly do not clearly state the intent of the developer. In some cases, the refactored operations comprised just a couple of lines that did not make much sense when separated from their original contexts. This problem may suggest that there is still room for improving AspectJ so that more join points of interest can be captured.

5 Limitations of this Study

Our study focuses on a single aspect-oriented language, namely, AspectJ. Although many ideas presented here also apply to other AO languages, some surely do not. For example, it is not necessary (or possible) to soften exceptions in Eos [14], an aspect-oriented extension to C#, because C# does not have checked exceptions.

Not all possible strategies for implementing exceptional behavior of systems are covered. In Telestrada, handlers are implemented exclusively by means of `catch` blocks. However, more complex applications may include methods and fields which are specific to the implementation of the exceptional behavior. Moving these additional elements to aspects would probably affect the quality attributes of the refactored system.

We do not attempt to evaluate the scalability of aspects for modularizing exception handling. Although the target of our study implements non-trivial exception handling policies, it is still just part of a system and comprises less than 2000 LOC. Moreover, we only modularize exception handling using aspects. We do not evaluate interactions between exception handling aspects and aspects implementing other concerns.

6 Future Work

Our most immediate future work is to derive a predictive model for using aspects to implement exception handling, based on the lessons learned from this study. With this model, developers will be able to recognize the situations in which it is advantageous to use aspects to modularize exception handling code. Moreover, we intend to document as patterns some strategies for structuring exception handling aspects.

As mentioned in Section 5, we have not evaluated the scalability of AspectJ for implementing exception handling. In the near future, we intend to analyze two scenarios: (i) whether aspects scale up well when the number of handlers grows; and (ii) whether it is difficult to integrate exception handling aspects with aspects implementing other concerns, such as distribution and persistence.

Acknowledgements

We would like to thank Alexandra Barros for the many insightful comments and suggestions.

Fernando is supported by FAPESP/Brazil under grant 02/13996-2. Cecília is partially supported by CNPq/Brazil under grant 351592/97-0. Fernando and Cecília are partially supported by FINEP/Brazil under grant 1843/04 of CompGov, which is a project for Shared Library of Components for e-Government. Alessandro is supported

by European Commission grant IST-2-004349: European Network of Excellence on Aspect-Oriented Software Development (AOSD-Europe), 2004-2008.

References

1. T. Anderson and P. A. Lee. *Fault Tolerance: Principles and Practice*. Springer-Verlag, 2nd edition, 1990.
2. S. Chidamber and C. Kemerer. A metrics suite for oo design. *IEEE Trans. on Soft. Eng.*, 20(6):476–493, June 1994.
3. N. Fenton and S. L. Pfleeger. *Software Metrics: A Rigorous Practical Approach*. PWS, 1997.
4. F. Castor Filho et al. A systematic approach for structuring exception handling in robust component-based software. *Journal of the Brazilian Computer Society*, 2005. To appear.
5. M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
6. E. Gamma et al. *Design Patterns: Elements of Reusable Software Systems*. Addison-Wesley, 1995.
7. A. Garcia et al. Separation of concerns in multi-agent systems: An empirical study. In C. Lucena et al., editors, *Software Engineering for Multi-Agent Systems II*, LNCS 2940. Springer-Verlag, February 2004.
8. A. Garcia et al. Modularizing design patterns with aspects: A quantitative study. In *Proceedings of the 4th AOSD*, pages 3–14, Chicago, IL, USA, March 2005.
9. G. Kiczales et al. Aspect-oriented programming. In *Proceedings of the 11th ECOOP'97*, pages 220–242. Springer Verlag LNCS 1241, 1997.
10. U. Kulesza et al. Aspectization of distribution and persistence: Quantifying the effects of aop. *Submitted to IEEE Software, Special Issue on AOP*, May 2005.
11. R. Laddad. *AspectJ in Action*. Manning, 2003.
12. M. Lippert and C. V. Lopes. A study on exception detection and handling using aspect-oriented programming. In *Proceedings of the 22nd ICSE'2000*, pages 418–427, Limerick, Ireland, June 2000.
13. G. Murphy et al. Does aspect-oriented programming work? *Communications of the ACM*, 44(10):75–77, October 2001.
14. Hridesh Rajan and Kevin Sullivan. Eos: Instance-leve aspects for integrated system design. In *Proceedings of ESEC/FSE'2003*, Helsinki, Finland, September 2003.
15. C. Sant'Anna et al. On the reuse and maintanance of aspect-oriented software: An assessment framework. In *Proceedings of the 17th SBES*, pages 19–34, October 2003.
16. Tigris. aopmetrics home page, 2005. Address: http://aopmetrics.tigris.org.
17. H. Zuse. History of software measurement, 2005. Address: http://irb.cs.tu-berlin.de/ zuse/metrics/History_00.html.

Position Paper: Handling "Out Of Memory" Errors ⋆

John Boyland⋆⋆

University of Wisconsin-Milwaukee, USA
boyland@cs.uwm.edu

Abstract. An "out of memory" error can be catastrophic for a program, especially one written in a language such as Java that uses memory allocation frequently. Handling such an error can easily lead to its re-occurrence. A handler will often need memory while it is freeing resources (by persisting data to secondary storage, or clearing caches). A simple technique involves pre-allocating a large chunk of memory that is then freed at the start of the handler. I report some experience using this technique and discuss some of the problems that arise when reasoning about the behavior of memory error handlers.

1 Introduction

A running program may encounter several kinds of severe conditions that result, not from errors in program logic, per se, but from over use of resources. Typically there is no hard bound on the time that can be used, but increasing the bounds for memory use (either stack or heap) can have disastrous and long-term performance problems (swapping). Thus even with a 64-bit address space, a Java program may be limited to (say) 2 GB of memory. If more memory is needed than is available, an OutOfMemoryError or a StackOverflowError is thrown. The latter case typically results from infinite or inappropriate recursion, but the former can occur easily.

Techniques are available for statically determining the required heap size of a program (see this year's FTJP for at least one paper), but these are typically not applicable for complex programs, and indeed one may wish to run a program that may exceed memory limits for some inputs. Furthermore, it is not always best practice to run a program to use the least space possible. Caching a computation exchanges space for time performance. Later if it becomes known that space is tight, the cache can be jettisoned with little semantic cost.

One argument against handling OutOfMemoryError is that one should instead simply check whether there is sufficient memory before continuing with a

⋆ This work is partially supported by the National Aeronautics and Space Administration (NASA) under the HRT program

⋆⋆ The author wishes to acknowledge support through the NASA High Dependability Computing Program under cooperative agreement NCC-2-1298.

memory expensive operation. This alternative suffers from all the same portability problems (one cannot be sure that one actually has all the memory one needs, and one cannot be sure that the recovery technique won't actually have its own memory problems) and additionally slows down the program by constantly checking for memory sufficiency. Exceptions are designed precisely to handle this sort of infrequently occurring behavior; it's easier to thrown an exception at the point an error is found rather than always to try to detect problems that might occur ahead of time.

I have been investigating importing large Java code bases (up to a million LOC) into our versioned persistent representation. The persistence mechanism can only be invoked once per "era" and for various reasons, one wishes to create as few eras as possible. Thus the desired process is to load as much code as will fit in memory, create an "era," persist the lot and then continue. Unfortunately it can be difficult to predict exactly how memory a given Java file will take in the code base before it is loaded, and thus the safest way to avoid a memory error is to create one "era" for every file. A less conservative solution is desired: creating "eras" on demand.

Creating an "era" and saving the code base takes memory itself, and thus I wrote a simple hook to allocate a large (16 MB) array and when the memory error is thrown during an import, the outer loop catches the error, frees the array (through the simple expedient of nulling out the global variable that points to it) and then creates the "era" and so on. After the cleanup is done, an new array is allocated and the process of importing code continues. I have used this code to import the source that came with JDK 1.4.2 (45 Mbytes) into our (verbose) intermediate form. With a maximum heap size set at 300 Mb (on a 512 Mb system), the program used only 12 eras (handling 11 OutOfMemoryError exceptions) to load 4500 files. The final result was stored in a custom serialization format that, in compressed form, is 150 Mb.

Reasoning about such code, and in particular finding errors (debugging) can be difficult. This paper discusses some of the problems that arise. Some language design issues are also touched upon.

2 Reasoning about Memory Error Handlers

Excepting program analysis for finding maximum heap requirements, "out of memory" errors are typically ignored when reasoning about a program; this exception could arise at almost any point. It is considered an unpredictable fatal event. Here we consider alternate approaches.

2.1 After Running out of Memory

If one doesn't just treat a memory error as a catastrophic event that kills the entire program, one needs to examine what happens after the exception is thrown. At the point that the error is determined, the program is left in a precarious state: unable to allocate any memory at all. It may seem no different than in any

other program state (where one could always been just one step away from an out-of-memory disaster), but a particular problem are "finally" clauses. These are used to simulate dynamic binding of global context variables (such as the current "version" of the state in our case). Such a "finally" clause must be written carefully to require no heap allocation.

Avoiding memory allocation in "finally" clauses is made harder since the Java specification does not indicate exactly which language constructs cause memory allocation (at least as far as I know). Some limited experience with JDK 1.4.2 seems to indicate that even a non-final instance method call can cause an out-of-memory error. It would be easier to write more portable handling of these errors if the language specification was able to guarantee certain operations as safe in low-memory conditions.

If a "finally" clause cannot avoid allocation of memory, then it is better to allocate the memory before performing the task that needs to be unwound. Alternatively, a buffer can be allocated, and then the pointer to it is nulled at the start of the finally clause. The latter case is less desirable, because then it may be necessary to "fool" the compiler into not freeing the memory earlier. In any case, this should be a general rule for "finally" clauses, since such cleanup actions should be runnable in all situations, including low-memory.

2.2 The "Hedge" Technique

Once the stack is safely unwound to the point where the error is caught, the memory error must be handled. Since so many operations in Java (e.g. opening a file output stream) involve creating new objects on the heap, it is essential that an out-of-memory error handler keep a safety "hedge" of memory. This is freed before we do whatever we can to free memory. In a garbage-collected system, "freeing" memory means to make it unreachable so that the garbage collector will reclaim it. The *recovery* process is the attempt to free normal program data, before returning to normal program execution. Since the last operation was aborted due to memory shortage, typically after recovery is complete, one attempts to redo the task that was interrupted by the out-of-memory error.

Recovery can fail in several ways:

1. Memory is exhausted while running the recovery: the hedge was not big enough.
2. After the recovery is done, the hedge cannot be reallocated: the recovery actually consumed memory rather than freeing it.
3. After recovery is done, not a sufficient amount of memory has been freed: if no memory has been freed then the retry of last task will simply fail again.

The third condition can be imperfectly tested by requiring that a minimum number of bytes are freed by the recovery. But this cannot ensure that the task will successfully complete this time; in fact it may simply not be possible to complete. Thus an interactive program should not blindly repeat any memory-error-interrupted task after recovery. My Java code importer suffers from this

problem; if the imported file simply cannot fit in memory, even by itself, the system tries over and over. This is a fine for my purposes now, but will need to be changed at some point.

Debugging in such situations is tricky. For obvious reasons, the OutOfMemoryError object does not keep a copy of the run-time stack at the time of the fault, and in handlers one must avoid the string concatenation operator. Thus it is difficult to determine precisely where the error was raised.

2.3 Systematizing the "Hedge"

It would seem attractive to have a system function keep track of the hedge so it could be freed right at the point the memory fault happens. A program could tell the system to reserve a certain amount of heap for low-memory recovery. This memory would be made available at the time the next OutOfMemoryError is thrown.

Unfortunately it is confusing to build this ability directly into the run-time system. Presumably after the first OutOfMemoryError is thrown, the reserve memory would be brought out. The problem is that the code is not necessarily immediately ready to start recovery, and in fact may catch the error, not to recover, but rather to change the behavior of later code. The hidden reserve simply confuses clients. There is also the issue of which module of the program "owns" the reserve memory.

Rather, the recovery process must be explicit. I have implemented an abstract class with instances that reserve memory. Such an instance can be asked to handle an OutOfMemoryError. It handles the bookkeeping (including releasing the reserve) and then calling an abstract method to perform the recovery.

2.4 Multi-Threading Issues

When multiple threads are active (as in any interactive Java program), the low-memory condition affects all threads, not simply the one that is using the most and is (presumably) capable of recovering by freeing currently held memory. In particular, the thread that first receives the exception may have no useful way to recover. If one is writing a multi-threaded program, then some co-operation is needed, so that the offending thread can be notified. In the meanwhile, it will be necessary to make some hedge memory be available for the other threads.

Such a situation may indicate a need for different memory resource bounds to be assigned to different threads. Real-Time Java has a way to configure per-thread heap restrictions, but I am not aware that standard Java has any such restriction; it seems that such an extension would be a good idea.

2.5 Optimization

A compiler can determine when memory allocation is unnecessary or may move allocation to a later point. Unfortunately both actions can defeat the attempt to

reserve a memory hedge for recovery purposes. A language specification should provide for a way to indicate that a particular memory allocation must not be moved or removed.

2.6 Summary

An out of memory error need not be fatal, but recovery is tricky.

3 Related Work

A recent paper by Biswas and others [1] discusses the importance of checking for memory overflow in embedded systems where one cannot rely on the hardware to avoid (say) the stack and heap overlapping. They use program analysis to determine a small set of points at which the stack can be tested for overflow at runtime. At such points, if the stack is "about" to overflow, a special (unspecified) handler can be called, so as to avoid a serious error when the stack and heap overlap. The authors also provide techniques for using existing memory more efficiently when a memory overflow happens, by finding dead global variables or by compressing heap or global variable data. The intent is that this additional space wrung out of the existing situation can be used by the handler to cleanly shut down the embedded system. There is no discussion of using exceptions, or of "finally" blocks in Java.

For the most part, there is little published about handling OutOfMemoryError exceptions. The few discussions on this topic I found (on the web) have one of two viewpoints:

- Don't catch it; it's just a fatal random event you can't do anything about; or
- Use it to check how much memory is *really* free by repeatedly allocating successively smaller arrays (starting with some unreasonably large size) until the error *doesn't* happen.

The last trick is much safer than it looks because, of course, the new array whose allocation causes the problem is not in fact allocated, and thus there remains a "lot" of memory. On the other hand, since Java provides a method to check the remaining memory, it seems like an unnecessary hack.

Contra this common Java advice, a user-friendly program such as Eclipse must make some attempt to handle OutOfMemoryError exceptions. There does not seem to be a public specification for when and where to handle such errors, although this area is one of the areas that Eclipse is tested for. One of my hopes is that others in the workshop can propose good principles to follow for this case. This position paper merely reports my own uninformed thoughts and experience.

4 Conclusion

Low memory conditions must be handled by an interactive program and yet they are particularly difficult to handle. These conditions need not be fatal if

the program is able to substitute less memory intensive algorithms. I am looking forward to discussions at the workshop for how to handle these situations. Some preliminary observations can be made in the mean time:

- A "finally" handler should allocate no memory;
- A per-thread memory limit would help protect threads from each other;
- Compiler writers need to know whether they can delay or elimination allocations;
- There is a need for a well-distributed "best practice" for handling low memory conditions.

References

1. Biswas, S., Simpson, M., Barua, R.: Memory overflow protection for embedded systems using run-time checks, reuse and compression. In: Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems table of contents, New York, NY, USA, ACM Press (2004) 280–291