# Proceedings of International Workshop on Exception Handling in Object Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms Workshop

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

# Exception Handling in Object Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms

## Workshop

## July 21, 2003

## ECOOP 2003

# Table of Contents

# Introduction

There are two trends in the development of modern object oriented systems: they are getting more complex and they have to cope with an increasing number of exceptional situations. The most general way of dealing with these problems is by employing exception handling techniques. Many object oriented mechanisms for handling exceptions have been proposed but there still are serious problems in applying them in practice. These are caused by

- complexity of exception code design and analysis
- not addressing exception handling at the appropriate phases of system development
- lack of methodologies supporting the proper use of exception handling
- not developing specific mechanisms suitable for particular application domains and design paradigms.

Following the success of ECOOP 2000 workshop, this workshop aims at achieving better understanding of how exceptions should be handled in object oriented systems, including all aspects of software design and use: novel linguistic mechanisms, design and programming practices, advanced formal methods, etc.

The workshop will provide a forum for discussing the unique requirements for exception handling in the existing and emerging applications, including pervasive computing, ambient intelligence, the Internet, e-science, self-repairing systems, collaboration environments. We invited submissions on research in all areas of exception handling related to object oriented systems, in particular: formalisation, distributed and concurrent systems, practical experience, mobile object systems, new paradigms (e.g. object oriented workflows, transactions, multithreaded programs), design patterns and frameworks, practical languages (Java, Ada 95, Smalltalk, Beta), open software architectures, aspect oriented programming, fault tolerance, component-based technologies.

We encourage participants to report their experiences of both benefits and obstacles in using exception handling, reporting, practical results in using advanced exception handling models and the best practice in applying exception handling for developing modern applications in the existing practical settings.

Our intention is to discuss the problem of perceived complexity in using and understanding exception handling: why do programmers and practitioners often believe that it complicates system design and analysis? What should be done to improve the situation? Why is exception handling the last mechanism to learn and to use? What is wrong with the current practice and teaching?

**Alexander Romanovsky**
**Christophe Dony**
**Jørgen Lindskov Knudsen**
**Anand Tripathi**

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

**Jørgen Lindskov Knudsen**
Mjølner Informatics A/S
Helsingforsgade 27
DK-8200 Århus 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

# Workshop Program

9.00 - 9.10 Introduction. Alexander Romanovsky

**Section 1: Exception Handling Engineering (9:19-10:30)**

Invited talk. William Bail (Mitre) Getting Control of Exception

Johannes Siedersleben (SD&M Research, Germany). Errors and Exceptions – Rights and Responsibilities

**Section 2: OO Systems (11.00 - 12.30)**

Darrell Reimer, Harini Srinivasan (IBM Research, USA). Analyzing Exception Usage in Large Java Applications

Peter A. Buhr, Roy Krischer (U. Waterloo, Canada). Bound Exceptions in Object Programming

Joseph R. Kiniry (U. Nijmegen, Netherlands). Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application

**Section 3: Mobile and Distributed Systems (13.30-15.00)**

Denis Caromel, Alexandre Genoud (INRIA Sophia Antipolis, France). Non-Functional Exceptions for Distributed and Mobile Objects

Giovanna Di Marzo Serugendo (U. Geneva, Switzerland), Alexander Romanovsky (U. Newcastle, UK). Using Exception Handling for Fault-Tolerance in Mobile Coordination-Based Environments

Robert Miller, Anand Tripathi (U. Minnesota, USA). Primitives and Mechanisms in the Guardian Model for Exception Handling in Distributed Systems

**Section 4: Components (15.30-17.00)**

Ricardo de Mendonça da Silva, Paulo Asterio de C. Guerra,, Cecília M. F. Rubira (U. Campinas, Brazil). Component Integration using Composition Contracts with Exception Handling

Tom Anderson, Mei Feng, Steve Riddle, Alexander Romanovsky (U. Newcastle, UK). Error Recovery for a Boiler System with OTS PID Controller

Frederic Souchon (LGI2P Nimes and LIRMM Montpellier, France), Christelle Urtado, Sylvain Vauttier (LGI2P Nimes, France), Christophe Dony (LIRMM Montpellier, France). Exception Handling in Component-based Systems: a First Study

**Section 5: Discussion and Wrap-up (17.00-17.20)**

# Invited talk

**William Bail (Mitre):** ***Getting Control of Exception***

## Abstract

Being able to define and use exceptions has provided a significant advantage in being able to write more reliable software. While not explicitly helping us avoid errors, they enable us to detect their presence and control their effects. Yet they act in opposition to much of what we have learned is good software design - simple structures with well-defined control flows. In addition, they complicate the process of performing formal analyses on our systems. This talk explores this issue and projects some potential ideas to help reconcile these challenges, especially with the use of OO concepts.

# Errors and Exceptions – Rights and Responsibilities

Johannes Siedersleben
sd&m Research, Munich[1]

**Abstract**

There is no generally accepted agreement on how exceptions are to be used. Many projects suffer from a mess of exceptions thrown across the system with no defined responsibility for catching them. This paper presents a simple component-based strategy addressing the following points:

- How many and which exception classes are useful?
- When should an exception be thrown? In Java: should it be a checked or an unchecked exception?
- Who is responsible for catching exceptions?
- How far may exceptions be thrown?

Preconditions are considered in detail. The paper is based on the experience of many real large software projects.

## I. Introduction

Exceptions are considered to be an important added value of modern programming languages, but they have turned out to be hard to use. We list some of the typical problems encountered in many projects in the range from 1 to more than 100 man-years:

- There is a mess of exceptions flying around. It is neither clear when exceptions should be thrown nor how they are caught.
- The code gets messy because of nested *try-catch* blocks.
- Many (sometimes all) catch blocks are either empty, contain nonsense code (output to the console, useless mappings of one exception class into another) or – at best – some logging, but no true exception handling.
- A huge number of exception classes create undesired dependencies between the caller and the callee.
- Exceptions are misused to return ordinary values.

This paper presents a guideline for dealing with errors and exceptions in large systems. Code examples are in Java, but most results apply to other object oriented languages as well. The paper is based on material published as early as 1991 [Denert1991] and further developed in [Siedersleben2002].

The rest of the paper is structured as follows: Chapter II and III discuss exceptions and the problems they cause in programming languages; Chapter IV introduces the idea of component-based emergencies; Chapter V and VI present strategies for exception handling.

[1] software design & management,
Thomas-Dehler-Str. 27
D 81737 München

Pre- and postconditions are addressed in Chapter VII, and Chapter VIII summarizes ten rules to follow.

## II. Exceptions and Normal Business

Software may fail as any other result of human engineering. Thus, software is expected to cope with failure: damage should be minimized; recovery and restart should be part of the system design. [Parnas1976] calls *undesired event* whatever is not considered to be part of the normal business and suggests to handling these events in separated traps. This is just another application of *separation of concerns*, which is a truism in other branches of engineering. Let us look at a fighter: as long as things go well, the fighter is fully functional without the ejector seat; the whole system is unaware of this feature. The ejector seat becomes functional only in the highly undesirable event of a hit or some other disaster; the seat may well be aware of the context it runs in: on the ground it is disabled. While Parnas' paper is certainly one of the main roots of current exception mechanisms, we find the term *undesired event* not appropriate when the roof is on fire. Instead, we suggest the term *emergency* which is introduced in Chapter IV.

## III. Exceptions and Programming Languages

Programmers have always felt the need for a supplementary information channel from the callee back to the caller: doubtful constructs like global state variables, message areas and the like have been in use to this day. Just returning one or more values is not enough. This is even more so in object oriented languages: Constructors return nothing; overloaded operators return exactly one value. So, there is no room to inform the caller about failures. This is the syntactic reason for object oriented languages to provide an exception mechanism: you return either nothing, a value, xor an exception.

Exceptions are a supplementary way for passing information to the caller, which can be used for any purpose. [Goodenough1975] mentions three: signaling failure, classifying a result (e.g. overflow, end of file) and monitoring (e.g. "that many records have been processed"). So, exceptions are not necessarily exceptional, but may well be part of the normal business. In Java, only runtime exceptions such as *ClassCastException* or *NullPointerException* indicate severe problems; checked exceptions are used for many different purposes (e.g. *InterruptedException*, *NumberFormatException*), which are not exceptional at all, but happen all the time. So, the term *exception* is not restricted to failures, undesired events or emergencies.

Designing sound exception mechanisms turned out to be extremely difficult. [Howell1991] complains how poorly Ada exceptions match with object orientation, Ada scoping rules, and concurrency. C++- and Java exceptions match with object orientation and the less harsh scoping rules of these languages, but C++-exceptions are completely unaware of concurrency; in Java, exceptions never leave the thread of their origin. The *InterruptedException* plays a very special role: it is thrown behind the scenes and serves for synchronizing threads.

Java distinguishes checked and unchecked exceptions. Checked exceptions must either be handled within the method itself or they are part of its signature as in:

```
void foo() throws RemoteException { .. }
```

While it is a good idea to be honest about possible failures, experience shows that checked exceptions tend to spread all over the system. To see this, look at the *foo*-example: The programmer calling *foo* has probably no idea how to handle a *RemoteException* and will therefore propagate it to the next level. So, *RemoteException*s will show up everywhere, creating an undesired dependency and making changes hard if not impossible. What should be done is to pass *RemoteException*s to the next safety facade as discussed in Chapter VI.

Exceptions don't need to be handled immediately. They fly to the next matching catch block, which can be at any distance; in the worst case, exceptions are caught by the outermost catch block, i.e. the one of the runtime system. So, exceptions have a lot in common with the notorious goto-statement; in fact, they offer similar opportunities for misuse.

Exceptions tend to reveal implementation details not intended for the caller. An example would be a stack implementation throwing an *IndexOutOfBoundsException* [Howell1991]. This information is welcome for debugging; at runtime it is useless.

Exceptions are for free if they don't happen, but very costly if they do. In the following example the first method is 750 times slower than the second one (under Windows XP and Java 1.4) if called with non-*Integers*; if called with *Integers*, there is almost no penalty:

```
public static boolean testForInteger1(Object x) {
  try {
    Integer i = (Integer) x;
    return true;
  }
  catch (Exception e) {
    return false;
  }
}


public static boolean testForInteger2(Object x) {
  return x instanceof Integer;
}
```

As a consequence, [Cunningham2002] goes back to the roots and suggests that *exceptions are exceptional*: they should only be used for rare events, not for the normal control flow – a good idea, but not in agreement with the actual use of exceptions in Java.

## IV. Components, Emergencies and Assertions

We use the term component in the usual meaning (cf. [Szyperski1998]): There is at least one interface and one implementation. The caller calls operations defined by the interface; he is unaware of the implementation. In Java, a component is implemented by means of packages; the top level package contains everything the caller needs to use the component. This top level package typically contains one or more Java interfaces and the classes (including exception classes) you need to use these interfaces. A component can *call* any number of other components.

A component is either in its normal state (that is, it is able to process calls), or it is not. In the latter case an *emergency* has occurred: an emergency is a situation where the programmer of the component doesn't know what to do – there is no *local help* available: the component

where the emergency happened is unable to solve the problem; the calling component however may or may not recover.

Emergencies range from programming errors to unreachable databases and crashed neighbor systems. They could be stated as violated assertions, but assertions (and in particular *assert* in Java 1.4) are often used for debugging purposes only; many programmers switch them off once the system has been tested. Emergency handling, however, cannot be switched off.
All Java runtime exceptions are emergencies from the JVM's point of view: the JVM has no way to solve the problem. At the programming level, we encounter a lot of emergencies as well: the database is not available, an SQL-statement is incorrect, or a neighbor system returns a meaningless value. The number of potential emergencies is as huge as the number of possible reasons for a car to break down: a complete enumeration is impossible.

Inexperienced programmers tend to invent and implement ad-hoc repair measures. If this happens at a large scale, the system is doomed to failure. It is the job of the system architect to precisely define for each component what an emergency is and what is not. Let us look at the case of an incorrectly working neighbor system: if this situation is considered to be an emergency the calling component has the right to give up – and that is cheap to implement. If it isn't, then you have to design a possibly complicated and expensive fallback procedure! The emergency-decision is always *binary* (there is no warning level) and *local* for the component: A *find*-operation may find zero, one or many matching objects – that's normal business, but the caller may consider a zero-result as an emergency. The *find*-operation would consider itself to be in an emergency if the database connection is down, but the caller may try to reconnect the database and call *find* again.

We suggest using a simple class *Emergency* throwing an *EmergencyException*:

```
        public class EmergencyException extends RuntimeException {
    public EmergencyException(String message) {
      super(message);
    }

}

        public class Emergency {

          public static void ifTrue(boolean condition, String message) {
            if (condition)
              throw new EmergencyException(message);
            }

          // ifFalse, ifNull, now, ...

        }
```

So you would code

```
        public void foo() {

          String result = ..              // must never be null
          Emergency.ifNull(result, ...);
        }
```

Emergencies should be detected as early as possible. Detected emergencies are not desirable, but can be handled (cf. Chapter VI). The later you detect an emergency the more damage it might have caused and the less debug information you will get. Emergencies not detected by the program finally lead to a crash with no or little information about the reasons – the best you can get is a dump (cf. [Denert1991]).

| | **detected** | **undetected** |
|---|---|---|
| **normal state** | ok | never happens |
| **emergency** | emergency handling | crash |

So there shouldn't be any undetected emergencies left, but this is not easy to achieve.

## V. Application Errors

Methods can also fail for application reasons: you cannot withdraw money from an overdrawn account. These are application errors as opposed to emergencies; they are part of the normal business. In general, there are only a few possible application errors: often it is sufficient to just return *ok* or *nok*; many methods (e.g. *getter*-methods or *rollback*) cannot fail at all from the application point of view. Application errors can and should be fully enumerated. They are completely different from emergencies.

There are two ways to inform the caller about application errors: return values and exceptions. Our advice is: use return values whenever possible, use exceptions (in Java: checked exceptions) otherwise. Example: *find*-methods should return *null* or an empty list if there are no matching objects. Using an exception to report the *found-nothing* result makes the code more complicated and is much slower if *found-nothing* is frequent. Another example: A *withdraw*-method that normally returns the balance after withdrawal should throw a checked exception if the account is overdrawn (and for performance reasons, we hope that this rarely happens).

There is a hard rule: Exceptions representing application errors must be handled immediately by the caller and not by some hidden exception handler. They don't fly. If they did you would invariably end up with a scattered, goto-like control flow. This rule implies that checked exceptions are far from being ideal for handling application errors: They are slow if the error occurs and we only use a tiny part of the exception machinery.

When designing return codes or exception classes, one should carefully separate control flow on the one hand and messages for the user on the other hand. It is quite common to have a large number of different messages, which inform the user about success or failure, but only two possible outcomes affecting the control flow: *ok* and *nok*. Our advice is to use self implemented exception classes only if they do affect the control flow.

## VI. Emergency Handling and Safety Facades

Emergencies are *not* handled by the immediate caller. Let us first discuss how emergencies can be handled at all and then ask who takes care. There are four ways to handle emergencies: *ignore*, *retry*, *call an alternative*, or *resign*. Resigning means: minimize damage, write log information, and signal *definite* and *safe* failure. *Definite* means: it makes no sense to try any harder, and *safe* says that all damage reducing measures have been taken.

The first option is used rarely and only mentioned for completeness: You just don't care about success or failure. Retry can be useful but should be handled carefully because cascaded retries multiply. In some cases there is an alternative you can call: if the main database is down, there might be a local fall back database. The most frequent option is to resign: minimize damage by undoing or neutralizing side effects[2], freeing resources and protocol the sad event. Once the emergency has been handled, there are only two possible outcomes: success or definite and safe failure.

All this is obvious, but who takes care of all this? Here's the answer: you can call a component *safely* or *unsafely*. An unsafe call calls the callee directly with no emergency handling in between. That is, caller and callee form a *risk community*: they succeed together or they perish together. Risk communities grow by transitivity: if component *a* calls component *b* unsafely then *a* and *b* belong to the same risk community.

Safe access to risk communities is provided by *safety facades*. The safety facade – a special case of the well-known facade pattern [Gamma1995] – is in charge of emergency handling. All emergencies detected within the risk community fly over all involved components and are finally caught by the safety facade. This includes all runtime exceptions as well. Risk communities must be designed carefully; all components belonging to the same community share the same emergency handling mechanism. The safety facade provides the context the risk community runs within. This architecture is quite opposed to the idea of plugged-in emergency handlers.

Safety facades can and will usually be cascaded: emergencies are handled by the nearest safety facade; the outcome (success or definite failure) would be reported to the calling component which is free to consider the definite failure as an emergency or not: a batch processing one million records would just protocol and skip unreadable records. So, at each stage the emergency can either be masked or propagated to the next safety facade, job abortion being the last resort at the outermost level (probably the main program).

What we are presenting can be considered as just a modern version of the trap mechanism suggested in [Parnas1976].

## VII. Pre- and Postconditions

Pre- and postconditions are part of the contract between caller and callee (cf. [Meyer1995, p.16]). This is an old idea: [Goodenough1975] discusses *domain failures* (some input assertion is tested and not satisfied) and *range failures* (the operation fails to meet its output assertion). In an ideal world, pre- and postconditions just hold – they must be valid, but in practice one has to decide how to use them, how and where they are checked and what to do if they don't hold.

[2] In practice, this is often done by a database rollback.

Preconditions are meant to protect the called component from illegal calls. They are stated in terms of the input parameters and/or the component's state. It is a good idea to assume all input parameters to be non-null as an implicit precondition and to allow null explicitly.

It is the caller's responsibility to make sure that all preconditions hold true. Thus, violated preconditions are the caller's problem, not the callee's. The callee would just *reject* a call if at least one precondition is violated. This is an emergency from the caller's point of view, but the callee doesn't care.

We suggest using a simple class *Reject*, which throws a *ViolatedPreconditionException*:

```
        public class ViolatedPreconditionException extends RuntimeException {
    public ViolatedPreconditionException(String message) {
      super(message);
    }

}

        public class Reject {

          public static void ifTrue(boolean condition, String message) {
            if (condition)
              throw new ViolatedPreconditionException(message);
          }

          // ifFalse, ifNull, now, ...

        }
```

So you would code

```
        public void foo(String s) {

          Reject.ifNull(s, ...);     // s must not be null
        }
```

Preconditions must be designed carefully: weak preconditions mean more work for the called method; strong preconditions more work for the caller. So, a *square root* function would reject negative input values, but a *matrix inversion* would accept all non-null square matrices regardless of their rank – if the caller had to compute the rank, it could invert the matrix as well. The safety facade (cf. Chapter VI) – if present – catches all exceptions but the *ViolatedPreconditionException*, which is passed unhandled to the calling component.

Postconditions are completely different from preconditions: postconditions protect the caller against erroneous implementations. So, it is mainly the caller's interest to check postconditions: the less confidence you have in a given implementation (e.g. when writing a test driver) the keener you are on checking postconditions. The implementation itself is a bad place for checking postconditions and leads to silly code like the following:

```
        int add(int a, int b) {

          int result = a + b;
          assert result == a + b;
          return result;
        }
```

## VIII. Ten Rules

1. Have a clear distinction between emergencies and application errors.
2. Detect emergencies as early as possible.
3. Reject calls if there is a violated precondition.
4. Assume all input parameters to be non null by default.
5. Design risk communities accessed by safety facades.
6. Concentrate emergency handling in safety facades.
7. Let safety facades catch all exceptions but the *ViolatedPreconditionException*.
8. Report application errors using special return values (e.g. null) if possible. Use checked exceptions otherwise.
9. Handle application errors immediately.
10. Don't use self implemented exception classes unless they are necessary for the control flow.

## IX. References


| | |
|---|---|
| [Cunningham 2002] | W. Cunningham, A. Hunt, D. Thomas: *The Pragmatic Programmer*. Addison-Wesley, 2002. |
| [Denert1991] | E. Denert, J. Siedersleben: *Software-Engineering*. Springer Verlag, 1991. |
| [Gamma1995] | E. Gamma, Helm, Johnson, Vlissides: *Design Patterns*. Addison-Wesley, 1995. |
| [Goodenough1975] | J.B. Goodenough: *Exception Handling: Issues and a Proposed Notation*. CACM, 18, 12, pp. 683-696, 1975. |
| [Howel1991] | C. Howell, D. Mularz: *Exception Handling in Large Ada Systems*. Washington Ada Symposium, Washington, June, 90-101, 1991. |
| [Meyer1995] | B. Meyer: *Object Success*. Prentice Hall, 1995. |
| [Parnas1976] | D.L.Parnas, H.Würges: *Response to Undesired Events in Software Systems*. Proceedings of the Second International Conference on Software Engineering, 1976, pp. 437-447 |
| [Siedersleben2002] | J. Siedersleben (Ed.): *Software-Technik*. Hanser Verlag, 2002. |
| [Szyperski1998] | C. Szyperski: *Component Software*. Addison-Wesley, 1998. |


Acknowledgments:
The author is grateful to two anonymous referees and to Ernst Denert for being an indefatigable discussion partner. The term *emergency* was coined on January 9, 2003 in his office in Berlin.

# Analyzing Exception Usage in Large Java Applications

Darrell Reimer and Harini Srinivasan

IBM Research, 19 Skyline Drive, Hawthorne, NY, USA 10532
{dreimer, harini}@us.ibm.com

**Abstract.** The Java programming language provides a way of identifying when semantic constraints of the program are violated using its *exception* mechanism. Whenever a semantic constraint in the program is violated, control flow is transferred from the point where the exception happened (throw site) to the point specified by the programmer (catch site). While this is indeed a robust and portable mechanism of handling semantic errors and exceptional program behavior, the mechanism is often misused and/or abused. In our experience working with large J2EE applications, we have encountered several inappropriate exceptions usage patterns that have made maintainability of these applications extremely difficult. Proper exception usage is necessary to minimize time from problem appearance to problem isolation and diagnosis. This article discusses some common trends in the use of exceptions in large Java applications that make servicing and maintaining these long running applications extremely tedious. The paper also proposes some solutions to avoid or correct these misuses of exceptions.

## 1 Introduction

The Java Virtual Machine [12] uses exceptions to signal semantic errors in a program. In particular, whenever a semantic error occurs, the JVM raises an exception. It is the responsibility of the application programmer to both (a) identify when such exceptions can happen, i.e., when semantic errors can happen and (b) catch these exceptions in a manner that helps identify them during program execution. Exceptions can also be used to *remedy* an incorrect execution behavior of the application.

Proper handling of exceptions is extremely important to be able to manage and service large J2EE applications. For example, consider a J2EE application that handles online banking transactions. Typically, the financial institution would like the application to run in a 24x7 mode to be able to service their customers continuously. The cost of stopping and starting these applications is usually very high for these institutions. Given this scenario, if a failure happens during application program execution, it is extremely important to be able to quickly locate the point of failure. In particular, if an exception is not logged, once a failure occurs, additional logging must be added, the application restarted, and the problem must be reproduced. Restarting such 24x7 applications is highly undesirable. Most of the J2EE API methods whose execution can

result in failures are designed to throw exceptions. Examples of such methods are those that result in interactions with other components of the application architecture, for example, database, network, LDAP etc. The most desirable programming practice that can help in tracing failure points is to catch the specific exceptions thrown by these methods and output some kind of log information indicating the failure. Logging the failure this way helps in understanding what happened during program execution later on. However, in the large J2EE applications we have worked with, we noticed that this exception handling practice is not common.

Before proceeding to talk about exception usage patterns in these applications, we give a brief overview of the exception mechanism in Java (Section 2). This section is not intended to be a tutorial of Java exceptions. The reader is advised to consult the Java language specification and the Java Virtual Machine specification for details on language and implementation semantics of Java exceptions. Section 3 discusses exceptions usage patterns in a handful of large J2EE applications we have worked with. This section is the primary contribution of this paper. In Section 4, we discuss approaches to solve this problem of improper exception handling. Section 5 discusses related work in the area of understanding exceptions usage in Java applications.

## 2 Overview of Java Exceptions

An exception can occur under one the following circumstances [11]:

- An abnormal execution condition was synchronously detected by the JVM. For example, integer divide by zero, array out of bounds, out of memory error, loading or linking errors.
- A **throw** statement was executed.
- An asynchronous exception occurred because the **stop** method of class Thread was invoked or an internal JVM error occurred.

The Java programming language defines class Throwable and allows the application programmer to extend this class. The Throwable class and its subclasses are collectively referred to as exception classes and instances of these classes are used to represent exceptions. Among these exceptions, all exceptions that are instances of class RunTimeException and its subclasses and exceptions that are instances of class Error and its subclasses need not be checked, i.e., these exceptions need not be explicitly handled by the application. All other exceptions are *checked exceptions* and need to be explicitly handled in the program. The language provides the `try` and `catch` clauses to define exception handlers. During the process of throwing an exception, the Java virtual machine abruptly completes, one by one, any expressions, statements, method and constructor invocations, initializers, and field initialization expressions that have begun but not completed execution in the current thread. This process continues until a handler is found that indicates that it handles that particular exception by naming the class of the exception or a superclass of the class of the exception. If no such handler is found, then the method uncaughtException is invoked for the

ThreadGroup that is the parent of the current thread-thus every effort is made to avoid letting an exception go unhandled. [11]. The following is a simple try – catch combination:

```
// in the main method:
try() {
       foo();
} catch (MyException me) {
       System.err.println(me);
} finally {
    clearResources();  // code to close resources.
}
```

The method `foo()` or one of its callees can potentially throw an exception of type `MyException` (or a subclass of `MyException`). `MyException`, in this case, is a checked Java exception. When such an exception is thrown, control is transferred from the call site of `foo()` to the beginning of the catch block and execution proceeds from that point on. When a method such as `foo()` throws an exception, the signature of the method must advertise the specific exception(s) thrown. What happens inside the `catch` clause is still up to the application programmer. It is possible that the exception is rethrown using a `throw` statement, or the exception is logged or some code is executed or nothing at all happens. There is also the `finally` clause to a `try` statement. The semantics of the `finally` block is that it is always executed. Whether a catch clause executes or not, the code within a `finally` is always executed. In the above code snippet, the method clearResources() executes both when the program exhibits normal and exceptional control flow. The Java API itself defines a number of checked and unchecked exceptions.

## 3. Exception usage in Large Java Applications

From our experience working with large J2EE applications, we have observed the following exception usage patterns that have hindered the maintainability and serviceability of these applications. All these applications are “real-life” i.e., customer applications that have been deployed and in production.

### 3.1 Swallowed Exceptions

Exceptions should not be ignored through empty catch blocks. In general, every path out of a `catch{}` block should result in the exception being logged or the exception being re-thrown or have some kind of *remedial code* that remedies the exceptional execution behavior. If a handler block has neither logging code nor a rethrow, we refer to the corresponding exception as *swallowed.* The following is an example of a swallowed exception:

```
// example swallowed exception
```

```
try{
    foo();
} catch (MyException me) {
}
```

where, there is no rethrow of an exception or some kind of logging code to record that the exception happened within the catch block. In one very large e-business customer application, the application code had ignored many exceptions, i.e., contained empty catch blocks without any logging information. Consequently, when such a system failed in production, it was extremely hard to track the cause of failures. Ignored exceptions severely impaired the effectiveness of monitoring systems in this application environment. For example, the application had a try block that had an SQL update in it. The corresponding catch was empty. [An SQL update (`executeUpdate()`) method call is part of the J2EE API to access the database component of the application. The method call is usually implemented in a driver that interfaces with C code that exchanges information with the Database system, typically via sockets. ] In this case, if the update failed, there will be no record of it for the application server.

However, under certain circumstances, a catch block without any logging code or rethrow is perfectly fine. These are usually cases where exception constructs are used to manage normal program control flow, or certain exceptions caught need not be logged, e.g., `InterruptedException`, or the `catch` block contains *remedial* code. Listed below are a few examples:

```
1. // example OK swallowed exception case
     key = null;
     try {
        key = foo();
     } catch (MyException me) { return key; }
     return key;
```

In the above example, the value of variable `key` is set to null before the `try` statement. When `foo()` throws an exception, the catch block executes but does not have to set the value of `key` to be `null` again. The exception thrown appears to be swallowed because of the absence of logging code or a throw statement within the catch block. Interestingly, in this example, if the programmer did not intend to use the try-catch for normal control flow, the calling method will likely see a `NullPointerException` that is not caught causing debugging nightmares.

```
2. // example OK swallowed exception
   try {
      key = foo();
   } catch(MyException me) {key = ValMaybeNull; }
   return key;
```

In this example, a semantically valid assignment to the variable `key` occurs within the catch block which can potentially execute as normal program control flow. A variation of this example is when `key` is initialized to `null` prior to the try block and the handler has no code in it.

```
3. // example OK swallowed exception – remedial code
     sentFlag = false;
     while(!sentFlag){
      for(int i=0; i<connectRetry && !sentFlag; i++){
           try {
                 send_something();
                 if (success) sentFlag = true;
           } catch (Exception ex){
                 Thread.currentThread().sleep(SLEEP_TIME);
                 retryCount=i; // used to track #failures
           }
      }
     }
```

In the above example, taken from a real J2EE application, whenever a failure occurs during the send, i.e., in method `send_something()`, an exception is thrown. The `while` loop iterates until the method execution succeeds. This is an example where the exception causes remedial code to be executed.

### 3.2 Single catch block for multiple exceptions

If exceptions are caught in the same block, it should be possible to identify which exception was handled by the exception handler by the logging. However, several times, we have encountered the following (undesirable) code in these applications:

```
// exceptions are not handled individually
try {
   foo();
   bar();
} catch (Exception e) {
   System.out.println("catching exception" + e);
}
```

The above example also points out the case where exceptions are *subsumed*. The following is preferred for debugging, where `foo()` can throw the specific exception `MyException` and `bar()` can throw the exception `MyException1.`

```
try {
  foo();
  bar();
} catch (MyException me) {
  System.out.println("MyException raised " + me);
} catch (MyException1 me1) {
  System.out.println("MyException1 raised " + me1);
}
```

Within a try block, exceptions of the same type should not be raised at multiple program points. If not, it will be difficult to identify within the catch block which program point (call site) raised the exception resulting in debugging difficulties. For example,

```
// multiple program points raising same exception
try{
   foo();
   bar();
   Object obj = baz();
} catch (MyException me1) {
    System.out.println("me1 raised by bar");
} catch (MyException me) {
    System.out.println("me raised by baz or foo");
}
```

where both methods `foo()` and `baz()` can throw an exception of type `MyException`.

Likewise, if an exception is being re-thrown, the application should avoid mapping multiple exceptions to the same exception since this hides problem sources from debuggers.

### 3.3 Exceptions not handled at appropriate level

Another coding pattern that makes debugging difficult is when exceptions are not handled close to the source of the exception. If exceptions are propagated a long way up the call chain, the error message and handling will become less meaningful and debugging much more difficult.

### 3.4 Log verbosity in catch blocks

A common coding style in a handler that can do as much harm as good is:

```
log("some exception happened");
e.printStackTrace();
```

For example, consider a system under a load surge – some resource in the system becomes overloaded, and *temporarily* fails, resulting in several exceptions getting thrown. Normally, this would just affect the requesting users, but if the exception handling is overly heavy (e.g. lots of I/O, and getting the stack trace), it just adds more load to the system, causing a cascading failure that feeds on itself.

### 3.5 Application Statistics

The table below shows some statistics on a handful of J2EE applications that we analyzed. The results show the number of swallowed exceptions as defined in this section, after filtering out cases that have `return` statements in the catch blocks and catch blocks that do not have to log exception information. The applications A1-A5 listed in this table are all J2EE customer applications and hence the names are hidden. The application PetStore is a J2EE sample application published by Sun Microsystems. The #classes, #methods and #handlers columns report the numbers in just the application code, not including the J2EE and J2SE libraries. The last row shows the results on JDK1.3.1 rt.jar J2SE library classes.

| Application | # classes | #methods | #handlers | #swallowed | #after filtering |
|---|---|---|---|---|---|
| A1 | 1724 | 19387 | 9951 | 182 | 162 |
| A2 | 781 | 8509 | 492 | 74 | 35 |
| A3 | 666 | 9355 | 16770 | 80 | 47 |
| A4 | 1151 | 10458 | 3373 | 97 | 27 |
| A5 | 2202 | 30964 | 16215 | 444 | 350 |
| PetStore | 353 | 2001 | 428 | 22 | 11 |
| rt.jar | 5484 | 46723 | 3670 | 974 | 723 |

The following table gives additional results on false positives for some of the applications, i.e., what goes on inside the handlers for some of the applications:

| Application | #handlers | # handlers with calls | #handlers with re-throws | #handlers with loads/stores |
|---|---|---|---|---|
| A1 | 9951 | 9363 | 0 | 27 |
| A4 | 3373 | 3119 | 0 | 18 |
| A5 | 16215 | 15487 | 0 | 54 |
| PetStore | 428 | 372 | 0 | 1 |
| rt.jar | 3670 | 2242 | 0 | 77 |

Note that a majority of the handlers had calls in them and in the applications we analyzed, we did not come across exception re-throws. When we examined application A1 in more detail and looked at the calls made inside the handlers, we noticed that a majority of these calls were not to logging code, but calls to application code doing business logic of some kind.

## 4. Approaches to handle the problem

How can exception handling be made more effective? Most of the J2EE API methods require the programmer writing these applications to enclose the methods in try – catch blocks. It appears the kinds of improper exception handling that we discussed in the previous section happens due to lack of rigor in writing these applications. For example, consider an application method that (1) gets a database connection using the `javax.sql.DataSource.getConnection()` call, (2) creates one or more database SQL statements using `java.sql.Connection.createStatement()` (3) executes the SQL statements created (that may be updates or queries to the database) using the `java.sql.Statement.executeUpdate()` or `java.sql.Statement.executeQuery()` methods and, (4) finally processes the results from the database using the `java.sql.ResultSet` interface methods. Almost all of these methods throw `SQLException`. It is tempting for the application developer (who does not practice rigor) to either (a) enclose all these methods within a single `try – catch` block that catches an `SQLException` or, (b) enclose each of the

above method calls in a `try – catch` block that catches an `SQLException`, but the handlers do not log any information. We have encountered both these coding patterns in the applications we have worked with. As mentioned in the previous section, both these exception coding patterns make debugging an exceptional condition during program execution extremely tedious. A log of which of the database operations caused the exception to occur will be extremely helpful in debugging not only the application on the Java side, but also any database errors.

Several approaches are possible to handle this "bad coding practice" problem:

- The programming environment (an integrated development environment, IDE) automatically inserts try – catch blocks for method calls whose signature has the `throws` clause in it. In addition, such an IDE could also insert a default print method call that logs some (minimal) information about the exception being caught by the handler. This approach saves a lot of trouble on the programmer's side and also reminds the programmer to log exceptions. However, a drawback of this approach is that, in cases where the try – catch block is used to capture normal control flow (see examples in Section 2), the programmer has to explicitly undo some of the operations of the IDE. While one could argue that try – catch – finally is used for normal control flow only rarely, when actually used, undoing the work of the IDE can be annoying to the developer.
- Another approach is to statically analyze the application program and point out program points in the application where exception handling has not been implemented properly. For example, check the application for swallowed exceptions, multiple exceptions handled by a single catch block, catch block not catching the exact exception thrown but it's supertype etc. Such an analyzer can be integrated as part of an IDE that checks for bad coding patterns. This is the approach that we have used in our tool called *SABER*. SABER does static program analysis (control and data flow) to check for many bad coding patterns, including swallowed exceptions and handlers catching supertype exceptions. The tool is integrated into the WebSphere Studio development environment and reports messages to the programmer in a manner within this IDE that links the error message to the program point where the bad coding pattern appears.
- Finally, is it possible for the JVM to provide more information? Typically, when an exception is thrown, the JVM dumps a stack trace. However, if the exception is rethrown within the catch block, the stack trace will include method calls only from the point where the exception was actually caught. Another factor prohibiting debugging of exceptions is when the JIT is on. Most JVMs do not provide line number information in the stack trace when the method has been JIT-compiled. The optimizing compiler should keep track of this information and convey the line number of the method invoked that caused the exception to occur. This approach will still not be able to provide any other logging information other than the stack trace, which is not always sufficient in debugging the problem.

## 5. Related Work

Several research papers have looked into proper handling of exceptions and have studied the control flow aspects of exceptions. Robillard and Murphy [5][6] describe their tool called Jex that can be used to illustrate to the programmer the structure of exceptions in application code. Based on the exception control flow, the programmer will be able to identify program points where exceptions are caught accidentally, error handling procedures are not being followed or finer-grained recovery code can be added in the program. Jex analyzes exception control flow and identifies exception subsumption, i.e., wherever a precise exception is not raised, and unhandled exceptions. By presenting the resulting information to the application programmer, the tool allows the developer to encode handlers for exception types that are missing, there by increasing the robustness of the code. Our work is along the same lines as Robillard and Murphy's and, we look at a wider range of exception usage issues *including* subsumed and unhandled exceptions within method bodies. We also present results of exception usage on large real-life applications that are typically developed by multiple development organizations and hence exhibit varying coding styles and conventions.

Ryder *et al* [10] describe another static analysis tool, JESP, for examining the usage of exceptions. The paper provides empirical results of exception usage on Java benchmarks and discusses the implications of the results on compiler optimizations. We have observed that not all the empirical results (#exception constructs (try, catch, finally), distance between the throw and the corresponding catch, prevalence of user-defined and Java-defined exceptions, #exception classes and the shape of the exception hierarchy) apply to larger Java applications that we have analyzed.

A number of other papers talk about control flow representations of programs written in languages that support exceptions: the Marmot compiler [9], and Choi *et al* [4] for Java, the Vortex compiler [3] that supports Modula-3, Java and the Cecil languages, Chatterjee *et al* [8] that talks about modeling exceptions in an interprocedural control flow graph. Another paper that raises issues related to flow analysis of Java programs in presence of exceptions talks about instruction scheduling in presence of these constructs [7].

Stevens [1,2] studies exception control flow in Java programs and the negative impact of this type of control flow on compiler driven optimizations. He also discusses approaches to reducing this effect using static, whole program analysis on the byte code representation of Java programs.

Romanovsky and Sander [13] talk about misusing exception handling in Ada programs that have a lot in common with how exceptions are misused in Java.

Miller and Tripathi [14] discuss how object-oriented techniques interact with exception handling and which OO features conflict with current exception handling mechanisms.

## References

1. Andrew Stevens, Des Watson. The Effect of Java Exceptions on Code Optimisation.

In *European Conference on Object Oriented Programming 2000: Exception Handling in Object Oriented Systems. Workshop Talk*

2. Andrew Stevens *JeX: An Implementation of a Java Exception Analysis Framework to Exploit Potential Optimisations*, University of Sussex, 2001.
3. Jeffrey Dean, Greg DeFouw, David Grove, Vass Litvinov and Craig Chambers. Vortex: An optimizing compiler for object-oriented languages. *In ACM Conf. On Object Oriented Programming Systems, Languages and Applications*, pages 83-100, October 1996.
4. Jong-Deok Choi, David Grove, Michael Hind, and Vivek Sarkar. Efficient and precise modeling of exceptions for the analysis of Java programs. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE99)*, Sep 1999.
5. Martin P. Robillard and Gail C. Murphy. Analyzing Exception Flow in Java Programs. In *Proceedings of the 7th European Software Engineering Conference and 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering (Toulouse, France)*, volume 1687 of *Lecture Notes in Comp. Sci.*, pages 322-337. Springer-Verlag, Sep 1999.
6. Martin P. Robillard and Gail C. Murphy. Designing Robust Java Programs with Exceptions. In *Proceedings of the ACM SIGSOFT Eight International Symposium on the Foundations of Software Engineering (FSE-8): Foundations of Software Engineering for Twenty-First Century Applications* (San Diego, California, USA), ACM Press, pages 2-10, November 2000.
7. Matthew Arnold, Michael Hsiao, Ulrich Kremer, and Barbara G. Ryder. Instruction scheduling in the presence of java's runtime exceptions. In *Proceedings of Workshop on Languages and Compilers for Parallel Computation (LCPC'99)*, August 1999. http://citeseer.nj.nec.com/arnold99instruction.html
8. Ramakrishna Chatterjee, Barbara Ryder, William A. Landi. Complexity of concrete type inference in the presence of exceptions. In *Lecture Notes in Computer Science*, 1381, pages 57-74. Springer-Verlag, April 1998. *Proceedings of the European Symposium on Programming.*
9. Robert Fitzgerald, Todd B. Knoblock, Erik Ruf, Bjarne Steensgard and David Tarditi. Marmot: An optimizing compiler for Java. Technical Report MSR-TR-99-33, Microsoft Research, June 1999.
10. Barbara G. Ryder, Donald Smith, Ulrich Kremer, Michael Gordon and Nirav Shah. A Static Study of Java Exceptions using JESP, *In Proceedings of CC 2000*, pp 67-81.
11. James Gosling, *Bill Joy, Guy Steele, Gilad Bracha. The Java Programming Language, Second Edition.* Addison-Wesley, 1997.
12. Tim Lindholm, Frank Yellin. *The Java Virtual Machine Specification. Second Edition.* Addison-Wesley, 1999.
13. A. Romanovsky and B. Sanden. Except for Exception Handling, *Ada Letters*, v. XXI, 3, pp. 19-25, 2001
14. Robert Miller and Anand Tripathi, Issues with Exception Handling in Object-Oriented Systems, *European Conference on Object-Oriented Programming, (ECOOP)*, June 1997

# Bound Exceptions in Object Programming

Peter A. Buhr and Roy Krischer

University of Waterloo, 200 University Ave. West,
Waterloo, Ontario, CANADA, N2L 3G1
{pabuhr,rkrische}@plg.uwaterloo.ca

## 1 Introduction

In this discussion, exception handling is treated purely as a control-flow mechanism versus a class-specific or a software-engineering mechanism [7]. From the perspective of control-flow, routine and exceptional transfers can be characterized by two properties: 1) static/dynamic call, where the routine/exception name at the call/raise is either looked up statically or dynamically, and 2) static/dynamic return, where completion of a routine/handler returns to its static or dynamic context; resulting in the following language capabilities:

| | call/raise | |
|---|---|---|
| return/handled | static | dynamic |
| static | sequel | termination |
| dynamic | routine/member | resumption |

While several main-stream object-oriented programming languages (OOPL) provide exception handling mechanisms (EHM), *e.g.* Ada, C++, Modula-3, Java, C#, to deal with *ancillary* control flow, the EHM is disjoint from the object paradigm. This position paper reexamines previous ideas for binding exceptions to objects, as well as suggesting some extensions to these ideas.

## 2 Bound Exceptions

Most EHMs usually rely on *only* the exception type to find a matching handler. For example, the exception FileErr raised during a file operation is caught by any handler for FileErr. The problem is the lack of connection between the exception and the object raising the exception. This lack of specificity makes it difficult to distinguish among multiple files in a catch clause when one file raises an exception, which may be crucial to correct handling. Hence, matching only by type is insufficient in complex situations, and especially in object-oriented systems. For example, in the left example of Figure 1, FileErr exceptions can be raised by the doWrite method of logFile, dataFile and tmpFile. For the matching-by-type strategy, all exceptions are handled by a single unbound handler, regardless of which object actually causes the error (note, SpecialFileErr is derived from FileErr). In many circumstances, it is unlikely that errors from three different file objects can be uniformly handled by a single handler. When appropriate, it should be possible to use a separate handler for each file object raising an exception.

```
class FileErr { ... };
class SpecialFileErr : FileErr { ... };
class File {
   void doWrite() {
      ... throw FileErr(); ...
   }
};
class SpecialFile : public File {
   virtual void doWrite() {
      ... throw SpecialFileErr(); ...
   }
};
File logFile;
File dataFile;
SpecialFile tmpFile;
try { ... logFile.doWrite(); ...
      ... dataFile.doWrite(); ...
      ... tmpFile.doWrite(); ...
} catch( FileErr ) { ... }
```

```
procedure BoundExceptions is
   generic package File is
      FileErr : exception;
      procedure doWrite;
   end File;
   package body File is
      procedure doWrite is
      begin ... raise FileErr; ...
      end doWrite;
   end File;
   package dataFile is new File;
   package logFile is new File;
   begin
      ... dataFile.doWrite; ...
      ... logFile.doWrite; ...
   exception
      when dataFile.FileErr => ...
      when logFile.FileErr  => ...
end BoundExceptions;
```

**Fig. 1.** Unbound (C++) versus Bound (Ada) Exception Matching

From an object-oriented standpoint, the conventional matching-by-type handling of exceptions is inconsistent. Objects are the main components in an object-oriented software design, and their actions determine program behaviour. Hence, an exceptional situation is (usually) the result of an object's action, suggesting the object responsible may need to be associated with the catching. While prior discussion exists on associating objects with exceptions for Ada [3], C[2], Lisp [8], Smalltalk [4] and Beta [5], none of this work has had an effect on main-stream OOPLs. Therefore, it is important to strongly reiterate this crucial object-oriented exception-handling design objective. This discussion focuses almost exclusively on this specific point.

For example, Ada has a partial solution by binding an exception to a package instance, so the same exception originating from different instances can be handled separately. In the right example of Figure 1, the FileErr exception is declared *inside* the generic package File. Pseudo-objects are created for both instances of File, *i.e.*, dataFile and logFile. Then it is possible in the handler to bind the FileErr exception with dataFile and logFile using the dot-notation. Matching during propagation now uses both the exception type and the object raising the exception. However, Ada disallows an unbound version of FileErr in a generic package, precluding the ability to handle some cases as instance specific and others by a general handler. While Lisp/Smalltalk/Beta can mimic bound exceptions, it is done through mechanisms that do not or cannot exist in main-stream statically-typed OOPLs, *e.g.*, continuations, dynamic typing, reflection, runtime compilation, virtual-machine. But most importantly, mimicking is a programming convention versus a language construct; we strongly believe this is a situation in which it is necessary to cast a convention into a specific construct, to ensure correct usage and for efficiency.

The following example construct for binding objects and exceptions is inspired by the Ada example:

```
try { ... logFile.doWrite(); ...
      ... dataFile.doWrite(); ...
      ... tmpFile.doWrite(); ...
} catch( logFile.FileErr ) {...}          // bound
  catch( dataFile.FileErr ) {...}         // bound
  catch( FileErr ) {...}                  // unbound
```

The dot-notation is an extension to the catch argument, where **catch(** *object* . *exception-type* **)** only matches if the raised exception is of type *exception-type* and bound to *object*. This syntax is backwards-compatible but unusual as the second operand of the *dot-operator* is a type rather than a field of an object. Notice that exceptions from logFile and dataFile are handled by bound handlers, while exceptions from tmpFile are still handled by an unbound handler. Finally, the exception raise has to be extended to transfer an object/exception pair in the event. Initially, it is assumed the object used at the raise is fixed during propagation, called *static binding* (see Section 5 for dynamic binding).

## 3 Static Bound Exceptions

Attempts are often made to simulate static-bound exceptions; however, we claim these simulations are either unsatisfactory or incomplete. A simple simulation is to embed each operation in its own try-block so each error condition can be handled individually. However, because the try-block is so *tight* around the method call, nonlocal error-handling is impossible, *i.e.*, error handling at outer scope levels. Additionally, since block positioning determines automatic storage allocation and execution control, it is often impossible to achieve an equivalent simulation. An intermediate approach is to only support bound exceptions among classes rather than the more general case of objects, while retaining the ability to have unbound handling by deriving class-specific exceptions from an unbound exception type. That is, matching is based on the type of object that raised the exception and the exception type. This approach provides discrimination among classes in bound matching, and may be accomplished with no language extensions. However, besides the restriction to only class-bound exceptions, there are several major disadvantages, such as the large number of required exception types and a restriction to languages supporting exception inheritance.

The most advanced approach to mimic bound exceptions is through the "catch and reraise" approach [1, §6.4]. The parameter mechanism is used to pass the "bound value" from the raise to the catch site, such as the object's id (most likely its address), and this association can be interpreted as a binding relationship. In C++, the latter can be done by introducing an attribute into the exception class (left example in Figure 2). After catching the exception, the passed value can be compared to the desired binding; if equal, the exception can be handled, otherwise it is reraised. This solution is now able to differentiate between exceptions raised by logFile and those raised by dataFile, which is a major advance over class-specific exception types. On the other hand, this approach

```
class BoundException {
  public:
    void * origin; // object's ID/address
    BoundException(void * p) : origin(p) {}
};
class FileErr : BoundException { ... };
class SpecialFileErr : FileErr { ... };
...
try { ... logFile.doWrite(); ...
      ... dataFile.doWrite(); ...
      ... tmpFile.doWrite(); ...
} catch( FileErr e ) {
    if (e.origin == &logFile) { ...
    } else if (e.origin == &dataFile) { ...
    } else if (e.origin == &tmpFile) { ...
    } else throw; // reraise
}
```

```
try { ... logFile.doWrite(); ...
      ... dataFile.doWrite(); ...
      ... tmpFile.doWrite(); ...
} catch( SpecialFileErr e ) {
    if ( e.origin == &tmpFile ) { ...
    } else throw; // reraise
} catch( FileErr e ) {
    if (e.origin == &logFile) { ...
    } else if (e.origin == &dataFile) { ...
    } else if (e.origin == &tmpFile) { ...
    else throw; // reraise
}
```

**Fig. 2.** Catch and Reraise / Reraise Anomaly

increases the program's complexity by adding additional data and code to the exception handling process. In particular, a programmer must follow the strict convention of inheriting from BoundException, and manually checking the binding information after catching the exception and reraising it if there is no handler for that binding. Following such a convention is always unreliable and error-prone. As well, there are situations in which the "catch and reraise" approach does not work. In the right example of Figure 2, a SpecialFileErr bound to tmpFile is to be handled, or a FileErr bound to logFile, dataFile, or tmpFile (note SpecialFileErr inherits from FileErr). If tmpFile raises a SpecialFileErr exception, the first catch matches and the handler is executed correctly. If one of logFile or dataFile raises a SpecialFileErr exception, the first catch also matches but the binding fails, and therefore, the exception is reraised. However, because a catch clause has already been matched for the guarded block, the reraise cannot perform further matching on the lexically following catch clauses of the same try-block. Thus, the "catch and reraise" strategy cannot reach the second catch clause, which would otherwise match and handle the exception. This behaviour does not match the usual semantics of exception handling and that necessary for bound exceptions, is counter-intuitive, and results in control flow that is difficult to predict.

By using a complex programming transformation, it is possible to eliminate the reraise anomaly [6]. The approach splits related catch-clauses into different (nested) try-blocks, but since the order of catch clauses is important, the catch clauses lexically following a related one must also go into the containing try-block. While try-block splitting can mimic bound exceptions with conventional exception handling, the conversion is complicated and can produce large amounts of additional code. As for the other simulations, it is unreasonable to rely on programmers to follow complex conventions to achieve a sophisticated programming concept. Therefore, if bound exceptions are a desirable feature, it is necessary to implement them as part of the language.

## 4 Bound Exception Design

There are multiple issues in designing bound exceptions: where and how bound exceptions are defined, and the object binding during exception propagation.

With respect to bound exception definition, there are multiple approaches in languages like C++. Probably the best mechanism is specifying the binding as part of the handler or the raise, giving four possible combinations of unbound/bound catching/raising:

| | **unbound raise** | **bound raise** |
|---|---|---|
| **unbound catch** | 1) unbound | 2) unbound |
| **bound catch** | 3) not handled | 4) bound |

Case 1) is the catch and raise are unbound, which is conventional exception handling with unbound handler matching. Case 2) is an unbound catch and a bound raise, so the handler is not object-specific. The unbound catch-clause can handle *all* exceptions of that type, both unbound and bound. Hence, the bound exception is handled by an unbound handler. Case 3) is a bound catch and an unbound raise, but the bound catch-clause cannot handle this exception because it is unbound. Case 4) is a bound catch and a bound raise (to the same object), so the catch clause is able to provide an object-specific handler and the exception is handled as a bound exception. It is possible to simplify the table (eliminate column 1), with only minor loss in functionality, by defining all raises to be bound, *i.e.*, always include the object raising the exception as part of the exception event. In this design, no functionality is lost between Case 1) and Case 2), as both perform an unbound catch. However, the functionality of Case 3) is eliminated as there is always a binding at the raise, which may be a null binding value (see below). Now the binding decision is made solely in the catch clause. The positive consequence of this design decision is that legacy code, which does not know about bound exceptions, continues to work after replacing all raises by bound ones (possibly by recompiling). A negative consequence is that all raises now require additional memory for the binding information and time to store the binding information, regardless of whether the binding information is used. However, the space/time overhead is not an issue because the space is small and exceptions normally occur infrequently.

The selection of the binding object seems obvious, *i.e.*, the object responsible for the raise, which precludes raising an exception bound to a *different* object (*e.g.*, **throw** logFile.FileErr); such a possibility would weaken an object's control. For special cases, such as non-member routines and static class-members, the binding value can be set to null. Hence, if a programmer does not want an exception to have a direct binding, the exception can be raised indirectly in a static-member routine.

Interestingly, extending the concept of bound exceptions to resumption propagation is straight-forward since there are no differences during propagation with respect to matching between termination and resumption exceptions. Nonlocal propagation among coroutines/tasks (*i.e.*, propagation across execution-stacks)

is possible by extending the object binding. The previous binding rule is inappropriate for nonlocal exceptions because the object raising the nonlocal exception in a coroutine/task may not be meaningful or even known in the target coroutine/task, and a nonlocal exception can essentially happen anytime, anywhere. A better solution for nonlocal exceptions is to bind to the raising coroutine/task, so it appears the exception emanates from it. For the case where a coroutine or task does not care about the specific tasks sending it nonlocal exceptions, *e.g.*, clients communicating with a server, it is possible to process the exceptions using unbound handlers.

## 5 Dynamic Bound Exceptions

The problem with static binding is that the object raising the exception may be a local variable or argument of a block. Therefore, once the exception propagates outside of the declaring block, the binding object may disappear or be invisible. This issue can be solved in many cases if the exception changes its binding during propagation at each object it traverses through, called *dynamic binding*. These cases are illustrated in the left example of Figure 3. Assume the declaration of db either passes a file for initializing variable DB::f or DB::f is created as a local variable. In either case, routine DB_Manager::flush does not know about this file object (especially when separately compiled). Now the catch clause inside commit is within the scope of f, so it can catch any exceptions raised by f. However, the attempted catches in flush are syntactically or semantically incorrect. The first catch is syntactically incorrect since f is invisible inside the scope of flush (unless coincidentally there is an f variable in the current scope, which would result in a difficult to locate error). The second catch is semantically incorrect since db does not raise the bound exception FileErr, so this catch is never matched. While these catch clauses are wrong, logically the user is trying to do the correct thing. That is, catch the specific FileErr exception associated with its operation (flush), but not catch FileErr exceptions associated with other operations, which might be handled at a higher level of abstraction. In fact, the catch clause **catch**(db.FileErr) is probably what a user really wants to write, as db exists in the current scope and (from a logical point of view) is responsible for raising the exception. This catch clause works for dynamic binding because the binding object changes from f to db when exception propagation terminates the call to commit. The right side of Figure 3 visualizes the binding change during stack unwinding when dynamic binding is used during propagation. In fact, we have identified cases where a user may need both static and dynmaic binding to correctly handle an exception, and with extra syntax it is possible to support both.

## 6 Conclusion

Bound exceptions truly incorporate exception handling into the object-oriented design process. The ability to associate exceptions with objects strengthens the relationship between an exception and the object responsible for its raise. This feature creates more powerful exception handling capabilities, contributing to

```
class DB {
   File [&]f; // parameter or local
   public: void commit() {
      try { f.doWrite(); // raises FileErr
      } catch( f.FileErr ) {...} // OK
   }
};
void DB_Manager::flush( DB &db ) {
   try { db.commit();
   } catch( f.FileErr ) {...}
     catch( db.FileErr ) {...}
}
DB db(...);
class Driver {
      public: static void run() {
            DB_Manager dbman;
            dbman.flush( db );
      }
};
```

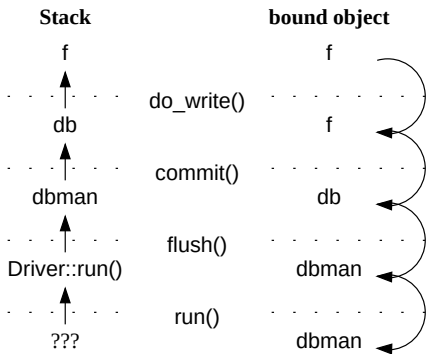



**Fig. 3.** Dynamic Binding

building more robust software. We believe this capability cannot be simulated in most OOPLs, and hence, requires a language construct. This work discusses two kinds of bound exceptions: static and dynamic. While some form of static binding has been discussed previously, this discussion extends static binding and presents dynamic binding as an interesting addition. As well, identifying that both kinds of bound exceptions can coexist and that each provides distinct capabilities to allow a user precise control in matching exceptions is an important idea.

## References


1. P. A. Buhr, A. Harji, and W. Y. R. Mok. *Advances in COMPUTERS*, volume 56, chapter Exception Handling, pages 245–303. Academic Press, 2002.
2. P. A. Buhr, H. I. Macdonald, and C. R. Zarnke. Synchronous and asynchronous handling of abnormal events in the $\mu$System. *Software—Practice and Experience*, 22(9):735–776, Sept. 1992.
3. Q. Cui and J. Gannon. Data-oriented exception handling. *IEEE Transactions on Software Engineering*, 18(5):393–401, May 1992.
4. C. Dony. A fully object-oriented exception handling system: Rationale and smalltalk implementation. In *Exception Handling*, volume 2022 of *LNCS*, pages 18–38. Springer-Verlag, 2001.
5. J. L. Knudsen. Fault tolerance and exception handling in BETA. In *Exception Handling*, volume 2022 of *LNCS*, pages 1–17. Springer-Verlag, 2001.
6. R. Krischer. Bound exceptions in object-oriented programming languages. Diplomarbeit, Universität Mannheim, Mannheim, Deutschland, Oct. 2002. ftp://-plg.uwaterloo.ca/pub/uSystem/KrischerThesis.ps.gz.
7. R. Miller and A. Tripathi. Issues with exception hnadling in object-oriented systems. In *ECOOP'97*, volume 1241 of *LNCS*, pages 85–103. Springer-Verlag, 1997.
8. K. M. Pitman. Condition handling in the lisp language family. In *Exception Handling*, volume 2022 of *LNCS*, pages 39–59. Springer-Verlag, 2001.

# Exceptions in Java and Eiffel: Two Extremes in Exception Design and Application

Joseph R. Kiniry

Computing Science Department, University of Nijmegen
Toernooiveld 1, 6525 ED Nijmegen
The Netherlands
kiniry@cs.kun.nl

## 1 Introduction

The designers of future languages will need to decide whether to include exceptions in their new languages. If they decide exceptions are warranted, they must then consider what exceptions represent: a structure for control flow, a structure for handling abnormal, unpredictable situations, or something in-between. Finally, the syntax and meaning of exceptions must be considered.

The syntax of exception mechanisms is important: syntax impacts how program code looks and is comprehended, it influences the design and realization of algorithms, and it affects the manner in which programmers handle unusual cases and unexpected conditions (what we'll call "errors"), thus indirectly impacts software reliability. While the syntax of exception mechanisms is the face most programmers see, their semantics is what tool developers and language theoreticians must wrestle with. In general, a small, elegant semantics is desired by all parties.

An excellent way to consider how to design a feature like exceptions in future languages is to analyze their design in today's languages. The analysis of exceptions in niche, historical, or research languages like Ada, PL/I, and CLU can reveal a gem or two, but perhaps more can be gained by examining the contrary viewpoints that exist in two modern languages.

The programming languages Java and Eiffel offer two opposing viewpoints in the design and use of exceptions. A detailed analysis of exceptions in these two languages: their language design, formal specification and validation, core library use, and non-technical "social" pressures, can help future language creators design their own exception mechanisms.

## 2 Language Design

Language design only partially influences the use of exceptions, and consequently, the manner in which one handles partial and total failures during system execution. The other major influence are examples of use, typically in core libraries and code examples in technical books, magazine articles, and online discussion forums.

This latter "social" effect is clearly seen in the use of exceptions in Java and Eiffel, as we will discuss in Section 5.

Exceptions in Java are designed to be used as control structures. This is also true of exceptions in most other modern programming languages including Ada, C++, Modula-3, ML and OCaml, Python, and Ruby.

Eiffel's exceptions, on the other hand, are designed to represent and handle abnormal, unpredictable, erroneous situations. The languages C#, Common Lisp, and Modula-2[1] use this general meaning for exceptions as well.

[1] Note that Modula-2 did not originally have exceptions; their addition caused a great deal of controvery through the early 1990s (i.e.,, compare [12] to [13]). See http://cs.ru.ac.za/homes/cspt/sc22wg13.htm for a historical discussion of such.

### 2.1 Java

Exceptions in Java are used to model erroneous circumstances. They always indicate situations that should not be witnessed during a typical execution of a program. Most Java exceptions are meant to be dealt with at runtime—just because an exception is thrown does *not* mean that the program must exit.

Java's exceptions are represented by classes which inherit from the abstract class `java.lang.Throwable`. They are generically called *throwables* because raising an exception is Java is accomplished with the `throw` keyword.

Java's throwables are one of two disjoint types: *unchecked exceptions* or *checked exceptions*. The former inherit from either the class `java.lang.RuntimeException` or the class `java.lang.Error`, the latter inherit from `java.lang.Exception` [4, Section 11.2].

**Java's Checked Exceptions** If a method can raise a checked exception, the checked exception type *must* be specified as part of the signature of a method. The `throws` keyword is used to designate such. A client of a method whose signature includes an exception $E$ (i.e., the method states "`throws E`") must either handle $E$ with a `catch` expression, or the client also must declare that it can throw $E$.

Checked exceptions are mainly used to characterize failure conditions for method invocations, like a file not being readable or a buffer overflowing. Not all erroneous conditions in Java are represented by exceptions though. Many method return special values, encoded as constant field of related classes, which indicate failure. This brings us to the first key point of this paper:

*Exceptions should have a uniform, consistent informal semantics for the developer.*

The state of Java with regard to point one is poor. While some attempt has obviously been made to use exceptions only for truly unexpected incidences, there are numerous examples of inconsistent use (e.g., `ArrayStoreException`, `FileNotFoundException`, and `NotSerializableException`).

Examples of common checked exceptions that are part of many method signatures include `IOException` and `InterruptedException`.

**Java's Unchecked Exceptions** Unchecked exceptions are either runtime exceptions or errors.

Runtime exceptions are conditions that can rarely (but potentially) be fixed at runtime, and thus are not errors. Examples include `ArrayIndexOutOfBoundsException`, `ClassCastException`, and `NullPointerException`.

Errors indicate serious problems with which most applications should not try to deal. Most errors indicate abnormal conditions with either the operating environment or the program structure. Examples of errors are `AssertionError`, `NoSuchMethodError`, `StackOverflowError`, and `OutOfMemoryError`.

### 2.2 Exceptions in Eiffel

The fundamental principle in Eiffel is that *a routine must either succeed or fail*: either it fulfills its contract or it does not. It the latter case an exception is *always* raised [9, 8]. Thus, exceptions are, by design, to be used in Eiffel exclusively to signal when a contract is broken.

Exceptions are not specified as part of the type signature of a routine, nor are they mentioned in routine contracts. In fact, there is no way to determine if a routine can raise an exception other than through an inspection of the routine's code, and all the code on which it depends.

Eiffel exceptions are represented by *INTEGER* and *STRING* values; there are no exception classes[2]. Exceptions that are part of the language definition are represented by *INTEGER* values, developer-defined exceptions by *STRING* values[3]. This limited and non-uniform representation of exceptions brings us to the second key point:

[2] Eiffel class names are always capitalized.

[3] Earlier versions of the Eiffel language standard permitted developer-defined integer exception values, but this seems to no longer be the case. It is unclear when and why this change was made.

*Exceptions should have a uniform representation,*
*and that representation should be amendable to refinement.*

Eiffel's exceptions have two representations, causing some design impedance when dealing with them. Additionally, because they are basic values and not objects, they have no internal semantics beyond that which can be expressed in a helper routine, which necessarily cannot be foolproof because of the representation overloading in effect.

**Contract Failure** Contracts can be violated in several ways, all of which are considered *faults*, but only some of which are under programmer control.

Operating environment problems, such as running out of memory, are one situation in which exceptions are signaled. In these cases a contract can fail, but not necessarily because the caller or the callee did something wrong.

Certainly, intentionally allocating too much memory, or otherwise using an extraordinary amount of system resources, is the fault of the program. But such situations are more malicious than typical.

Software infrastructure failures can cause exceptions. Some operating system signals raise an exception. Failures in non-Eiffel libraries that are used by an Eiffel application can cause exceptions as well. For example, Eiffel programs that link with Microsoft Windows COM components can witness an exception specific to COM routine failure. Eiffel programs that use UNIX libraries can see an exception which indicates that an external library failed but did not set the `errno` system variable. A floating point exception is raised on some architectures when a division by zero is attempted.

But most exceptions used in Eiffel are not due to external factors, but instead are *assertion violations* or *developer exceptions*, both of which are used to indicate program errors.

If assertion checking is enabled during compilation, assertion violations cause an exception to be raised. These exceptions are classified according to the type of assertion that has been violated.

The `check` instruction, which is equivalent to C's or Java's `assert` construct, cause a `Check_instruction` exception to be raised. A `Loop_variant` exception is another assertion violation; it is raised when a loop variant does not monotonically decrease during loop execution.

Violating a contract, either by failing to fulfill a class invariant, a method precondition or postcondition, or a loop invariant, is the final kind of exception. Contract violations fall into two categories: those that are the fault of the client of a class, and those that are the fault of the supplier of a class. The classification of an exception is determined by the context of the failure during program execution.

If a contract is broken at the time a method is called, regardless of whether the caller is another object or the current object (in the case of a callback, or the use of the `retry` keyword, see below), then the fault lies with the caller.

Exactly one kind of exception, called `Void_call_target`, can be the fault of either the caller or the callee. If a method is invoked on an object reference with value `Void`, a `Void_call_target` is raised. If the caller set the value to `Void`, or did not check the reference prior to making the invocation attempt, then the fault lies with the caller. In situations where the reference was obtained via a routine call, either via a formal parameter or a return value, and the value is `Void`, the fault lies with the callee, as the specification of the routine is not strong enough to eliminate the possibility of the `Void` value.

The uniform design for assertion violation signaling with exception in Eiffel is contrary to that which exists in Java. Several tools exist to permit the formal specification of contract for Java code. We use the excellent JML tool suite [7]. Unfortunately, because assertion violation semantics is so primitive in Java, there is no uniformity of exceptions across different assertion tools and specification languages. This brings us to point three:

*If exceptions are used to represent assertion failure, their design and semantics*
*should be incorporated into the core language specification.*

The users of these tools have suffered tremendously because the creators of Java ignored this key point in language design.

### 2.3 Comparing Eiffel's Exceptions to Java's Unchecked Exceptions

Eiffel's exceptions and Java's unchecked exceptions are exclusively focused on unexpected, erroneous behavior that an application should not try to handle. Thus, one would expect every Eiffel exception to map to a single Java unchecked exception. This is not the case.

Some of Eiffel's built-in exception types are equivalent to standard *checked* exceptions in Java. For example, Eiffel's `Io_exception`, `Runtime_io_exception`, and `Retrieve_exception` are semi-equivalent to `IOException` and some of its children.

A number of *unchecked* exceptions are equivalent to standard Eiffel exceptions. For example, `Void_call_target` is equivalent to `NullPointerException`, and `Floating_point_exception` is equivalent to `ArithmeticException`.

Finally, some children of `java.lang.Error` are equivalent to the remaining Eiffel exceptions: `AssertionError` is equivalent to the set of specification-centric Eiffel exceptions (`Check_instruction`, `Class_invariant`, `Loop_invariant`, `Loop_variant`, `Postcondition`, and `Precondition`), and `No_more_memory` is equivalent to `OutOfMemoryError` and `StackOverflowError`.

**Missing Mappings** Several exceptions that exist in each language have no peer in the other language.

`Rescue_exception` has no mapping, as Java does not perform any special handling of exceptions thrown in a `finally` clause.

An equivalent for `Signal_exception` is not part of the core Java language as Java's definition focuses on multiplatform development and not all platforms have signals[4]. The Eiffel language specification states that such system-specific exceptions should be contained in system-specific classes, but no compilers implement this suggestion.

An error like `Void_assigned_to_expanded` is not possible in Java as Java has no expanded types and the type system prohibits assignment of `void` to built-in types like `int` and `boolean`.

The Eiffel literature claims that Eiffel has no casting (cf., [10, page 194], thus there is no equivalent to Java's `ClassCastException`. This claim is a bit disingenuous because Eiffel's assignment attempt operator '`?=`' is simply a built-in conditional downcast in the form of an operator[5].

`Routine_failure` is a generic exception that indicates a routine has failed for some reason. The reason is sometimes recorded (as a `STRING`) in the `meaning` associated with the exception, but this is not mandatory. This is also true of Java exceptions, each of which has an optional message associated with it obtainable via `Throwable`'s `getMessage` method. Unfortunately, there is absolutely no uniformity to the use of these representations in either language.

*When defining a new type of exception, human and machine comprehensible representations (e.g., a string value and a predicate) should either be mandatory, or not exist at all.*

None of the various Java exceptions dealing with out-of-bounds access to arrays and strings exist in Eiffel because the contracts of accessor routines for these types prohibit such. Cloning-related exceptions do not exist because all objects can be cloned in Eiffel.

*Integrated contracts significantly decrease the number and complexity of exceptions.*

This point is emphasized by the quantative analysis of Section 4.

Standard Eiffel also does not have several features of Java: reflection, introspection, concurrency, and sandboxing. These features contribute significantly to the complexity of Java's exception class hierarchy.

[4] One can catch and handle signals in Java, but internal classes like `sun.misc.Signal` and `sun.misc.SignalHandler`, or a package like [6], are needed.

[5] This is not the only pragmatic circumvention in Eiffel. Other examples include the dual semantics of routine calls (with and without an explicit "`Current`") and the semantics of the `equal` and `clone` routines of ANY.

**Controlling Exceptions in Eiffel** Exceptions are primarily controlled in Eiffel using *rescue clauses* and the `retry` instruction. Exceptions are also indirectly controlled by the choice made in *compilation mode* during application development.

A routine may end with a rescue clause. A *rescue clause* of a routine is a block of code that will execute if any exception is raised during the execution of the routine.

The rescue clause does not discriminate between different types of exceptions. In this respect, it is functionally equivalent to the surrounding every Java method body with a `try`/`catch` block where the catch expression is "`catch (java.lang.Throwable)`". The rescue clause is *not* equivalent to Java's `finally` construct. The code enclosed in a finally block is *always* executed when a method completes, whether it completes normally or abnormally, while a rescue clause only executes when a routine fails.

The `retry` instruction causes a routine to restart its execution, attempting again to fulfill its contract. This instruction can only be used within a rescue clause. If a rescue clause does not contain a retry instruction, then the routine fails and the current exception is raised in the immediate caller.

We will return to the details of `finally` and `rescue` in the sequel.

Exceptions are manipulated in Eiffel using the EXCEPTIONS class. Using this class one can find out information about the latest raised exception (much like `errno` in C), handle certain kinds of exceptions in a special way, raise special developer-defined exceptions, and prescribe that certain exceptions must be ignored at run-time.

The EXCEPTIONS class is part of the Eiffel Kernel Library, thus is available in all Eiffel compilers.

## 3 Exceptional Specifications and Validation

The key difference between the use of exceptions in the two languages in that exceptions are *part* of a method contract in Java and are *not* part of a routine contract in Eiffel. Thus, a fundamental notion of "Design by Contract", that of exceptions exclusively indicating contract failure, has a different interpretation in Java.

### 3.1 Contracts with Exceptions in Java

We use the Java Modeling Language to write formal specifications of Java components [1]. We have participated in the development of a coalgebraic semantics for Java and JML [5]. The discussion in this section are based upon that experience.

The semantics of Java, and thus JML, are significantly complicated by the possibility of abrupt method termination (i.e., an exception being thrown). Validation proofs must deal with three cases in Java: normal termination, abrupt termination, and divergent behavior, sometimes tripling proof size.

The default specification for a failure is simply "true", which means that the routine guarantees nothing in particular when a failure takes place. Rarely can nothing stronger be said, and in fact exceptional cases are often the first part of a formal specification we write.

This information helps the caller deal with the exceptional cases in a more reasonable manner than just halting. We have also found that the specification of a postcondition for abrupt termination is *mandatory* for reasoning about systems during abrupt termination. Without such assertions, class invariants would become significantly more complex because ghost variables would be needed to represent failure states for all of the routines of a class.

### 3.2 Specifications of Eiffel Exceptions

In Eiffel, the semantics of *exceptional-correct* routines is rolled into the definition of *class correctness* [10, Chapter 15 and Section 9.16].

The definition [10, Section 15.10] of *exception-correct* is:

A routine $r$ of a class $C$ is exception-correct if and only if, for every branch $b$ of its rescue block:

1. If $b$ ends with a `Retry`: `{true} b {INV_C and pre_r}`
2. If $b$ does not end in a `Retry`: `{true} b {INV_C}`

where `INV_C` is the class invariant of $C$; `pre_r` is the precondition of routine $r$.

This existing semantics is a problem in practice because it means that Eiffel code must always have a rescue block that "puts everything right" (fulfills the normal postcondition), which usually means either a significant weakening of the postcondition (one can barely state anything is true if things can either fail or succeed) or an extremely complex postcondition with a large set of disjuncts with error-flag guarded expressions.

For example,

```
   method_call_failed implies (F || G || H)
|| not method_call_failed implies (I || J || K)
```

This kind of specification is evident in the very few places where exceptions are handled in Eiffel code, and we speculate this is true because of the inherent complexity in such specifications.

Specifications in JML that use keywords like `exsures` and `exceptional_behavior` which are simply shorthand for these more complex expressions. We believe that Eiffel could benefit from such expressions as well.

This semantics significantly complicates contracts and weakens their application. Neither case is surprising: either (in case 1) a rescue clause must fulfill the invariant and the precondition of the retried routine or, (in case 2) a retry does not happen so the routine has to leave the object in a legitimate state by fulfilling its invariant. What is surprising is that *nothing* is know about when or why the exception happened in the first place, since both preconditions are as weak as possible, and nothing *new* can be specified about the state of the objects when a failure takes place, since the postcondition is exactly the invariant.

JML is exactly contrary here—it provides the ability to state a stronger postcondition in these exceptional cases, and this information is essential to validating programs with exceptions.

This brings us to our next key point:

*The specification of object state when an assertion is raised, either via an exceptional postcondition or an exception predicate, is very desirable if programs are to be formally verified.*

The Java Modeling Language fulfills this key point admirably, while Eiffel fails in this regard.

## 4 Qualitative and Quantitative Comparisons

In the end, it is unclear how important exceptions are in the Eiffel world. This might be due to exception's perceived second-class nature in the Eiffel universe of "correct" software, as evidenced by their rare use (see below).

If exceptions in Eiffel are equivalent to unchecked exceptions in Java, and if library programmers for the two languages equally careful and capable of handling unexpected circumstances, then an analysis of exception usage in the two core code bases should yield comparable results.

The data in Table 1 is the result of such an analysis. In the case of the Gobo and SmartEiffel systems, all code, library and applications, was analyzed for this data. The number of declared exceptions is determined by counting and classifying all calls to *EXCEPTIONS.raise* and *EXCEPTIONS.die*, in the case of Eiffel, and counting all descendants of `java.lang.Throwable`, in the case of Java. The number of raised exceptions is determined by a count of the number of calls to *EXCEPTIONS.raise* and *EXCEPTIONS.die*, in the case of Eiffel, and the number of `throw` expressions, in the case of Java. The data on stack traces is determined by counting and analyzing all calls to routines `exception_name`, `tag_name`, `meaning`, and `developer_exception_name` of class `EXCEPTIONS`. All numbers are approximate and only measured using the `wc` command.

Consider that in Java an unchecked exception is thrown for approximately every 140 lines of code, where in Eiffel one is used for every 4,600 lines of code; that is a difference of over thirty times in frequency. The above statistics clearly show that either or both (a) exceptions in Eiffel, either through technical issues or social pressure, have a second-class (or perhaps even ignored) status, or (b) the built-in existence of adequate specification technologies inherently leads to fewer assertions being thrown. Given the preponderance of quality Eiffel software available, the latter point holds much more weight. This is especially highlighted in the complete lack of exception usage in the GNU SmartEiffel system.

| Library | Gobo 3.1 | ePosix 1.0.0 | ISE Eiffel 5.3 | SmartEiffel 1.0 | JDK 1.4.1 (java.) |
|---|---|---|---|---|---|
| Number of direct/indirect mentions of EXCEPTIONS, or unchecked exceptions | 18 | 3 | 17 | 0 | 525/ 15,000 |
| Number of unchecked/checked exceptions declared | 3/- | 6/- | 5/- | 0/- | 50/ 150 |
| Number of raised unchecked/ checked exceptions | 66/- | 87/- | 13/- | 0/- | 3,000/ 2,650 |
| Number of `rescue` or `finally` clauses | 6 | 10 | 29 | 0 | 50 |
| Number of `retry` commands | 81 | 3 | 15 | 0 | N/A |
| Number of times a stack trace is (a) checked or manipulated, or (b) printed or ignored | 0/0 | 0/0 | 0/0 | 0/0 | 8/79 |
| Total lines of code and documentation | 250,000 | 25,000 | 372,000 | 115,000 | 421,000 |

**Table 1.** Use of Exceptions in Eiffel and Java

This data should be carefully considered by the committee performing ECMA language standardization of Eiffel. It also provides evidence for potential avenues for language refinement, particularly with regards to the specification of exceptional conditions.

## 5 Exception Equivalency

Both languages have exceptions mechanisms that can be treated as equivalent. A hierarchy encoding can be represented by integer or string values, so we could build an artificial type hierarchy for Eiffel exceptions if we felt it necessary.

Likewise, the minimal exception interface of Eiffel, embodied in the *EXCEPTIONS* class, could be modeled in Java. In fact, some Java developers advocate avoiding checked exceptions entirely, instead inheriting exclusively from `RuntimeException` [3].

We can find no evidence of the converse, that of Eiffel programmers using exceptions as flow control mechanisms.

As any Java programmer knows, the volume of `try`/`catch` code in a typical Java application is far larger than the comparable code necessary for explicit formal parameter and return value checking in other languages that do not have checked exceptions.

In fact, the general consensus among in-the-trenches Java programmers is that dealing with checked exceptions is nearly as unpleasant a task as writing documentation. Thus, many programmers report that they "resent" checked exceptions. This leads to an abundance of checked-but-ignored exceptions, as evidenced by the next to the last line of the table of the previous section.

Additionally, the presence of checked exceptions percolates through the system. As discussed by the designers of C# [2],

> Examination of small programs leads to the conclusion that requiring exception specifications could both enhance developer productivity and enhance code quality, but experience with large software projects suggests a different result – decreased productivity and little or no increase in code quality.

This attitude guides the design of error handling in the .NET framework as well [11, see Section "Error Raising and Handling Guidelines"].

These issues lead us to our last, and perhaps crucial point:

*Checked exceptions generally increase system fragility (because of signature refactoring), increase code size (due to explicit, localized, mandatory handling), and cause programmer angst (as evidenced by the number of empty or spiteful* `catch` *blocks in public Java code), so their inclusion in a language should be carefully considered.*

In the end, so long as an exception mechanism has a simple semantics, is consistently used, and provides a tool which programmers can understand, depend upon, and not resent, then they should be included in future languages.

*Acknowledgments.* This work was supported by the Netherlands Organization for Scientific Research (NWO). Thanks to the anonymous reviewers and Alexander Kogtenkov for their comments.

## References

1. Lilian Burdy, Yoonsik Cheon, David Cok, Michael Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. Technical Report NIII-R0309, Dept. of Computer Science, University of Nijmegen, 2003.
2. Posted by Eric Gunnerson. Original author unknown. Why doesn't C# require exception specifications?
3. Bruce Eckel. Does Java need checked exceptions? See http://www.mindview.net/Etc/Discussions/CheckedExceptions, particularly the ensuing feedback on this issue.
4. James Gosling, Bill Joy, and Guy Steele. *The Java Language Specification*. Addison-Wesley Publishing Company, first edition, August 1996.
5. Bart Jacobs and Erik Poll. A logic for the Java modeling language JML. Technical Report CSI-R0018, Computing Science Institute, University of Nijmegen, November 2000.
6. What is JavaSignals?, 1999. See http://www.geeksville.com/ kevinh/projects/javasignals/.
7. Gary T. Leavens, Albert L. Baker, and Clyde Ruby. *Behavioral Specifications of Business and Systems*, chapter JML: A Notation for Detailed Design, pages 175–188. Kluwer Academic Publishing, 1999.
8. Bertrand Meyer. Disciplined exceptions. Technical Report TR-EI-13/EX, Interactive Software Engineering, 1988.
9. Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., second edition, 1988.
10. Bertrand Meyer. *Eiffel: The Language*. Prentice-Hall, Inc., 1992.
11. .NET framework general reference, 2003. Documentation version 1.1.0.
12. Richard J. Sutcliffe, editor. *Modula-2 (Base Language)*. Number 10514-1:1996 in ISO/IEC Modula-2 Standardization. ISO/IEC, 1999.
13. Niklaus Wirth. *Programming in Modula-2*. Springer–Verlag, 1982.

# Non-Functional Exceptions for Distributed and Mobile Objects

Denis Caromel and Alexandre Genoud

INRIA Sophia Antipolis, CNRS - I3S - UNSA
BP 93, 06902 Sophia Antipolis Cedex - France
First.Last@inria.fr

**Abstract.** While there is quite a lot of techniques to separate non functional properties from functional code, the handling of induced exceptions remains often blurred within application. This paper identifies *Non-Functional Exceptions* as exceptions related to various failures of non-functional properties (distribution, transaction or security). We propose a hierarchical framework where reified exception handlers are attached to various entities (proxies, remote objects, futures). Such handlers allow middleware and application oriented handling strategies for distributed and mobile computation. The mechanism tries to handle exceptions at non-functional level as much as possible.

## 1 Introduction

Distributed environments provide synchronous and asynchronous calls, remote references, migration of activities. Complex communications are subject to various failures such as the remote communication failure. It is always unclear whether the failure occurred in the communications medium or in the remote process, and the state of the system is in general uncertain. Unfortunately, the `try/catch` construction is heavy to use, and only convenient for simple communication errors.

In this article, we define *non-functional exceptions* as exceptions related to distribution. We present a hierarchical model based upon *handlers of exception*. Sets of handlers are dynamically attached to various entities (JVMs, remote and mobile objects, proxies, ...) in order to provide a generic and flexible recovery mechanism at a non functional level. This model has been implemented and bench marked in a framework for parallel, distributed and mobile computing known as ProActive[1]. As implementation remains simple, the port to other middlewares is possible.

The first section presents previous works related to exception handling in distributed architectures. Then, non-functional exceptions are defined and those related to distribution are classified. The next chapter describe a flexible model used to handle simple communication failures but also to create advanced fault-tolerance strategies. Finally, pragmatic examples are presented. Performances are discussed in the appendix.

[1] http://www.inria.fr/oasis/ProActive

## 2 Related Work

Exceptions have been created in ADA in the 1970s and are now a standard mechanism to report errors and failures. In distributed environments, they are raised from host to host and thus are difficult to handle. Through the development of our distributed library, we realized that standard handling mechanisms are not appropriate to distribution, as developers must define handling code for every distributed exception.

Authors of [4] highlight a critical problem that appears when several failures occur simultaneously. While communications between distant processes are broken, an unstable state is probably reached. This article suggests to gather communicating processes into a *conversation* before starting any kind of communication. Participants first save their own state ; then a set of handlers is associated to the conversation. All action participants are involved in co-operative handling of any exception raised by any action participant : the conversation is paused until the handling process is terminated. When handlers are not sufficient to recover from failure, the conversation is canceled. Every process checks possible side effects and rollbacks to its initial state. This collaborative strategy seems really promising but fails with asynchronism. As the return time of an asynchronous call is unknown, the lifespan of the conversation is also unknown. The recovery process could be maintained as long as no result is delivered.

Agents are active objects having autonomous behavior according to their environment. As mobility is one possible behavior, an agent can decide to migrate on a different virtual machine. In this context, authors define *guardians* in [5] as centralized mechanisms helping agents to handle exceptions related to distribution. Only one guardian is needed for every agents-based application. When an agent cannot handle an error, the exception is raised to the guardian which send back instructions. Of course, the handling behavior depends not only of the nature of the exception but also of the agent environment. When distant objects become unreachable, the guardian can advise to delay communication. When critical failures occur, the guardian can terminate agents. An interesting strategy to handle failures related to the migration of agents could be to find an equivalent destination using the replication strategy. This centralized model offers simplicity as it provides only one single guardian even for large distributed systems. However, many problems would occur if the guardian becomes unreachable or crashes.

## 3 Non-Functional Exceptions

During the conception process, we identified three majors features required for distributed handling mechanisms : flexibility, genericity and dynamicity. Considering that previous models did not meet all of these requirements, we decided to create an original model from scratch based upon a new classification of exceptions.

### 3.1 Functional versus Non-Functional

In recent literature, classifications of exceptions are proposed. According to [7], exceptions can be divided into *internal exceptions*, raised from and handled within a method, and *external exceptions* propagated toward other methods. This classification is not useful for distributed environments which require complete description of internal failures. In our framework, we consider the mechanism of distribution as a *non-functional property* [8]. We use this specificity to define non-functional exceptions as exceptions raised from any non-functional property.

**Definition 1** *Non-functional Exceptions announce failures occurring in non functional properties. They are raised in non-functional code and handled, as much as possible, within it.*

We make a clear difference between functional exceptions, related to abnormal behavior of applications, and non-functional exceptions, related to failures of non-functional properties. Exceptions related to distribution should be considered as non-functional exceptions coming from the middleware. We agree with the recommendation of [9] which claims that exceptions have to be handled at meta level. It is much more simple indeed to handle exceptions directly in internal mechanisms of distribution.

### 3.2 Location of Non-Functional Exceptions

Distributed environments provide synchronous and asynchronous communications as describe in [1]. Failures in such communications result in non-functional exceptions as shown in 1. While in synchronous calls, those exceptions are simply handled at results delivery, asynchronous calls lead to two solutions. Exceptions are eventually handled when requests containing reified calls are synchronously queued. But non-functional exceptions have to be handled in future objects when pending requests are served or when results are stored within them.

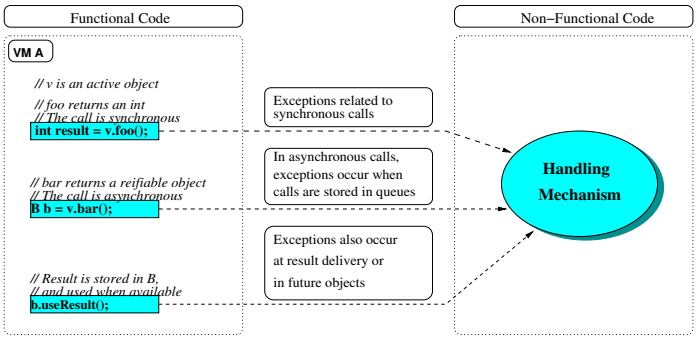



**Fig. 1.** Exceptions Raised from Synchronous and Asynchronous Calls

### 3.3 A Hierarchy of Distributed Exceptions

We first identified and classified potential failures (figure 2) of distributed environments. Then, we built a *hierarchy of potential failures*, opened to developers who can add new failures and topics. We kept this structure customizable as flexibility is the most important feature of recovery mechanism. Finally, we associated non-functional exception to every failure. This hierarchy is used to define handling strategies for specific exceptions as well as for groups of exceptions.

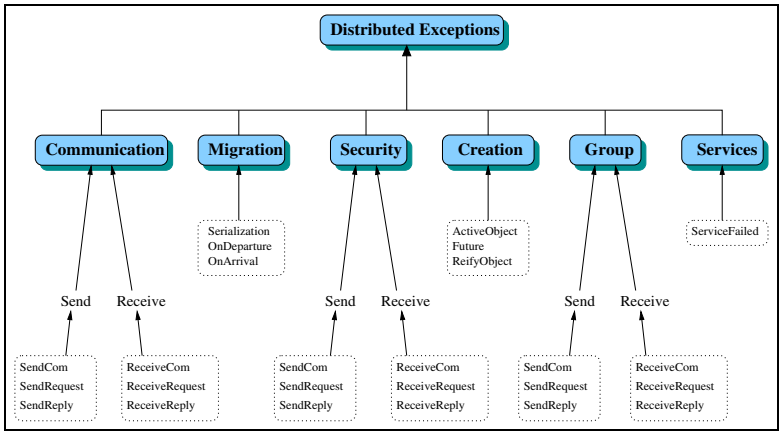



**Fig. 2.** Hierarchy of Failures Encountered in Distributed Environments

## 4 A Hierarchical and Dynamic Handling Mechanism

The hierarchy of failures described above is used in the construction of hierarchical handlers, working indifferently at functional or non-functional levels.

**Definition 2** *Handler of exceptions handle non-functional exceptions as well as groups of such exceptions, thanks to object inheritance.*

For instance, a handler can be associated to `SendRequestGroupException` or to every member of `GroupException` (see [10] for detail about group communications). Handlers provide basic strategies in non-functional code, but application-specific strategies are also possible. They reify the `try/catch` construction to support both genericity and flexibility required by any handling mechanism. Handlers implement a common interface and provide functional as well as non-functional treatments of non-functional exceptions.

### 4.1 Prioritized Levels of Handling

Our mechanism is based upon a default and static level, created during the middleware initialization, and some dynamic levels set during execution. Each

structure can provide a specific fault tolerance strategy created from an appropriate set of handlers. Every non-functional exception is associated to one handler in the default level. The default strategy is basic but always present while complexes strategies appear occasionally in higher levels. We defined six different levels, associated to constants within the implementation and presented below from lower to higher priority.

1. *Default level* is static and initialized in core of applications. This level provide a basic handling strategy for every non-functional exception.
2. *Virtual Machine level* is the first level that can be created dynamically. It offers the possibility to define a general handling behavior for every VM.
3. *Remote and Mobile Object level* is used to bind handlers to remote objects. Handlers associated to mobile entities migrate along with them.
4. *Proxy level* is used to define strategies for references to active objects. When reference are passed to other VMs, handlers are passed also.
5. *Future level* is attached to the results of asynchronous calls.
6. *Code level* allows temporary handlers to be set in the code.

As describe above, the default level provides a basic handling strategy, defined during the initialization of middleware. Virtual machine level and higher ones are set dynamically to improve this strategy. Dynamic handlers are created at runtime and added to an appropriate level (VM, remote object, proxy, future or code levels).

### 4.2 Presentation of the API

The API is both used for middleware adaptation (e.g. wireless oriented) and for distributed application. It consists in two major static functions which offer settings and configurations of handlers into appropriate levels. The five dynamic levels are defined with constants.

```
    // Binds one handler to a class of exception at the specified level.
void setExceptionHandler(level, Handler, Exception, Target);

    // Removes handler associated to a class of exception at specified
    // level. Target is different from null when level is object-related.
Handler unsetExceptionHandler(Level, Exception, Target);
```

The following example show how to protect an application from communication failures. We add a handler with the *setExceptionHandler* primitive. Communication failures are thus handled for that object.

```
    // Creation of a remote and mobile object with handlers
RO ro = (RO) ProActive.newActive("RO", "//io.inria.fr/VM1");

    // A communication handler is dynamically associated
    // to the remote object trough its proxy.
setExceptionHandler(ProxyLevel,
                    "CommunicationHandler",
                    "CommunicationException",
                    ro);
```

### 4.3 Dealing with Mobility

Most of the distributed environments offer remote and mobile objects. Such objects can migrate from host to host. This additional constraint can break the continuation of the handling mechanism. The migration process must be modified to take into account the migration of mobile object handlers. As explained later, mobile objects and their associated levels remain always gathered. Handling mechanism can be associated to proxy also in order to attach a specific strategy to remote references.

### 4.4 Implementation

As explained before, the handling strategy is built upon one static level improved occasionally with dynamic levels. Handlers are searched with the following dedicated function.

```
    // Searches through prioritized levels the handler associated
    // to the given class of exception
Handler searchExceptionHandler(Exception, Target);
```

The following code is part of the middleware and describe how to activate the handling mechanism. Instead of providing a treatment directly in the *try/catch* block, we use the *searchExceptionHandler* primitive.

```
try {
        // Send the reified method call
    sendRequest(methodCall, null);

} catch (NonFunctionalException e) {

        // Looks for an appropriate handler and
        // use the handler if possible
    Handler handler = searchExceptionHandler(e);
    if (handler) handler.handle(e);
}
```

We tried to keep implementation as simple as possible but performance issues were also considered. Levels are implemented with hashmap to provide fast access to handlers. Considering the memory available in modern computer, we support time complexity instead of space complexity even if migration increase memory requirements because of levels associated to mobile objects. The cost is proportional to the number of handlers contained in the object level.

Reflexion is used to search handlers for a specific class of exception or for the mother class of a group of exceptions. The algorithm supports generic handlers of higher levels instead of specific handlers from lower level ; Levels have precedence over the type of exceptions. For instance, on figure 3, the most suitable handler for exceptions related to class 02 is found in the highest level. When no handler is available at remote object level, the search continue in VM an lower level. This choice, which seems more natural, can be invert.

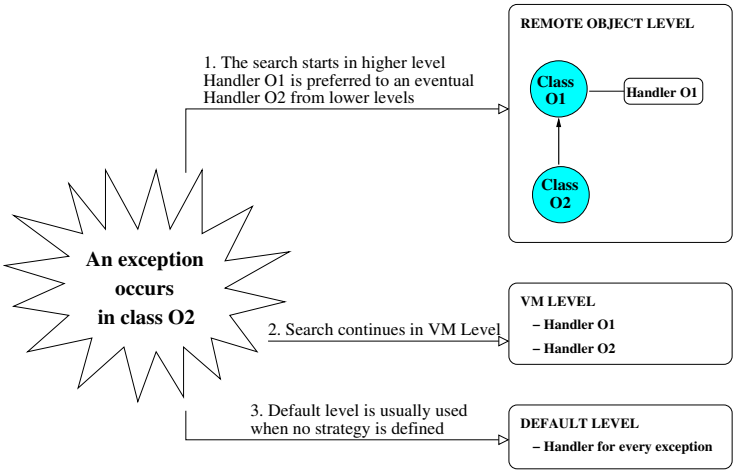



**Fig. 3.** Levels have Precedence over Exception Type when Searching Handlers

## 5 Canonical Examples

We present in this section two applications which use our handling mechanism.

### 5.1 Handling Exceptions in Unconnected Mode (e.g. wireless PDA)

Distributed applications for *Personal Digital Assistants* should provide an unconnected mode to handle at least communication exceptions due to broken connections. We defined a strategy where handlers store requests sent to unreachable PDAs in a queue. Time by time, a thread checks if the connection is restored in order to deliver requests. The point is not to define a sophisticated strategy, but to show how easily it can be activated. Here is the scheme of such a *PDA-Handler*.

```
Class PDACommunicationHandler implements Handler {
    public boolean isHandling(Exception e) {
        return (e instanceOf CommunicationException);
    }
    public void handle(Exception e) {

            // A thread testing connectivity is created
        if (firstUse) {
            connectivityThread = new ConnectivityThread();
        }
            // Then reified method calls are stored in the
            // queue and exceptions are not propagated anymore
        queue.store(e.getReifiedMethodCall());
    }
}
```

Imagine now that an entity is about to create a mobile object that migrate on some wireless PDA.

```
    // Creation of a remote and mobile object with handlers
RO ro = (RO) ProActive.newActive("RO", "//io.inria.fr/VM1");

    // A communication handler is dynamically associated
setExceptionHandler(ProxyLevel,
                    "PDACommunicationHandler",
                    "CommunicationException",
                    ro);

    // The mobile object can now migrate safely
ro.migrateTo("//pagode.inria.fr/VM2");
```

### 5.2 Simulating a Centralized Error Manager

The handling mechanism can easily be configured into a centralized error manager similar to the one presented in [5]. We create first a remote object containing a complete set of prioritized handlers. This object is located on one virtual machine but is known from every active object of the application. Non-functional exceptions reporting failure are not handled directly in the active object but are raised to the centralized error manager instead. A handler corresponding to the failure is sent back to handle the exception. This strategy does not avoid the typical problems common to every centralized error manager but offers at least an efficient centralized handling mechanism, easy to configure.

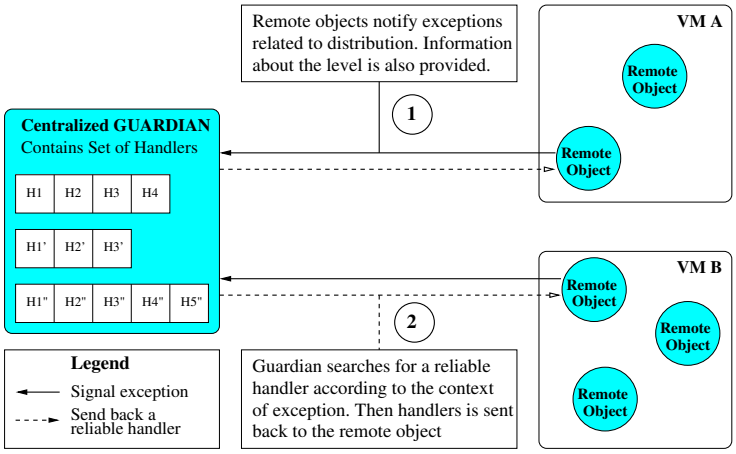



**Fig. 4.** Centralized Error Managers are Easy to Implement

## 6 Conclusion and Perspectives

We defined a dynamic, flexible and generic model to handle non-functional exceptions. We proposed a classification for non-functional exceptions along with a hierarchy of prioritized levels. As implementation use the classical `try/catch` language construct, the model is reliable for a large panel of modern, object-oriented, programming languages.

## References

1. D. Caromel, W. Klauser and J. Vayssiere. *Toward Seamless Computing and Metacomputing in Java.* Concurrency Practice and Experience (September-November 1998) p. 1043-1061 Editor Geoffrey C. Fox, published by Wiley & Sons
2. E. F. Walker, R. Floyd, P. Neves *Asynchronous Remote Operation Execution in Distributed Systems.* In Proc. of the Tenth International Conference on Distributed Computing Systems, May/June 1990.
3. F. Baude, D. Caromel, F. Huet and J. Vayssiere, *Communicating Mobile Active Objects in Java* HPCN Europe 2000, Amsterdam - The Netherlands, May 2000
4. Jie Xu, Alexander B. Romanovsky and Brian Randell. *Coordinated Exception Handling in Distributed Object Oriented System (Revision and Correction).* Department of Computing Science, University of Newcastle upon Tyne, Newcastle upon Tyne, UK.
5. Arnand Tripathi and Robert Miller . *Exception Handling in Agent-Oriented Systems.* Advances in Exception Handling Techniques, Springer-Verlag LNCS 2022, March 2001.
6. Valerie Issarny. *Concurrent Exception Handling.* Advances in Exception Handling Techniques 2000: 111-127. Inria Rocquencourt.
7. Alessandro F. Garcia, Cecilia M. F. Rubira, Alexander Romanovsky and Jie Xu. *A Comparative Study of Exception Handling Mechanisms for Building Dependable Object-Oriented Software.* Journal of Systems and Software, Elsevier, Vol. 59, Issue 2, November 2001, p. 197-222.
8. Kiczales, Lamping, Mendhekara, Maeda, Lopes, Loingtier, Irwin. *Aspect-Oriented Programming.* Proceedings of ECOOP 97, n 1241 LNCS, Springer-Verlag, June 1997, p. 220-242.
9. Ian S. Welch, Robert J. Stroud and Alexander Romanovsky. *Aspects of Exceptions at the Meta-Level (Position Paper).* Department of Computing, University of Newcastle upon Tyne.
10. Laurent Baduel, Françoise Baude, Denis Caromel. *Efficient, Flexible, and Typed Group Communications in Java.* Proceedings of the Joint ACM Java Grande - ISCOPE 2002 Conference. Nov. 2002.
11. Anh Nguyen-Tuong. *Integrating Fault-Tolerance Techniques in Grid Applications.* Partial Fullfillment of the Requirements for the Degree Doctor of Computer Science. University of Virginia.

## Appendix: Time and Space Performances

**Space Complexity :** Each system contains at least default and virtual machine levels : some handlers contained by two hashtables.

| Strategy | Description | Number of Handlers | Size in Byte |
|---|---|---|---|
| No Handler | No handler is provided. We just pay the cost of an empty level based upon Hashtable | 0 | 82 |
| Minimal | One global and generic handler achieve application soundness | 1 | 209 |
| Per Group | One handler is provided for each group of non-functional exception (see 2) | 7 | 1561 |
| Per Communication | Every communication exception has 2 handlers : remote object level and VM level | 2 * 6 = 12 | 2833 |

**Table 1.** Space Requirements Depends of the Number of Handlers

**Time Complexity :** Adding and removing handlers do not break overall performance of the system. Research of handlers is complexity-less, thanks to hashtable properties. We raised a huge number of exceptions and measured time to find handlers. The ratio is 1:4.

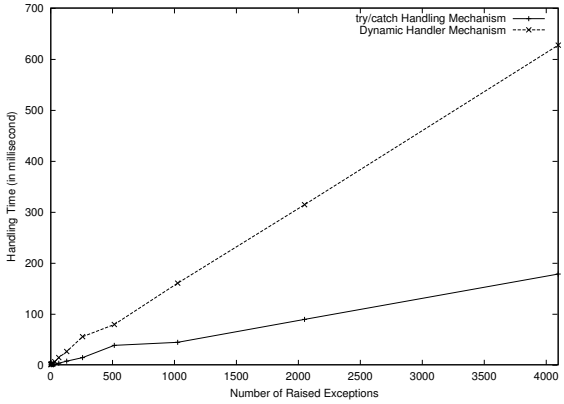



**Fig. 5.** Time Complexity

# Using Exception Handling for Fault-Tolerance in Mobile Coordination-Based Environments

Giovanna Di Marzo Serugendo[1] and Alexander Romanovsky[2]

[1] Centre Universitaire d'Informatique, University of Geneva,
CH-1211 Geneva 4, Switzerland
Giovanna.Dimarzo@cui.unige.ch

[2] School of Computing Science, University of Newcastle,
NE1 7RU Newcastle upon Tyne, UK
Alexander.Romanovsky@newcastle.ac.uk

## 1 Introduction

Mobility of users and code, coupled with todays powerful handheld devices, allows us to anticipate that a class of applications, which will take more and more importance in the next years, is that of mobile agent-based applications. Indeed, handheld devices need light code that can be freely moved from one device to another, according to the user's mobility or needs. Mobile agent-based applications typically run on a mobile coordination-based environment, where programs communicate asynchronously through a shared memory space. There is a number of outstanding issues in providing fault tolerance of such applications. The aim of this paper is to propose an exception handling model suitable for mobile coordination-based environments.

### 1.1 Exception Handling

One of the chief trends in providing dependability of modern systems is the decreasing role played by tolerance to hardware-related faults. This is due to several factors: improvements in hardware quality, a dramatic growth of the complexity of software, the increasing involvement of the unexperienced users in managing such systems, a growing variety of abnormal situations in the system environment. As a result of this, modern applications have to be designed in such a way that they are capable of dealing with a growing number of various abnormal situations in a disciplined fashion. The only general solution to these problems is to systematically incorporate software fault tolerance into the applications. One approach is to use backward error recovery techniques (such as rollback) which can be made almost transparent for the application. Unfortunately it is not general, usually quite expensive, and often not-applicable at all. This is why employing application-specific exception handling is nowadays playing a major role in building complex dependable applications.

Exception handling relies on features for declaring exceptions in different scopes (exception contexts) and associating handlers with them, as well as on

ability to raise exceptions and propagate them outside the (nested) scopes. Most practical languages (including Java and Ada) as well as a number of component technologies (such as EJB) provide features for handling exceptions. Exception handling features should match the characteristics of the application to be developed and the environment, the design paradigm, the computational model and the language/technology used [8]. Developing specialized exception handling mechanisms is an area of a very active research. For example, a number of object-oriented mechanisms were proposed around 1990, atomic action based mechanisms for providing fault tolerance (exception handling) in concurrent cooperative systems were developed in mid 80s.

### 1.2 Coordination-Based Mobile Environments

Mobile coordination-based environments usually follow a data-driven coordination model [3], using a shared data space à la Linda. Agents coordinate in a Linda space by inserting, reading or removing tuples of data from a blackboard. Tuples are retrieved according to an associative search. Mobile coordination-based environments provides several tuple spaces and ways for agents to denote and access the different spaces.

Among mobile coordination-based environment, we can cite Lime [7], MARS [2], and Lana [1]. Lime (Linda in a mobile environment) is well suited for both logical mobility of agents and physical mobility of devices. Lime mobile agents coordinate through Linda tuple spaces using a fixed set of interaction primitives. Lime tuple spaces at different sites are merged together and agents access them transparently as if they were local. MARS (Mobile Agent Reactive Spaces) is an object-oriented Java-based environment, where Java agents coordinate through an object-oriented tuple space à la Linda, using a programmable set of primitives. A MARS agent can only access the tuple space associated to the host where it is currently executing. Lana is an object-oriented Java-based environment, which combines provision of coordination and security and is designed to run on both standard user PCs connected to the Internet and handheld devices. Lana agents coordinate through an object-oriented tuple space à la Linda, using a default fixed set of primitives, which can nevertheless be extended by the programmer. An agent can access both remote and local tuple spaces. Lana defines protection domains for information access, and prevents application crashes by considering network failures as normal events.

### 1.3 Fault-Tolerance in Mobile Coordination-Based Environments

Early work on fault-tolerance in coordination-based environments focused on using transactions in Linda-like environment. Two additional primitives allow all produced tuples to be retained until the transaction commits, the tuples are then actually added to the tuple space. Retaining tuples ensures the transaction semantics, but raises problems among dependent agents. A relaxed version [9]

provides the all or nothing transaction semantics, avoiding problems among dependent agents, by relaxing the atomicity property. Tuples are available immediately in the tuple space, even if later the transaction does not commit.

Mobile coordination-based environments, like WCL and JavaSpace, define notification mechanisms. Agents ask to be notified whenever a matching entry is written to the space. The notification mechanism, coupled with the notion of transactions, provides a fault-tolerant mechanism, where an event catcher is notified of matching entries for the duration of the transaction. In the case of Lana, fault-tolerance is achieved in an asynchronous fashion, through the notion of events. Events are deposited into the tuple space and retrieved by the corresponding agent: immediately, if the agent was waiting for the event; later on, if it searches for it later in the tuple space; or never, if the agent does not care about the event.

However, all the above mentioned mechanisms for dealing with fault-tolerance at the application level, suffer from the fact that exception handling, in its traditional sense, cannot be realized. Indeed, an exception, signaling an abnormal event, is not necessarily caught or handled, neither synchronously or asynchronously, by an agent.

### 1.4 Contribution

The focus of this paper is on discussing a new model of exception handling suitable for mobile applications developed in coordination-based environments. Our general view here is that exceptions are special (abnormal) events that cannot be treated as usual (normal) events. Exceptions have to be *always* caught and handled, they are abnormal events always needing reaction. Generally speaking, the agent putting tuple in the space and continuing its work acts under wrong assumption that its processing would go smoothly. The problems here are: due to decoupling event producers from consumers this agent continues its execution as if no exception has been raised, so it is not ready to be involved in handling. Moreover, it can leave the location. There is clearly a need to have (to dynamically create or link) a local handler that would synchronously deal with exceptions, while the processes raising exceptions keep communicating asynchronously. In addition to this, it is important to make exception handling as flexible as possible. For example, to offer features to change dynamically the direction of exception propagation.

## 2 Running Example

We introduce a small banking system that operates with mobile agents and requires fault tolerance, and which is implemented in the Lana environment. This example will be used to illustrate the discussion on the fault-tolerant techniques intended for mobile coordination-based environments.

### 2.1 The Lana Environment

Since Lana will serve as a basis for our discussion on fault tolerance, we will first explain some Lana features. Communication inside a program occur through synchronous method calls. However, Lana programs communicate using *asynchronous* method calls. These calls are secure in that the calling program is given a fresh *Key* object when the call is issued, and only this key can be used to interpret the method reply. This mechanism prevents malicious programs from intercepting or tampering with messages destined for others.

*Events* are used for signaling returned values of asynchronous method calls. They are also used for signaling errors or exceptions, such as: security violation (no permission for a method call), or the fact that the required program has moved. Pre-defined event types include those: that indicate that a method has returned (`MethodReturn`); that the method call has failed due to the invoked program having moved (`MigrationEvent`); no access right having been granted (`SecurityViolation`); the execution of the method code has generated an exception; the remote platform generated an (system) exception. *Keys* are used to lock objects and events. *Unique* keys are automatically generated for each asynchronous method call. *Fixed* keys can be generated by several programs, they allow transfer of well-known object copies through the use of a common key.

The asynchronous notification of events is at the heart of the current fault-tolerance mechanism provided by Lana. The typical scenario is the following: an agent performs first an asynchronous method call, it then continues its execution (without waiting for the method return), once it is interested in having the method return, it performs an `observe()`, which blocks the agent until the method has returned. The return can either be the expected result, or an "exception" explaining the problem. Lana allows such exceptions to be generated from a remote platform, however the `observe()` command must be local, i.e., an agent that moves must delegate the `observe()` to an agent that stays in the platform from where the call has been issued.

### 2.2 Market Place Example: Lana Design

Figure 1 shows a user, wishing to acquire some product. He launches a buyer agent that will roam the Internet searching for a seller agent offering the requested object. The buyer and seller agents will meet in an electronic market place, where they will exchange information regarding products that their respective users desire to buy or sell. Each agent is equipped with an e-purse holding some amount of electronic money. If the two agents have matching requests, they reach an agreement (the product is booked, and a contract is concluded), and the payment is then realized by transferring some money from the e-purse of the buyer agent directly to the e-purse of the seller agent. In this case the e-purse acts as cash money, there is no need to ask for a bank certification. If the payment fails, then the agreement has to be cancelled, and the seller agent releases the product. An e-purse is implemented by an additional agent that handles

the amount of money of the e-purse, access privileges, deposit and withdrawal operations.

In this scenario, the buyer agent can be either alone or composed of several agents roaming the Internet simultaneously. It may be difficult or even impossible for the user to contact the buyer agent or for the buyer agent to contact the agents distributed world-wide. Indeed, a mobile IP scheme enabling to contact mobile agents irrespectively of their position is difficult to consider when agents neither know each other nor participate in the same application.

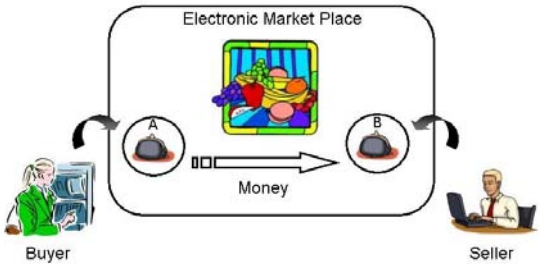



**Fig. 1.** Market Place Example

This examples consists of two main Lana agents: the buyer and the seller agent that work on behalf of the buyer, and seller respectively. Once both the buyer and seller agents have reached the market place, a sequence of data exchange occurs through the message board (in the case where there is no error): the buyer (seller) agent inserts a buying (selling) request respectively in the tuple space. The buyer agent retrieves the selling request and proposes a contract, which is accepted by the seller. The interaction ends with the payment, i.e., by transferring money from the buyer e-purse to seller e-purse.

We will now consider three types of errors: system error, application errors, network failures; and see how they can be handled within the current Lana model:

- *System error.* The local or remote platform generates an error, e.g., the code cannot be executed, the called agent has moved, etc.
- *Local Application Errors.* For instance, we can mention:
  - there is no offer matching the request;
  - there is a bug in the seller agent code: it is impossible to reach an agreement or to access to e-purse;
  - the buyer e-purse does not contain sufficient money;
  - the buyer e-purse does not present sufficient privileges, e.g., a configuration error does not authorize money to be withdrawn from the e-purse.

- *Distributed Application Errors.* Several agents work cooperatively, e.g., to buy several items of the same series. In the case of an error, there is a need for a collective recovery of the error.
- *Network Failures.* There are communication failures between the buyer and seller agent platforms, or between the agent platform and its user.

In the case of system errors, the local or remote Lana platform generates an event, i.e., it inserts the corresponding tuple in the shared memory space. Thanks to the key, the tuple is then retrieved by the agent that was waiting for the corresponding operation.

In the case of application errors, e.g., there is an error during the payment due to insufficient money, the buyer agent inserts a payment fail event, retrieved by the seller agent that was waiting for the payment. The latter cancels the transaction, and reinserts the selling request. Finally, the buyer agent informs the buyer in the original platform that an error occurred during the payment, it inserts the corresponding event in the original platform.

The case of distributed applications errors is the most interesting one. Indeed, the problem here is to reach, even asynchronously, but necessarily, agents whose location is not known. A possible approach to realizing this in Lana is for the agents to agree upon a common platform where events, related to abnormal situations, are stored. After that both the Buyer and the Seller agents leave an additional assistant agent in this platform, whose role is to observe those events, and to inform their respective agents.

Finally, the Lana model allows network disconnections to be handled in a very similar way. Agents are informed of the failure through events, and either wait for the availability of the connection, or continue their execution until the connection is up, or definitely give up.

### 2.3 Analysis

Lana treats exceptions as the conventional tuples and offers basic primitives for implementing exception handling. But we have found that there is clearly a space for expanding these features to help application programmers in developing fault tolerant applications in a safer and less error-prone way. First of all, Lana allows exceptions to be left unhandled - which is clearly error-prone and can have serious effects on system reliability. Secondly, this model mixes normal and abnormal flows of control and code, and does not separate sufficiently the normal system behaviour from the abnormal one - which is the main idea behind exception handling [6] (as a matter of fact in their early work the authors of Lana stated clearly [1] that exceptions should not be introduced as normal events). Thirdly, the Lana model does not include any specific support for exception handling and leaves all complicated issues with the application designers. These has several serious consequences which may complicate system design and make it more error-prone. One of the example is that to guarantee that the information is delivered the producer should wait for the notification to be implemented at the application level, in the code of both the producer and the consumer. Another

example is that Lana does not provides any systematic ways for transferring responsibility for handling exceptions from the producer of an event to some other process/code. Moreover, this model does not introduce the concept of exception context, which is crucial for any exception handling methodology - which makes it impossible to understand who handles exceptions, when and if they are handled at all. Too many responsibilities (and room for mistakes) are left with application programmers. In addition to this Lana exception handling does not support nesting, nor it supports cooperative handling that involves several processes.

## 3 Exception Handling Model

In this section we discuss our approach to introducing coordination-based exception handling which specifically focuses on code mobility.

### 3.1 Requirements

Novel fault tolerance techniques to be applying in developing complex mobile systems have to correspond to their specific characteristics. As our analysis in [4] shows the main way of ensuring fault tolerance in such systems is by employing application-specific exception handling mechanisms. Among other important properties these mechanisms should be light and flexible, they should allow for dynamic adjustments and for autonomous (localized) handling of abnormal situations. These mechanisms should be suitable for open systems in which mobile agents are to operate in unknown environments and be capable of dealing with abnormal situations that are not known in advance.

### 3.2 Model

The first problem here is to stay within decoupled (asynchronous) communication model and at the same time to guarantee delivery of exceptions to handlers. Only in this case we can guarantee that all exceptions are caught and after that handled. Our idea is to keep using usual standard asynchronous ways of communication (i.e. keys or events) but to make sure that, should an exception be raised, the handler always exists in the location.

As depicted in Figure 2, the underlying idea here is that we assume that, if an `event` is consumed by a program `EventConsumer`, and an exception `E` is raised during processing of this event, then this event is the cause of the exception, and such exception should be treated outside `EventConsumer`. One possible solution is to allow only synchronous communication, in which case the producer of the event, `EventProducer` would be the best handler. But if we want to allow asynchronous communication and agent mobility we cannot bind the producer. So the only possible solution is to create a new process `H(E)` to handle the exception.

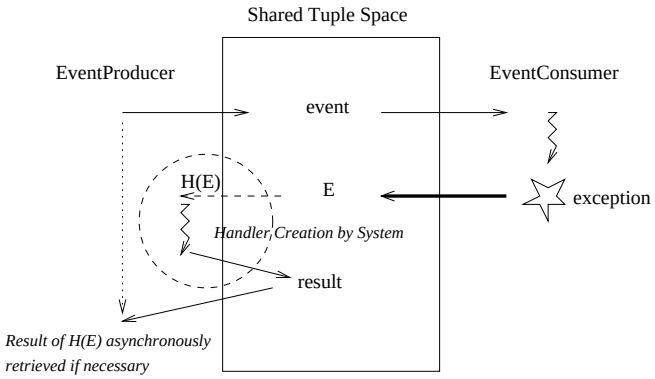



**Fig. 2.** Exception Model

The specialized process `H(E)` is created when an exception `E` is signaled by `EventConsumer`. In our approach any exception is a tuple of a special type: there is always a process waiting to handle (i.e. to consume) it. Such handler `H(E)`, representing the exception context, is usually designed by the developer of `EventProducer` but it can be designed by the designer of `EventConsumer` as well. In the latter case either the handler provided by `EventProducer` is ignored in runtime and only the handler supplied by `EventConsumer` is used, or both handlers receive exception `E` (in which case they can decide to handle it cooperatively). The handler process usually completes handling and dies. But they do not have to. Generally speaking, they can be involved in further system execution for as long as necessary.

In our approach each tuple `T` produced by `EventProducer` has a number of exceptions declared in its signature: `E1`, .., `En` in addition to a set of parameters. When `T` is put into a tuple space the system should have references to declarations of n handlers: one for each exception. The handler process is created locally, when an exception is signaled but it does not have to be always local: it can move to better handle the exception. But it is important to always create it when an exception is signaled. We believe that it would be wrong to make any existing process to be a handler because there is no guarantee that it will handle the exception without delays: it can move to another location before an exception is signaled or it can be busy doing other job and because of the asynchronous nature of communication may decide to handle it when it is too late.

Our scheme allows defining a process handling several (or, even, all exceptions) from the signature of `T`, or, even, from the signatures of different tuples.

One problem we are planning to address is: what happens if `EventConsumer` moves to another location and signals exception `E` while continuing processing the consumed tuple `T` in this location. We should be able to create handler `H(E)` in the new location because this is where the exception tuple is put in the local tuple space.

To make our scheme even more flexible we plan to allow: an exception to be propagated to several handler processes and the handler and the exception to be dynamically associated (e.g. while inserting `T` into a local tuple space).

Another feature which would add flexibility to our scheme is to allow any existing process, which is local to `EventConsumer` and to exception `E`, to handle it together with standard handler processes. In particular, it seems to be useful to allow `EventProducer` to join in the handling.

### 3.3 Market Place Example Revisited

We demonstrate our approach using the market place example introduced earlier.

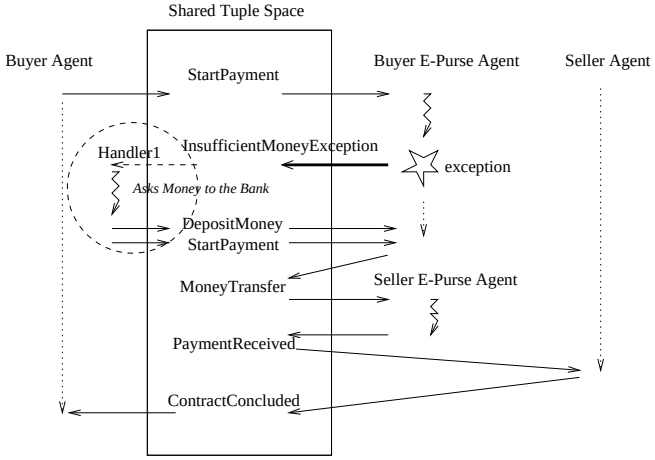



**Fig. 3.** Example Revisited: Successful Recovery

In this particular scenario the Buyer Agent asks its E-Purse Agent to release the money and provides the E-Purse Agent with a name of its proxy agent (Handler1) that can be locally created when necessary to deal with exceptions. The Buyer E-purse Agent raises an exception due to the fact that there is not enough money in the e-purse for realizing the payment. The system then created Handler1, which catches the exception. Handler1 accesses the bank, in order to transfer money from the bank account to the e-purse (see Figure 3). It deposits then the money on the e-purse, and starts again the payment procedure. The money transfer from the Buyer e-purse to the Seller e-purse occurs now correctly. The Seller e-purse agent then informs the Seller agent that the payment has been realized, and the Seller agent finally informs the Buyer agent that the transaction was successful.

Figure 4 shows a different case. Handler1 does not succeed in retrieving additional money from the bank (the bank account is not sufficiently furnished, or there are no privileges). Handler1 informs the Buyer Agent through a regular

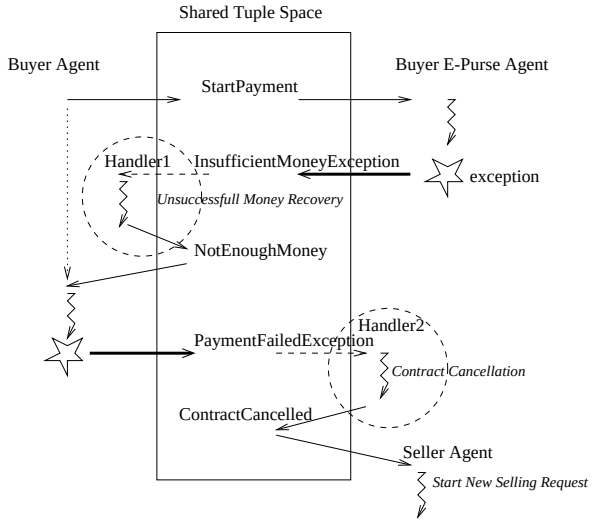



**Fig. 4.** Example Revisited: Failed Recovery

event about the insufficient money present in the e-purse. The Buyer Agent is not able to cope with this situation, and raises an exception, caught by Handler2, provided by the Seller Agent. Handler2 is in charge of cancelling the on-going contract, it then informs the Seller Agent, that subsequently starts a new selling request. As an alternative, we could also envisage that Handler2 creates an additional Seller Agent that will start the same selling request, leaving the original Seller Agent free to continue its work (e.g., buying a series of items). This is particularly useful, when the original Seller Agent needs to move.

This simple example allows us to draw several conclusions:

- in Lana the interacting agents have to always wait until the end and until all notifications have been received, but in our approach they do not because the handler agents can handle problems (our solution allows us to build really asynchronous systems);
- moreover, our approach allows agents to freely move to other locations and start other work without waiting for all notifications - if an exception is signaled in the original locality there always is a handler to deal with it locally;
- in Lana all handling is a part of the Seller, the Buyer and the E-Purse Agents but in our approach handling is implemented in separate handler agents - our design is cleaner and we separate the normal code from the abnormal one. Because of this, our approach is more flexible. We can, for example, use different handler processes for different locations which the Buyer Agent visits, while in Lana handling is hardwired into the Buyer Agent.

With respect to dealing with distributed application errors we will be extending our approach with the concept of the context (see the Discussion Section)

but in Lana everything again should be developed explicitly by the application programmers with no additional support from the system.

It is clear that any particular feature which our approach provides can be implemented in Lana or any other coordinated models. The real question is how expensive and error-prone these solutions are and what sort of support the programmers have to systematically employ exception handling. This is where our solution provides a number of benefits ensuring, for example, that any exception is always treated. In addition to that our approach respects asynchronous communication mechanism and does not restrict unnecessarily process mobility.

## 4 Related Works

To the best of our knowledge there are no general exception handling features developed for mobile coordination-based environments meeting the requirements above. There are only few relevant papers on exception handling in coordination-based systems. Diaz et al [5] put forward a basic exception handling framework for the logic channel-based coordination model. In this model when a process is created a special logical channel is associated with it, so when the process raises an exception it is propagated through this dedicated channel. The idea here is that when a process is created a special process used for handling all its exceptions is created and sent as well. This scheme is not oriented towards mobile environments, it relies on asynchronous exception handling making no difference between normal and abnormal events (we will discuss this issue in detail later on) and, besides, associating handlers with the process processing information appears to be very static. Two more issues with this approach are: there is one handler for all possible exceptions raised by a process and there is no support for raising exceptions in several processes.

Another relevant work, by Rowstron [9], considers fault-tolerance for distributed tuple space, in the framework of stationary agents. It introduces the notion of "mobile coordination", where mobile is related to the movement of coordination primitives, forming together a coordination operation, to the server storing the tuple space. The operation is executed according to an all or nothing semantics, which nevertheless allows intermediate tuples to be available in the tuple space. Fault-tolerance for small segments of programs is then realized by transferring the corresponding primitives, and associated state, to the server. This approach does not enable the propagation of exceptions, in their traditional sense, outside the coordination operation. However, the additional notion of "agent will" allows to perform some exception handling outside coordinations. Wills are mainly used to cleanup the tuple space if an agent fails, and are associated with a tuple space, and the same will is applied to all operations.

## 5 Discussion

In our future work we plan to apply the ideas discussed in the paper for developing a novel exception handling mechanism for Lana. One topic of our following

work will be introducing rules for scoping to involve all processes from the same scope in cooperative handling of any exception signaled by any of them. All processes belonging to the same locality is the most likely candidate for introducing scoping rules but we are considering other ways of scoping as well. This would resolve the following problem with the solution proposed in Section 3: we implicitly assume here that the consumer always detects an error and signals an exception before it produces any tuple or before it starts consuming the next tuple - which is in effect a very strong assumption for many practical situations.

## 6 Acknowledgment

We are grateful to M. Pawlak and C. Bryce for their invaluable comments regarding Lana and its exception model. A. Romanovsky is partially supported by EPSRC/UK DOTS Project. G. Di Marzo Serugendo is supported by Swiss NSF grant 21-68026.02.

## References

1. C. Bryce, C. Razafimahefa, and M. Pawlak. Lana: An approach to programming autonomous systems. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming*, volume 2374 of *LNCS*, pages 281–308. Springer-Verlag, 2002.
2. G. Cabri, L. Leonardi, and F. Zambonelli. Reactive tuple spaces for mobile agent coordination. In K. Rothermel and F. Hohl, editors, *Proceedings of 2nd International Workshop on Mobile Agents*, volume 1477 of *LNCS*, pages 237–248. Springer-Verlag, 1998.
3. P. Ciancarini, A. Omicini, and F. Zambonelli. Coordination Technologies for Internet Agents. *Nordic Journal of Computing*, 6(3), 1999.
4. G. Di Marzo Serugendo and A. Romanovsky. Designing fault-tolerant mobile systems. In G. Reggio N. Guelfi, E. Astesiano, editor, *Scientific Engineering for Distributed Java Applications. International Workshop, FIDJI 2002*, volume 2604 of *LNCS*, pages 185–201. Springer-Verlag, 2003.
5. M. Diaz, B. Rubio, and J. M. Troya. Distributed Programming with a Logic Channel-based Coordination Model. *Computer Journal*, 39(10), 1996.
6. J. B. Goodenough. Exception Handling: Issues and a Proposed Notation. *Communications of the ACM*, 18(12):683–696, 1975.
7. G. P. Picco and G. Catalin-Roman. Lime: Linda meets mobility. In *Proceedings of International Conference on Software Engineering*. IEEE Computer Society Press, 1999.
8. A. Romanovsky and J. Kienzle. Action-oriented exception handling in cooperative and competitive con-current object-oriented systems. In A. Romanovsky, C. Dony, J. Lindskov Knudsen, and A. Tripathi, editors, *Advances in Exception Handling Techniques*, volume 2022 of *LNCS*, pages 147–163. Springer-Verlag, 2001.
9. A. Rowstron. Mobile co-ordination: Providing fault-tolerance in tuple space based co-ordination languages. In P. Ciancarini and A. Wolf, editors, *Coordination Languages and Models: Coordination99*, volume 1594 of *LNCS*, pages 196–210. Springer-Verlag, 1999.

# Primitives and Mechanisms of the Guardian Model for Exception Handling in Distributed Systems

Robert Miller and Anand Tripathi

University of Minnesota, Computer Science Department,,
Minneapolis, MN 55455 USA
{rmiller, tripathi}@cs.umn.edu

**Abstract.** We believe the fundamental problem with distributed exception handling is invoking the semantically correct exception handlers in all the distributed processes that are required to participate in the recovery. Existing distributed exception handling techniques emphasize raising the same exception in all the required processes with a transaction-like program structure. As useful as that is, there are many applications that do fit easily into that paradigm. A single raised exception must be able to represent adequately all concurrently signaled exceptions, and a transaction-like structure may be too rigid for an application. We present the primitives and mechanisms for an abstraction called guardian for exception handling in distributed systems that can overcome those limitations. Using an example, we show how the guardian can be used to augment and enhance an existing distributed exception model.

## 1 Introduction

This paper discusses the primitives and mechanisms of the *guardian exception handling model* for distributed systems. The conceptual foundations of the guardian model were presented in [6], and in [7] the initial set of guardian primitives was introduced. In this paper, the full set of guardian primitives and mechanisms are described, and an example using the guardian that shows how it can augment or enhance an existing distributed exception handling model is shown.

Exception handling in distributed systems differs significantly from sequential exception handling. In [9] two reasons are given: distributed systems need exception handler communication and coordination, and multiple exceptions may be concurrently signaled. Concurrent exception models have the capability of handling concurrently signaled exceptions, and are based on *exception resolution* and a program structure based on transaction-like semantics [1]. Exception resolution translates or maps concurrent exceptions into one exception (a *resolved* or *concerted exception*). An exception that is to be signaled to the processes (the *participants*) of a distributed application is called an *external exception.* An exception that is handled locally in the process is an *internal exception*. Examples of concurrent exception models are CA actions [9], Arche [3], OMTT [4], and conversations [1].

Concurrent exception models assume that a semantically correct handler is invoked in each participant. To allow this, the exception is raised in all participants, including the signaling participant. Since a distributed system is asynchronous, it is possible for one participant to have a different exception handler enabled than another participant. Relying on handler communication to ensure the correct handlers are invoked may be a highly complex task. To simplify this, concurrent exception models assume a program is structured in such a way as to ensure that the correct handlers are invoked so handler communication is not required. Typically, the structure is transaction-like that has synchronized entry and exit points. Any external exceptions raised in a participant are within that boundary. Because all participants are executing within the same boundaries, it can be assured that the correct handler is invoked in all participants.

The next section describes the limitations of the existing concurrent exception handling models. Section 3 describes the programming primitives and mechanisms for the guardian model. Section 4 presents an example using this model, and finally Section 5 concludes this paper.

## 2 The Problem

Concurrent exception models have a defined behavior for concurrently signaled exceptions, and exception handlers coordinate not by direct communication, but by having handlers for the same exception invoked in all the participants. However, there are still three limitations with these models.

First, concurrently signaled exceptions are assumed to be related in some way so that a meaningful resolved exception may be obtained. This is not always easy to do [8]. Secondly, external exceptions are signaled explicitly from a participant, so exception conditions outside the participants may not be detected. Finally, there is the restriction of a transaction-like program structure. Transaction-like structure is useful in many applications, but there are also other applications that have an asynchronous aspect to them, such as management or monitoring applications, where such a structure may be difficult to apply.

We believe the problem with distributed exception handling is invoking the semantically correct handler in each participant. Determining the correct handler is the main issue, and not the raising of exceptions. In the guardian model, determination of the correct handler is by using the guardian to direct each process to a correct exception handler by raising in each process a possibly different exception and specifying the context in which it should be handled by the process. This is different from the approach of concurrent exception models, in which the same exception is raised in all processes that need to participate in the recovery.

The purpose of allowing different exceptions to be raised in each participant is to allow the guardian to orchestrate the recovery action. A raised exception implies a specific recovery action a participant is to do. For example, say there is a pipeline of three processes *A, B,* and *C*. Should *B* fail, the guardian would signal to *A* an exception that its downstream neighbor has failed, and to *C* an exception that its upstream neighbor has failed. With a guardian, participants are freed from the burden of main-

taining any configuration information and relating it to a process failure to determine the semantically correct recovery action. No transaction-like structure is needed for the correct exception handlers to be invoked (though that structure may be useful for other reasons).

## 3 The Guardian Model

The *guardian exception handling model* [6] [7] is based on the timed asynchronous computation model [2], global exception handlers, separation of global exception handling from the exception handling local to a process, and an extended fault model. The guardian is a distributed global entity that orchestrates the exception handling action by directing each involved process. The directing is by raising in a process an appropriate exception, which may differ from the exception raised in another process. Application defined recovery rules determine the exception the guardian raises in each participant, which in turn causes the correct exception handler to be invoked. The guardian model has three elements: the concept of an *exception context*, a global entity called a *guardian*, and a set of guardian primitives that the participants use.

### 3.1 Exception Context

An *exception context* is an execution phase or region of a program. Contexts may be nested by a process entering a new context reflecting the static structure of the program (as nested blocks), or the dynamic function call sequence. Contexts are application specified as a symbolic name, and so they may have different meanings based on the application. A context may be used to represent a number of different abstractions such as an invocation stack frame, checkpoint, recovery block, transaction or conversation context, a barrier synchronization point, or an assert point.

The *raising context* is the context a process is in when an exception is raised in it. A *target context* is the context that an exception is to be handled in. There is a reserved context called *Init*, which is used as top-level context. Its purpose is to allow a process to have a context before it defines one, and to have a target context to prevent a process from handling an exception, such as a terminate exception.

The purpose of contexts is to provide a mechanism to invoke correct exception handlers. This is done in two ways. First, when an exception is raised in a process, a target context is specified in the exception object, since the raising context and the target context may not be the same. Second, contexts provide a means to give a dynamic meaning to an exception based on the current program flow, similar to a dynamic call chain.

### 3.2 The Global Exception Handling Model

The basic elements of the guardian model are shown in Figure 1. Each process, such as $P_{i.}$, in the application is a *participant.* The guardian is logically replicated at all

participant environments $E_i$ with a *guardian member* $GM_i$ as a co-process. The guardian members form a process group implementing the global abstraction for its guardian.

The members in this group communicate with each other using reliable group communication primitives. Group messages are delivered and processed within time bounds; if the bound is not met then the message is considered lost [2]. Groups allow for reliable and totally ordered message delivery within the bounds and support virtual synchrony (i.e., membership change messages are ordered with all other messages). The guardian defines *membership exceptions*, which indicate a change in the membership of the guardian group. The guardian raises an exception when a participant joins or leaves the activity.

The guardian supports global exception handling in the following manner:

1. Using the guardian method *enableContext*, a participant defines a context and the exceptions that the context can handle. The guardian member for the participant maintains a stack of these contexts and the related set of exceptions. When a context is returned from, the participant invokes *removeContext* which pops the last context from the stack. A program typically has a small number of contexts, where each context represents a recoverable block, such as a *try-catch*. Each participant may have a different context stack.
2. When a participant signals a global exception, it invokes a guardian method called *gthrow*. The participant blocks in *gthrow* until an exception is raised. The associated guardian member sends a message that represents the exception to the other guardian members through the guardian group. If exceptions are signaled simultaneously by multiple participants, then group communication orders the respective messages for all guardian members
3. When a guardian member receives the exception message from its group, it checks if its associated participant is suspended. If the participant is not, then the guardian member suspends the participant. This ensures that all concurrently signaled exceptions are known to the guardian.
4. To suspend a participant that supports interrupts, the guardian member interrupts the participant, and the interrupt handler invokes a guardian method called *checkExceptionStatus*. The method checks if there are any exceptions that have to be delivered to the participant. If not, the method returns, otherwise the method blocks until an exception is raised from it. No group communication is used for the check. To suspend a participant that does not support interrupts, the participant periodically invokes *checkExceptionStatus*.
5. As each participant is suspended, the guardian assembles all the participant context lists together. Once all participants are suspended, the guardian invokes application-defined recovery rules, for example exception handling patterns [5].
6. The guardian provides to the recovery rules the exceptions signaled, and all context lists. A matching rule defines, for each participant, the target context and exception to raise in that participant. The guardian members collectively raise in their respective participants the exceptions defined by the rules.
7. After the exception is raised in a participant, exception handlers are searched in the participant. Each handler is expected to invoke the guardian method *propagate*, which returns to the handler whether it should handle the exception or

```
    if signaled exception is E then {
  for each specified identifier expression Pe do {
    let PL be the list of participants  satisfying Pe;
    for each specified selection predicate S do {
      let Ps be the  subset of PL satisfying S;
      for each participant  p in Ps do {
          someComputation();
          //insert exception object in Output Exception List
          OEL.insert(p, Ep(Cp));
      }
    }
  }
}
```

**Fig. 1.** Single exception rule structure

propagate it. The method simply compares the exception target context with the handler's context.

The recovery rules may map a single signaled exception into a different exception or target context on each participant. Typical exception resolution is possible by transforming all the concurrently signaled exceptions into the same exception and target context in all participants.

The guardian also supports the notion of an interrupt with the methods *enableInterrupts* and *disableInterrupts*. It is assumed a guardian member may interrupt its associated participant, and the participant invokes an interrupt handler that can query the guardian to determine if the interrupt is from the guardian. If the interrupt is due to the guardian, then the interrupt handler can signal an exception to the participant program. This is similar to the Java model of interrupt using the *interrupt* method.

If a participant has interrupts enabled, then *checkExceptionStatus* does not block. When a participant receives an interrupt, the interrupt handler invokes *checkExceptionStatus*. If the interrupt is due to the guardian, then an exception message has already been sent, and *checkExceptionStatus* will throw the exception received. Should a participant execute *gthrow* with interrupts enabled, then *gthrow* returns after sending its exception message to the guardian. When an interrupt from the guardian occurs in the participant, *checkExceptionStatus* is invoked as explained above that signals the received exception.

### 3.3 Recovery Rules

A program extends the guardian with application-specific rules. The rules are logically replicated with each guardian member. The rules mechanism has as its input all concurrently signaled exception objects, and the context stack for each participant. The rules mechanism outputs a list of exception objects: one object to be raised in each participant that matches the rule. Each exception object contains the target context in which the participant should handle the exception.

Participants are identified using their contexts rather than fixed symbolic names. A participant identifier is a context list expressed as slash-separated context names. An

```
if (E is a subset of the signaled exceptions)
    OSL.insert(exceptionResolution(E));
```

**Fig. 2.** Concurrent exception rule structure

identifier represents a subset of participants whose current context matches the specified identifier.

A fully qualified identifier includes the entire context as the identifier, while a partially qualified identifier is expressed as a regular expression of context names. For example, a context stack *C1→C2→C3* has a fully qualified identifier as *C1/C2/C3*. Using a partially qualified identifier allows greater flexibility in identifying a subset of participants, e.g., **/C2* matches all participants whose current context is *C2*.

There are two kinds of rules, one for single exceptions, and the other for multiple concurrent exceptions. Rules are searched in lexical order, with concurrent exception rules first. A rule for a single exception constructs a list of exceptions, one exception object that the guardian will raise for each member of a set of participants.

The general structure of a single exception rule is in Figure 1. For a given signaled exception, one or more identifier expressions $P_e$ are evaluated serially. The list $P_L$ of participants is constructed for each $P_e$ by matching the context lists of the current set of participants with $P_e$. For each selection predicate S, a subset $P_S$ of $P_L$ is computed using S. Furthermore, for each participant in $P_S$, the guardian may perform some application-specific computation to determine the exception to be raised in that participant. The guardian then adds that exception to the Output Exception List (OEL). In Figure 1, for participant p an exception object of type $E_p$ with target context $C_P$ is added to the list.

After the guardian has completed building the OEL by executing all applicable rules, it raises each of the exception objects in this list in the correspondingly specified participant. Each exception object specifies the target context in which the participant should handle the exception.

For concurrent exceptions, the guardian provides flexible exception resolution that can be used with existing exception resolution methods or new ones, such as the highest priority exception or to serially apply the concurrent exceptions to the sequential rules. The application defines *N* levels of priority, with level 0 the highest priority. Each exception that may be signaled is assigned a priority, with a default priority being the lowest level (*N-1*). The conversation and Arche models would use only one level.

When exceptions are signaled concurrently, the exceptions are first sorted by priority level. After the sort there is a vector of priority levels, with each level having a set of signaled exceptions for that level. Each level, in priority order with level 0 first, has the concurrent rules applied to the signaled exceptions at that level. Figure 2 shows the general concurrent exception rule structure. Note that *E* in Figure 2 is a set of one or more exceptions, and all exceptions specified by *E* must be in the currently signaled set of exceptions. A concurrent rule for resolved or concerted exceptions uses a value of *E* or *null*, since *null* is a member of all signaled exceptions sets.

The concurrent rule uses an application-defined resolution function that outputs zero or more exception objects that are put on the *Output Sequential List* (*OSL*). As

each priority level has the concurrent rules applied to it, the sequential exception objects from that level are appended to the *OSL*. When all levels have been processed, there is one *OSL* that has all the exception objects in priority order that the sequential rules are applied to serially. To the sequential rules, each exception object on the *OSL* appears as though an exception had been signaled by a participant.

The guardian defines a default rule of exception resolution by finding the root of the smallest sub-tree that contains all the concurrently signaled exceptions. This is the same rule as conversations and CA actions use.

### 3.4 Guardian Model Comparison

As has been discussed above, the two main differences between the guardian model and other models is that the guardian may raise different exceptions in each participant and a context is used to determine at what execution point a program is at. This has several implications.

First, the guardian model is not meant to replace a program's structure. The guardian model can be used to implement an exception model in a rules-based way. For example, a context is not meant to replace a conversation, but rather can be used to implement a conversation. Second, there is no known exception model that is suitable for all known exceptional conditions. A guardian allows the exception model to be changed during a program to better match the program's exception handling to the current execution phase of the program. Third, a model implemented using the guardian may be enhanced, such as with increased error detection capability (e.g., participant death or other system-types of exceptions). Fourth, a guardian is rule-based, meaning the exception handling intelligence could be partially removed from a program and placed in a system repository, particularly for conditions that require sophisticated actions. Programs that were not designed to have coordinated recovery could through if the programs are using the same guardian. If some contexts and recovery actions can be standardized or predefined, then a program gets these behaviors through the system, and so different systems can tailor the behavior. Fifth, exceptions could be used to indicate non-error situations, such as warnings of abnormal behavior that is not an error (such as an input being at a limit of the input's range).

The guardian also has disadvantages. For contexts to be meaningful, a program structure is needed that a context can be associated with. In order for different programs to use the same guardian concurrently or for system-defined guardian rules that can augment a program's rules, the context names need to be well-defined so that the guardian rules do not become confused.

## 4 An Example of Guardian Programming

The guardian model is not meant to replace existing exception handling models, but rather to enhance them. An example of this is with the conversation model. The normal conversation model can only detect failures that are explicitly signaled by a participant. For example, unexpected participant termination is not detected. Using the

```
Guardian myG;
GuardianBarrier b;
Context c1 = new Context("Conv", GlobalException);
 myG.enableContext(c1);
 try {
     myG.enableInterrupts();
      b.barrier(myG);
      myG.disableInterrupts();
      doWork();
      if (error)
        myG.gthrow(new GlobalException('Error');
     myG.enableInterrupts();
     b.barrier(myG);
     return;
 } catch(GlobalException ge) {
       if (myG.propagate()) throw;
      doCleanup();
 }
```

**Fig. 3.** Conversation using a guardian

guardian to implement conversations, failures detected by the guardian fault model (such as unexpected participant termination) are signaled as global exceptions automatically. The guardian model can also sustain K of N failures at a synchronized entry or exit point, while the conversation model can not. Lastly, implementing a related model, such as CA actions or OMTT, is mostly accomplished by changing the guardian rules.

Figure 3 shows how a guardian can implement a conversation. The conversation entry and exit points are implemented using a barrier that has been guardian-enabled. The *enableInterrupts* method allows the barrier to be interrupted with a global exception. The barrier is modified to check for a global exception if it is interrupted. After the barrier, *disableInterrupts* is invoked to preserve the conversation semantics of raising an external exception at exit points.

Each conversation is represented by an exception context. In Figure 3, the conversation has the context *Conv*. If a nested conversation is used, then the nested context simply has a different name, such as *Conv2*. The full context name is *Conv/Conv2*.

To see how contexts are used, say there are three processes *P1, P2,* and *P3*. All three processes are in the outer conversation *Conv*, and *P1* and *P2* are in a nested conversation *Conv2*. Should *P1* signal a global exception, it should only be raised in *P1* and *P2*. When *P1* signals the exception, it has a target context of *Conv/Conv2*. Only *P1* and *P2* are in that context (*P1* is in *Conv*), so the exception is raised only in *P1* and *P2*. The target context could also be specified as **/Conv2* to indicate all processes that are in the conversation *Conv2*.

If the exception is unhandled in *Conv2*, then the guardian re-raises the exception with target context *Conv* (the next enclosing context). It can be seen that now all three processes will have the exception raised in them.

The guardian recovery rules for conversations use the default rules. Multiple concurrently signaled exceptions are resolved into one resolved exception, and each participant has the same exception raised in it as all other participants.

## 5 Summary and Conclusions

The conceptual foundations of the guardian model were presented in [6], and in this paper we have presented the details of its programming primitives and its runtime

execution model. To understand the limitations of the existing distributed exception handling models, we have analyzed distributed exception handling requirements with respect to sequential exception handling models. Three basic differences between distributed and sequential exception handling are identified. This leads to what we consider is the fundamental problem with distributed exception handling: *each affected process must invoke the correct exception handler*.

We have shown how the guardian model for distributed exception handling addresses this fundamental problem using the concept of exception contexts and a set of programming primitives supported by a global exception handler called the guardian. We currently have a test implementation using multiple, distributed Java virtual machines. Future work includes incorporating the guardian in a Java-based agent programming system, and formalizing the guardian model.

## 6 Acknowledgements

This work was partially supported by NSF grants ANI 0087514 and ITR 0082215. Sujay Patankar extended the guardian model to include multiple JVM instances.

## References

1. Campbell, R. H., Randell, B.: Error Recovery in Asynchronous Systems. IEEE Transactions on Software Engineering, Vol. 12, (1986) 811-826.
2. Cristian, F., Fetzer, C.: The timed asynchronous distributed system model. IEEE Transactions on Parallel and Distributed Systems, Vol. 10, (1999) 642-657.
3. Issarny, V.: An Exception Handling Mechanism for Parallel Object-Oriented Programming: Toward Reusable, Robust Distributed Software. Journal of Object Oriented Programming, Vol. 6, (1993) 29-40.
4. Kienzle, J., Romanovsky, A., Strohmeier, A.: Open multithreaded transactions: keeping threads and exceptions under control. 6th International Workshop on Object-Oriented Real Time Dependable Systems (2001) 197-205.
5. Klein, M., Dellarocas, C.: Exception handling in agent systems. Third International Conference on Autonomous Agents (1999) 62-68.
6. Miller, R., Tripathi, A.: Exception Handling in Timed Asynchronous Systems. In: Ezhilchelvan, P., Romanovsky, A (eds): Concurrency in Dependable Computing. Kluwer (2002) 209-227.
7. Miller, R., Tripathi, A.: The Guardian Model for Exception Handling in Distributed Systems. Symposium on Reliable and Distributed Computing, (2002) 304-313.
8. Xu, J., Randell, B., Romanovsky, A., Stroud, R.J., Zorzo, A.F., Canver, E., von Henke, F.: Rigorous development of a safety-criticial system based on coordinated atomic actions. 29th International Symposium on Fault-Tolerant Computing, (1999) 68-75.
9. Xu, J., Romanovsky, A., Randell, B.: Concurrent Exception Handling and Resolution in Distributed Object Systems. IEEE Transactions on Parallel and Distributed Systems, Vol. 11, (2000) 1019-1031.

# Component Integration using Composition Contracts with Exception Handling

Ricardo de Mendonça da Silva, Paulo Asterio de C. Guerra, and
Cecília M. F. Rubira

Instituto de Computação - Universidade Estadual de Campinas
C.P. 6176, Campinas, SP, Brazil, 13084-971.
{ricardo.silva, asterio, cmrubira}@ic.unicamp.br

**Abstract.** The development of modern software systems usually needs to integrate autonomous component-systems which are developed independently from each other. Examples of such component-systems include COTS components, legacy software systems and Web Services. For developing this new kind of software system, we need innovative software engineering approaches that rely on the system's software architecture to achieve the desired quality properties of the resulting system, such as fault tolerance. In this paper, we propose an architectural approach to the dependable composition of component-systems based on composition contracts and an exception handling scheme which considers the concurrent execution of architectural components.

## 1 Introduction

Modern software systems, such as e-commerce and e-banking, usually are developed integrating component-systems, which are autonomous systems developed, maintained and concurrently operated by independent organizations. In a rapid changing world, these new software systems should be easily adaptable to changes in the business rules [6]. Moreover, many of these new software systems are becoming safety-critical because financial loss and even life loss can result from their failure [1].

In general, no assumptions can be made about the internal design and implementation of a component-system. For instance, when integrating a web service [2] whose actual implementation is dynamically bound at run-time, the only information available to the system integrator is the specification of the public interface. So, there are no guarantees about the quality attributes of the actual implementation, such as its correctness, availability and reliability. This implies that: in order to achieve quality properties, such as adaptability and dependability, we should focus on solutions mainly at the software architecture.

The proposed approach employs the C2 architectural style [4], which is a component-based style directed at supporting large grain reuse and flexible component composition, emphasizing weak bindings between them. By architectural style we mean a set of design rules that identify the kinds of components and connectors that may be used to compose a system or subsystem, together with local or global constraints on the way the composition is done [5].

In this paper, we propose an architectural approach to the fault-tolerant composition of concurrent component-systems, based on *composition contracts*. A composition contract extends the concept of *coordination contracts* [6] to include an exception handling scheme based on Coordinated Atomic Action (CA Action) [3]. A coordination contract is a connection that can be established between a group of objects through which rules and constraints are imposed on their collaboration, thus enforcing specific forms of interaction or adaptation to new requirements [6]. In essence, a coordination contract consists of a prescription of *coordination effects* that will be imposed on a collection of *partners* when the occurrence of one of the contract *triggers* is detected in the system. Coordinated Atomic Actions (CA Actions) [3] is a mechanism for structuring fault-tolerant concurrent systems that unifies the notions of forward and backward error recovery into its exception handling schema.

The rest of this paper is organised as follows. Section 2 gives a brief overview of how we have adapted the CA Action concept to obtain fault tolerance in component integration. Section 3 provides an overview of the C2 Architectural Style. Section 4 presents a software architecture that uses composition contracts with exception handling to build composite systems. Section 5 shows a case study illustrating the behavior of a fault-tolerant system. Finally, section 6 summarizes the conclusions of this work and discusses related work.

## 2 Exception Handling in CA Actions

A CA Action is designed as a multi-entry unit with *roles* which are bound to *action participants* which cooperate within the CA Action. The action starts when all *roles* have been activated and finishes when all of them reach the action end. If a participant raises an exception within a CA action, appropriate recovery measures should be invoked cooperatively, by all the participants, in order to reach some mutually consistent exception handling. A *resolution scheme* is used to combine multiple exceptions into a single exception if they are raised at the same time. The participants may initially apply a forward error recovery strategy aiming to mask the exception and complete the CA Action successfully, either with a normal result or a degraded (exceptional) one. If this initial strategy fails then the CA action should trigger backward error recovery in their participants in order to undo the undesired effects. If the CA Action cannot complete successfully but is able to restore its initial state, then the action is aborted, otherwise it fails.

*Composition contracts* differ from CA Action in allowing the component-systems to interact with external objects that are not transactional. This implies that a composition contract will guarantee only atomicity and consistency of its services, not embracing the full set of the ACID properties (atomicity, consistency, isolation, durability) of a CA Action. Systems can be designed recursively using action nesting. Fault tolerance features are always associated with such units confining all errors. When an action is not able to mask an error an exception is propagated to the containing action. This exception may be an *abort exception*, when the participants are left in a state free of effects of the action, or a *failure exception* otherwise, when the undo fails. In this last case the containing action is responsible for recovering the system state.

## 3 The C2 Architectural Style

In C2 architectural style, components of a system may be completely unaware of each other. The components communicate only through asynchronous messages mediated by connectors that are responsible for message routing, broadcasting and filtering. Both components and connectors have a *top interface* and a *bottom interface*. Systems are composed in a layered style. The *top interface* of a component may be connected to the *bottom interface* of a single connector. The *bottom interface* of a component may be connected to the *top interface* of another single connector. Each side of a connector may be connected to any number of components or connectors.

There are two types of messages in C2: *requests* and *notifications*. By convention, *requests* flow up through the system's layers and *notifications* flow down. In response to a request, a component may emit a notification back to the components below, through its bottom interface. Upon receiving a notification, a component may react, as if a service was requested, with the implicit invocation of one of its operations.

## 4 The Proposed Architecture

The proposed software architecture is organized in three layers (Figure 1). The *computational layer* encapsulates the service interfaces of component-systems, called *participant components*. Examples of participant components include COTS components, legacy systems and Web services. The *coordination layer* consists of a *composition connector* that mediates the interactions between computational layer and the application layer. The top interface of the composition connector, or *basic service interface*, is the sum of all the participant service interfaces. The composition connector may impose business rules to the basic service interface providing new composite services. The bottom interface of the composition connector, or *composite service interface*, is the sum of the basic service interface and the new composite services provided by the composition connector. The composite service interface also adds fault tolerance to its new services. The *application layer* contains the components that implement the application logic (*client components*) and that may use the composite service interface.

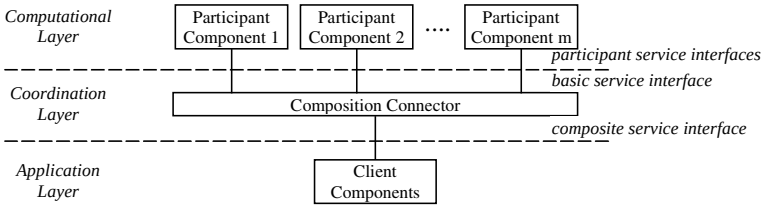



**Fig. 1.** Software Architecture using C2 style

The composition connector is a C2 connector built from an *interceptor component* and a set of *composition contract components* (Figure 2). A *composition contract* defines a set of related composite services. In this context, a composite service specifies

an action that may be required to add new business rules upon a service. The composite service can extends a single basic service or it can compose a more sophisticated service, which defines a coordinated action of two or more participant components. The composition contract components can be organized in various *contract layers* that are connected by specialised C2 connectors. Figure 2 shows a composition connector with two contract layers. The composition contract components are responsible for: (i) to provide new composite services; (ii) to impose the business rules upon basic and composite services; and (iii) to implement fault-tolerance for basic and composite services. This is done by means of the fault-tolerant composition of one or more basic services. The composition contract components of a contract layer can use composite services provided by contract components located at upper contract layers, allowing action nesting. The interceptor component is responsible for monitoring the events flow at the basic service interface and activating the composition contract components when needed.

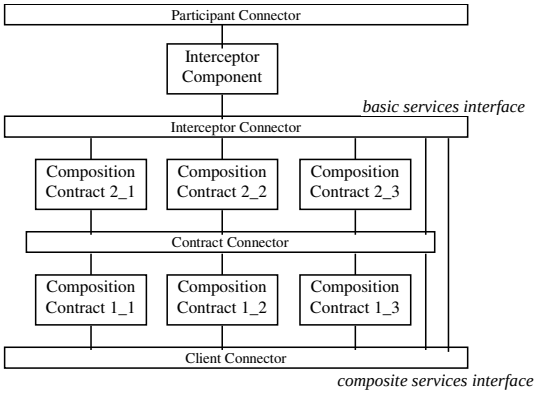



**Fig. 2.** Basic Structure of the Composition Connector

### 4.1 The Composition Contract Component

A composition contract component implements a composition contract as a CA Action relaxed in its transactional requirements over the participant components to guarantee the atomicity of the composite services. A composition contract is activated by a notification of an event associated with a *contract trigger*. This notification is sent by the *interceptor component*. The activation of a composition contract implies in the implicit invocation of an associated composite service. The invocation of a composite service normally results in one or more service requests accordingly with the coordination rules defined by the composition contract. The service requests can be concurrently executed. Moreover, they may activate composition contract components included in upper contract layers, creating nested CA Actions.

When a composite service completes successfully it ends either with a normal notification or, if its result is degraded, an exceptional notification. If an exception is raised by a requested service, the composition contract component collects the responses from the services and activates an *exception handler component*. The excep-

tion handler component resolves the raised exceptions and coordinates the appropriate recovery actions that should be taken by the participant components. If the exceptional condition cannot be masked, the composition contract component reacts with: (i) an *abort notification*, if the participants are recovered successfully, or (ii) a *failure notification*, if one or more participants are left in an inconsistent state.

The composition contract definition is shown in Figure 3a. The attributes clause specifies the configuration parameters an instance of a composition contract. The coordination clause specifies a list of interactions which defines how the participant components cooperate to perform a particular composite service (Figure 3b). The condition after the when clause specifies the *contract trigger* for this interaction. The do clause specifies a set of actions to be concurrently executed by the participant components. The raises clause specifies the types of exceptions that may be raised during the execution of the interaction.

```
contract <contract_name> {
  attributes <list_of_attributes>
  coordination <list_of_interactions>
end contract
```

(a) The contract definition

```
<interaction name>:
  when (<condition>)
  do <set_of_actions>
  raises <list_of_exceptions>
```

(b) The interaction definition

**Fig. 3.** The composition contract definition

### 4.2 The Interceptor Component

The *interceptor component* acts like a proxy object intercepting the messages sent to the participant components. It is responsible for notifying the composition contract components that a *contract trigger* was enabled. Currently, we consider that only service requests enable *contract triggers*. These notification messages contain the service request and the parameters of the composition contract being activated.

### 4.3 The Exception Handler Component

In our model, an architectural component can be a participant component, an interceptor component or a composition contract component. It is composed of two subcomponents: a NormalActivity component and an ExceptionHandler component (Figure 4). The NormalActivity component implements the normal behaviour of an architectural component, when no exceptions occur. The ExceptionHandler component is responsible for: (i) handling exceptions raised by its associated NormalActivity component; and (ii) providing *handler services*.

More specifically, the ExceptionHandler component of a participant component implements *handler services* to undo operations that affect the participant's state. The ExceptionHandler component of the interceptor component can handle system configuration errors. The ExceptionHandler component of a composition contract component coordinates the activation of *handler services* of the participant components. The ExceptionHandler component implements the exceptional contract defined for the composition contract (Figure 5). The ExceptionHandler component of a composition contract tries to mask an exception and return the control flow to the NormalActivity component

using a forward error recovery strategy. If this strategy fails the ExceptionHandler component starts backward error recovery executing compensation action to undo undesirable effects over the participant components, when possible

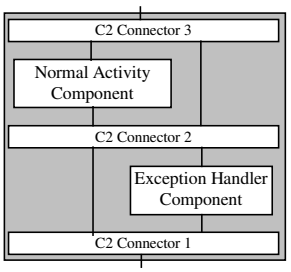



**Fig. 4.** Architectural Component

```
exceptional contract <contract_name> {
  attributes <list_of_attributes>
  coordination <list_of_handlers>
end contract
```

**Fig. 5.** Exception Handler Component Definition

## 5 Case Study

This case study illustrates the behavior of a fault-tolerant banking account system applying the software architecture described in this paper.This case study integrates two autonomous components: the Checking Account component and the Savings Account component, which wrap existing (off-the-shelf) components providing operations to withdraw and to get the current balance.

The system-specific business rules are enforced through a single composition contract that is the FlexibleContract. The FlexibleContract tries to avoid overdrafts in the CheckingAccoung using funds from an associated SavingsAccount, when needed.

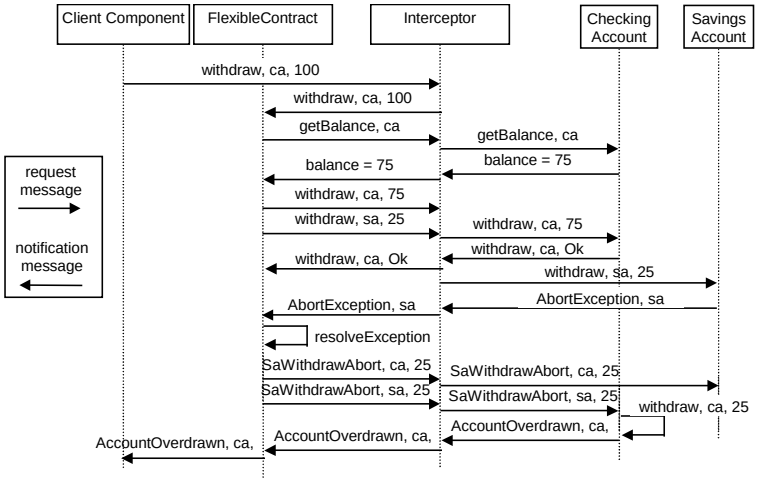



**Fig. 6.** Sequence diagram for the scenario

The sequence diagram in Figure 6 shows the processing of a request to withdraw an amount of $100 from a checking account "ca" which is associated with a savings account "sa" by means of a flexible contract. Initially, the FlexibleContract component checks the balance of "ca", which in this example is only $75. So, the FlexibleContract tries to satisfy the original request by means of two concurrent withdrawals: $75 from "ca" and $25 from "sa". In this example the SavingsAccount component refuses this second withdrawal and raises an AbortException. The FlexibleContract resolves the exceptions and sends a SaOverdrawAborted exception to both SavingsAccount and CheckingAccount components. The SavingsAccount ignores this exception. The CheckingAccount handles this exception with a self-invocation of a $25 withdrawal. This recovery action leaves "ca" overdrawn. The Client component is notified of this degraded result by means of an AccountOverdrawn exception that is propagated by the FlexibleContract component.

## 6 Conclusions and Related Work

In this work, we propose an architectural solution for the development of dependable software systems out of concurrent autonomous component-systems. This solution favours the adaptability, extensibility and reliability attributes of the resulting system. The concepts of coordination contracts and CA Actions were adapted to a service-oriented approach applied to the system's software architecture.

The WSCA (Web Service Composition Action) concept also exploits the concept of CA Actions to enable the dependable composition of Web Services [7]. The primary difference between our work and the WCSA concept is that our approach also includes concerns about coordination contract, which improves the adaptability, and an architectural design to be applied to a more general class of service-based systems, not only restricted to Web Services.

Similarly to our approach, Pires [8] proposes an architectural solution for providing reliable Web services compositions using a layered architectural style. However, this work only provides backward error recovery, not considering concurrent exception handling.

## Acknowledgments

This work was supported by FAPESP/Brazil under grant no. 01/06064-3 for Ricardo Silva, and by CNPq/Brazil under grant no. 351592/97-0 for Cecília Rubira. Paulo Guerra is partially supported by CAPES/Brazil

## References

1. Knight, J.C. SafetyCritical Systems: Challenges and Directions (summary of state-of-the-art presentation). International Conference on Software Engineering, Orlando, FL (May 2002).
2. Kreger, H. Web Services Conceptual Architecture. IBM Software Group. May 2001.

3. J. Xu, B. Randell, A. Romanovsky, C. Rubira, R. Stroud, Z. Wu. Fault Tolerance in Concurrent Object-Oriented Software through Co-ordinated Error Recovery. Proc. 25 th FTCS, pp 499-508, Pasadena, USA, 1995.
4. R. N. Taylor, N. Medvidovic et al. A component- and message-based architectural style for GUI software. In Proc. of the IEEE TSE, 22(6):390-406, June 1996.
5. M. Shaw and P. Clements. A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems. In Proc. of the COMPSAC'97, Washington DC, USA, August 1997.
6. L.F.Andrade, J.L.Fiadeiro et al. Coordination Patterns for Component-Based Systems. In Proc. of the SBPL'2001, pp. B29-B39, Curitiba,PR, Brazil, May 2001.
7. F. Tartanoglu, V. Issarny, A. Romanovsky & N. Levy. Dependability in the Web Services Architecture. In Architecting Dependable Systems. LNCS 2677. June 2003.
8. P. F. Pires at al. Building Reliable Web Services Compositions.In Proc. of the NET.Object Days Conference (WS-RDS'02), pp 551-562, Erfurt, Germany, October 2002.

# Error Recovery for a Boiler System with OTS PID Controller

Tom Anderson, Mei Feng, Steve Riddle, Alexander Romanovsky

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

**Abstract.** We have previously presented initial results of a case study which illustrated an approach to engineering protective wrappers as a means of detecting errors or unwanted behaviour in systems employing an OTS (Off-The-Shelf) item. The case study used a Simulink model of a steam boiler system together with an OTS PID (Proportional, Integral and Derivative) controller. The protective wrappers are developed for the model of the system in such a way that they allow detection and tolerance of typical errors caused by unavailability of signals, violations of range limitations, and oscillations. In this paper we extend the case study to demonstrate how forward error recovery based on exception handling can be systematically incorporated at the level of the protective wrappers.

## 1 Introduction

Although integration of Off-The-Shelf (OTS) components into systems with high dependability requirements (including those that are safety-critical) is becoming a viable option for system developers, care must be taken to avoid a deterioration in overall system dependability. OTS components are often of a lower quality than bespoke components, may not have been specifically intended for the environment in which they are to be employed, and may be poorly documented. These factors all contribute to a higher risk of failure for complex systems employing OTS components.

It must be accepted both that OTS components will be employed in such systems, and that their use will be a source of failure in spite of all efforts to improve the quality of OTS components and of the system in which they are to be integrated. The solution we advocate is to employ specialised fault tolerance techniques for integration of OTS components into complex systems.

### 1.1 Protective Wrappers

In previous work [1,7] we illustrated an approach to the development of protective wrappers, a bespoke software module which intercepts all information going to and from an OTS item. This approach is developed further in this paper using the same case study.

Fault tolerance techniques have three main phases: error detection, error diagnosis and error recovery [5]. The first phase identifies an erroneous state; error diagnosis is then used to examine and assess the damaged area, to enable it to be replaced by an error-free state during error recovery. We have previously concentrated on detection and diagnosis, providing only limited recovery actions.

Component wrapping is an established technique used to intercept data and control flow between a component and its environment [6]. A protective wrapper may detect errors or suspicious activities, and initiate appropriate recovery when possible, and must be rigorously specified, developed and executed as a means of protecting OTS items against faults in the Rest Of the System (ROS), and the ROS against faults in OTS items. Sources of information for wrapper development include specification of the OTS item behaviour, known “erroneous” behaviour of the OTS item, and specification of the correct behaviour of the ROS with respect to the OTS item.

### 1.2 Case Study

The case study used in this paper concerns the development of a protective wrapper for an Off-The-Shelf PID (Proportional, Integral and Derivative) controller. This case study is intended to illustrate how the approach could be applied in practice, employing software models of the PID controller and the steam boiler system rather than conducting a potentially risky experiment in a real-world

environment. Use of such software models is an active area of research and development carried out by many leading control product companies (including Honeywell [8]), and we use a third-party model of a steam boiler in this case study. We believe that this decision adds credibility to our results. The model simulates a real controller and steam boiler system, enabling us to investigate the effect of wrapping with a representative model. In the course of our work we have extended the model by incorporating protective wrappers.

### 1.3 Roadmap

The remainder of this paper is organised as follows. In the following section we describe the simulation environment, the controller and the boiler models we are using, and our approach to monitoring the model variables. Section 3 discusses the requirements for a protective wrapper to be developed and outlines the causes of errors to be detected and tolerated at the level of the wrapper. The next three sections discuss design and implementation of the wrapper to detect, diagnose and select an appropriate recovery action for errors caused by unavailability of signals, violations of range limits, and signal oscillations. Section 7 concludes the paper by discussing the generic error recovery strategy and the possible effects of wrappers executing on the overall execution of the integrated system.

## 2 Simulation

### 2.1 Simulink

Simulink (Mathworks) [10] is one of the built-in tools in MATLAB, providing a platform for modelling, simulating, and analysing dynamical systems. It supports linear and nonlinear systems modelled in continuous time and sampled time, as well as a hybrid of the two. Systems can also be multi-rate, i.e., have different parts that are sampled or updated at different rates. Simulink contains a comprehensive block library of sinks, sources, linear and nonlinear components, and connectors to allow modelling of very sophisticated systems. Models can also be developed through self-defined blocks by means of the *S-functions* feature of Simulink or by invoking MATLAB functions. After a model has been defined, it can be simulated and, using scopes and other display blocks, simulation results can be displayed while the simulation is running.

Simulink provides a practical and safe platform for simulating the boiler system and its PID control system, for detecting operational errors when boiler and control system interact, and for developing and implementing a protective wrapper dealing with such errors.

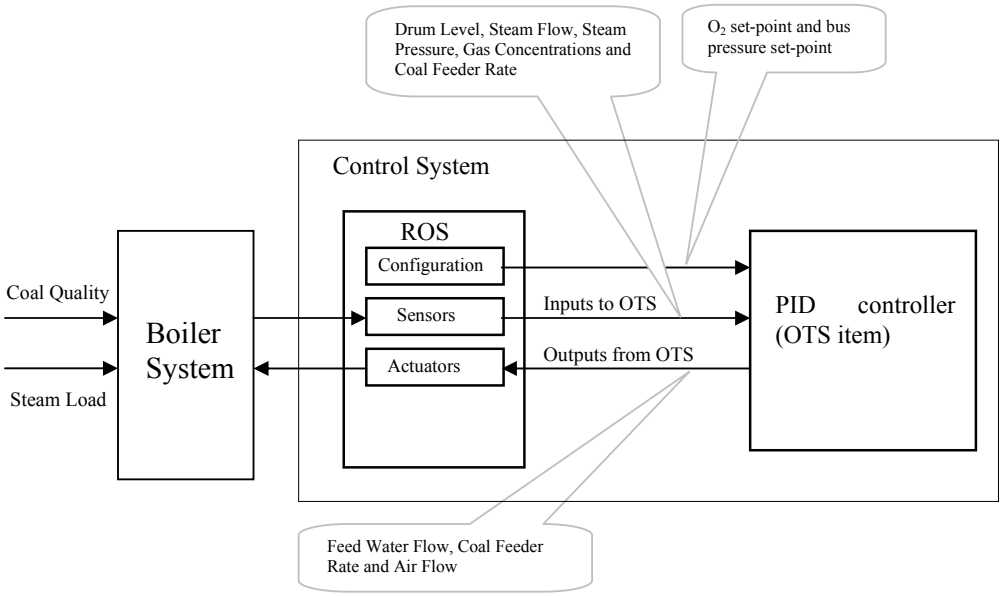



**Fig. 1.** Boiler System and Control System (including the PID Controller)

### 2.2 The Structure of the Model

The abstract structure of the system we are modelling is shown in Fig. 1. The overall system has two principal components: the boiler system and the control system. In turn, the control system comprises a PID controller (the OTS item), and the ROS which is simply the remainder of the control system.

The ROS consists of :

- the boiler sensors. These are "smart" sensors which monitor variables providing input to the PID controller: Drum Level, Steam Flow, Steam Pressure, Gas Concentrations and Coal Feeder Rate;
- actuators. These devices control a heating burner which can be ON/OFF, and adjust inlet/outlet valves in response to outputs from the PID controller: Feed Water Flow, Coal Feeder Rate and Air Flow;
- configuration settings. These are the "set-points" for the system: oxygen and bus pressure, which must be set up in advance by the operators.

Smart sensors and actuators interact with the PID controller through a standard protocol. Simulink output blocks can be introduced into the model in such a way that the variables of the MATLAB working space can be controlled as necessary. Working with the Simulink model we were able to perform repeatable experiments by manipulating any of the changeable variables and the connections between system components so as to produce and analyse a range of possible errors that would be reasonably typical for the simulated system.

### 2.3 The Simulink Model

The Simulink model (shown in Fig. 2) actually represents the OTS item as three separate PID controllers that deal with the feed water flow, the coal feeder rate and the air flow. These controllers output three eponymous variables: Feed Water Flow (F_wf), Coal Feeder Rate (C_fr) and Air Flow (Air_f); these three variables, together with two external variables (Coal Quality and Steam Load) constitute the parameters which determine the behaviour of the boiler system. There are also several internal variables generated by the smart sensors; some of these, together with the configuration set-points, provide the inputs to the PID controllers. Table 1 lists all of the variables used in the model.

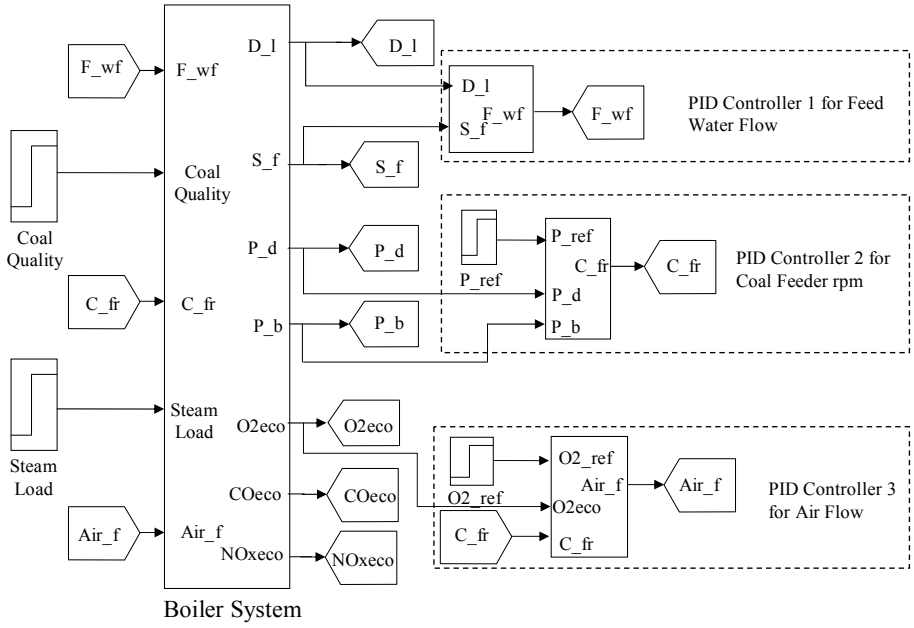



**Fig. 2.** Simulink Model of the Boiler System with PID Controllers

### 2.4 Variable Monitoring

Simulink scopes and other display blocks enable us to develop modelling components that observe the intermediate results while the simulation is running. In our experiments we can monitor and display a total of 15 variables, comprising all of the variables listed in Table 1 (except for the two set-points), plus three internal variables which represent two internal air flows and one internal steam flow. The simulation time for all of our experiments is set to 12000 steps. Some monitoring results are presented in Fig. 3. In particular, this chart demonstrates the behaviour of the three PID outputs and two external inputs of the boiler system when at step 2000 the steam load is increased, and at step 5000 the coal quality changes: in both these scenarios the boiler system returns to steady operation reasonably soon.

**Table 1.** Variables used in the model

| Variable | Representation | Variable | Representation |
|---|---|---|---|
| Coal Quality | Coal quality, ton per hour | D_l | Drum level |
| Steam Load | Steam Load, fraction of pure combustibles | S_f | Steam flow |
| F_wf | Feed water flow | P_d | Steam pressure / drum |
| C_fr | Coal feeder rate | P_b | Steam pressure / bus |
| Air_f | Air flow (controlled air) | O2eco | O2 concentration at economizer |
| P_ref | Bus pressure set-point | COeco | CO concentration at economizer |
| O2_ref | O2 set-point | NOxeco | NOx concentration at economizer |

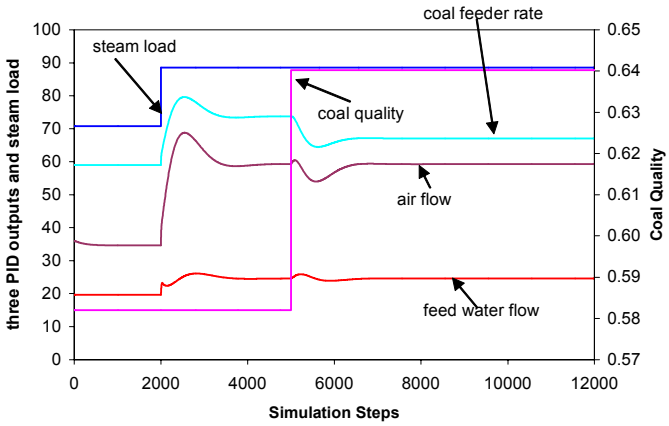



**Fig. 3.** Normal Performance of the Boiler System with PID Controllers

### 2.5 Properties of the Boiler System and the PID Controllers

In this section we summarise the information which we collected to guide us in developing the protective wrappers. The basic boiler specification provides information on steam flow, bus pressure, output temperature and coal calorific value. As the OTS item (the PID controller(s)) is treated as a black box, any information about its properties must be deduced from the interface or from relevant sources where available. In an ideal world the system designer would have a complete and correct specification of the boiler system, the PID controller and the ROS. Unfortunately, we only had access to limited information about the boiler system and the ROS (which is typical for many practical

situations). From an investigation of the boiler model and information acquired from all available sources, we have formulated the following description.

Information from the documentation available to us is as follows:

- Output temperature 540 deg C
- Coal calorific value 16-18 MJ/kg
- Steam load 50-125 ton/hour
- Coal quality is measured as a fraction of pure combustibles (where pure = 1; actual value about 0.55-0.7)
- Three controlled outputs (F_wf, C_fr, Air_f) are given as a percentage

Information obtained by analysing the interface and by investigating the simulated model:

- Set-point of bus pressure ranges from 0 to 20 (usual value about 9.4)
- Set-point of O2 concentration at economiser ranges from 0 to 0.1 (usual value about 0.03)
- Internal variables input to PID controllers:
  - Drum level: output value between -1 and +1 (usual value close to 0)
  - Steam Flow: 0 to 125
  - Bus pressure: 0 to 20
  - O2 concentration at economiser: 0 to 0.5

## 3 Requirements for a Protective Wrapper

In the previous section we presented an outline characterisation of the boiler system, as deduced from the model and other sources. In the following sections we consider the errors which could arise from integrating an OTS PID controller in the system, in order to derive requirements for a protective wrapper. We make the following assumptions:

- The value of each variable can be checked instantaneously through microprocessors. In particular, we assume that the values of input and output variables of the PID controller are available instantaneously. This (highly) simplifying assumption enables us to illustrate the method for protective wrapper development without regard to issues relating to response times.
- The wrapper program can be inserted into the control system, either by a partial hardware implementation which intercepts the physical connections, or purely in software. There are, of course, significant issues involved in deciding on the implementation of a protective wrapper, but we do not address these in this paper.

In order to clarify the requirements for a protective wrapper, it is necessary to form a view of what the PID controller and the ROS should, and should not, do at the interface between them. This view can be formulated as a collection of *Acceptable Behaviour Constraints (ABCs)* [7] defined from the perspective of the systems integrator. Once defined, these ABCs can be thought of as contracts [11] which a system designer could use as the basis for defining a protective wrapper, which can then employ conventional mechanisms for error detection, containment and recovery [2].

## 4 Safe Boiler Operation

Many aspects of the operation of the boiler and control system, such as the flow of gases, fuelling, pressures and levels, could lead to a failure of some type. However, some features are much more significant in terms of safety; in a steam boiler, the drum level is a key parameter. This parameter represents the quantity of water in the boiler more accurately than a direct measurement of the water level, due to changes in mass caused by differences in temperature. By monitoring and controlling the drum level we can maximize steam quality and maintain the proper water quantity to prevent damage to the boiler. Too low a level could expose the water tubes to heat stress and damage; too high a level could allow water to go over the steam header, exposing the steam turbines to corrosion and damage [12, 13]. Steam pressures on the drum and the bus are the two other key variables, since they indicate the balance between the supply and demand for steam. The consequences of excessively high steam pressures are obvious and explosive. Thus, any deviation from normal values of steam pressure and drum level must be corrected immediately, whereas abnormal values of the other variables can be tolerated for a time period (which must be defined by the system designer).

We classify the detectable variables into two loops, the control loop and the safety loop, which operate as follows:

- an alarm from the safety loop will shut the system down;
- alarms from the control loop are processed on-line and either resolved or, if this is unsuccessful, the safety loop is triggered.

The wrapper envelopes the PID controllers as shown in Fig. 4: it monitors the values of variables which go into and come out from the PID controllers. Three variables – the drum level and the steam pressures on the drum and bus – are classed as belonging to the safety loop, and all other variables (PID outputs to the boiler (via the ROS), set-points and other input variables) belong to the control loop.

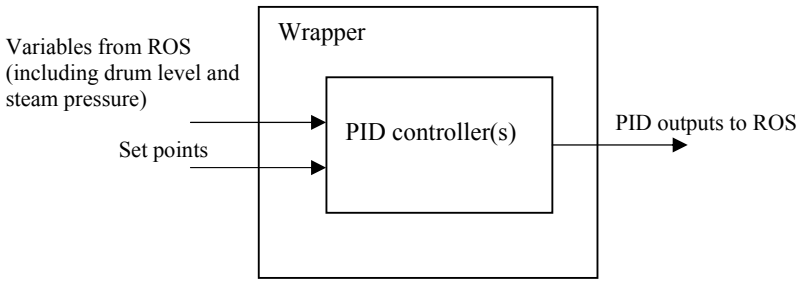



**Fig. 4.** Variable categories around the PID controller(s)

## 5 Error Recovery

Error recovery transforms a system state that contains errors into an error free state. The transformation typically takes the form of either *backward* or *forward* error recovery [2]. Backward error recovery returns the system to a previous (assumed to be correct) state; typically, the techniques used are application-independent and often operate transparently for the application (e.g. atomic transactions and checkpoints). Forward error recovery aims to move the system into a correct state using knowledge about the current erroneous state; this recovery is application-specific by its nature. The most general framework for achieving forward recovery is *exception handling* [3]. It is not difficult to see that backward error recovery is not generally applicable in dealing with OTS items [4].

Protective wrappers offer a structured approach for incorporating fault tolerance measures in systems with OTS items. In previous work [1] we demonstrated how error detection can be developed in a wrapper by cyclically checking for each type of possible error identified during the analysis phase. We characterise these errors into three distinguishable types: (a) signal not available, (b) signal violating specifications, and (c) unacceptable signal oscillations. When the wrapper detects an erroneous situation it immediately initiates recovery action by classifying the error and invoking a corresponding exception handler. Three possible recovery actions are suggested here.

- Handler1:
  - Reset the signal to a standard normal value and send an alert to the operators.
- Handler2:
  - Wait $\Delta t$, if error resolved, take no action;
  - Otherwise, send an alarm to the operators and wait $\Delta T$, if error resolved, take no action;
  - Otherwise, invoke handler 3.
- Handler3:
  - Shutdown the system and send an alarm to the operators.

In Handler2, the delay times $\Delta t$ and $\Delta T$ would be determined by the wrapper designer after consulting the system specification. In the Simulink model we took $\Delta t$ as 500 steps and $\Delta T$ as 1500 steps, representing reasonable values for a genuine industrial application.

Analysis of the error types for different signals then enabled us to define a recovery strategy, which is discussed in the following subsection, and then illustrated in Fig. 5.

### 5.1 Recovery Strategy

In the case of an erroneous situation detected for a set-point value, whether it is missing, out of specification or oscillating is of secondary interest. The wrapper is aware of the appropriate range of set-point values, and Handler1 provides appropriate recovery by forcing the input to a suitable value, and alerting the operators (since the mistake could be theirs, or an internal corruption).
The situation is rather different when a PID output value is detected as being erroneous, but the same response can be made; either by adopting standard operating output values, or by storing a recent history and using a smoothed average, the wrapper can apply Handler1 to impose a valid output signal which should result in stable, though suboptimal, performance from the boiler system. It would be possible to differentiate between the three error types in terms of determining the signal value to be imposed, but we have not exploited this in our simple demonstrator.

Now consider inputs to the PID controllers which are not set-points, and are not in the safety loop. When one of these is detected as erroneous there is little point in feeding a fixed value to the PID, since this will not reflect the actual conditions monitored by the ROS. However, given that there is no immediate safety concern, the optimistic strategy of "wait and see" may be successful; indeed for a short interval it may not be appropriate even to alert the operators, since the phenomenon may be completely transient in nature (it would, of course, still be logged for an off-line report). If the problem persists an alarm report to the operators may enable them to cure the problem, but if not an eventual shut-down is inevitable. So the wrapper can deploy Handler2; again, although differentiation of the response in line with error type is possible, we have not exploited this option.

Lastly, consider the variables in the safety loop: drum level, drum and bus steam pressures. Detecting an erroneous condition on one (or more) of these variables implies a risk to safety, and the natural response is to shut the boiler system down, despite the economic consequences. Handler3 provides this response, but to illustrate error category differentiation in our model we invoke Handler2 in the particular case of an oscillating value. The justification is that although a missing signal value or an out of specification value indicates a clear and present risk of accident, an oscillating value *within specification* does not necessarily pose the same immediate threat – the oscillations may die out or an operator response could stabilise the situation. If the oscillations persist then Handler3 will still be brought into play. Of course, in a real boiler system this strategy would only be acceptable if justified by a safety case.

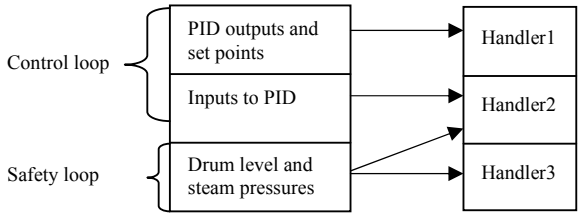



**Fig. 5.** Recovery action implemented by exception handlers

### 5.2 Exception Handling

When any of the errors above is detected an exception is raised. Error diagnosis is performed to select the appropriate handler, depending on which variable caused the exception to be raised. The diagnosis is straightforward for variables in the control loop: in the safety loop, we differentiate between error categories.

Errors detected by the protective wrapper can be caused by malfunctioning of the OTS item, by faults arising in the ROS, or by misinterpretation between the OTS item and the ROS (Fig. 1). In the PID case study considered here, the exception handlers implemented in the protective wrapper always act at the level of the integrated system, which constitutes the exception handling context [4]; they can send an alert or alarm signal to the operator, replace an erroneous value with an alternative "normal" value, await a natural rectification, or (when safety requires it)) shut the system down.

Clear separation of the normal and abnormal system behaviour by employing exception handling in the wrapper facilitates the integration of the OTS PID into the composite system [4].

## 6 Implementation of the Recovery Actions in MATLAB

This section illustrates the operation of recovery when an error occurs, with the wrapper implemented in the MATLAB model. The example presented below applies error recovery by invoking Handler1. The error was introduced by artificially simulating an incorrectly valued pressure set-point. At step 5000 the operator "accidentally" recalibrates the pressure set-point to be 94 instead of the normal value of 9.4.

Fig. 6 shows the boiler system behaviour with no wrapper protection; note that the bus steam pressure is superimposed over drum steam pressure, so only one pressure variable is actually displayed.

The first chart of Fig. 6 shows that the faulty pressure set-point results in a huge drop in the drum level followed by a peak at too high a value before returning to a level close to normal. Similarly, the three PID outputs shown in the second chart jump up beyond their specified levels after step 5000 but return to normal performance by step 7000. Very much more serious is the steam pressure, which the first chart shows rising and remaining at an excessive level.

Fig. 7 shows the behaviour of the boiler system with wrapper protection active. The wrapper detects the incorrect value of the set-point and invokes Handler1. Consequently the set-point is very quickly corrected, and the impact on system behaviour and on the other variables is greatly reduced. From the two charts in Fig. 7 we see that although there are some alterations to the variables when the error occurs, the changes are actually quite minor, and lie within the range of the specifications for the system.

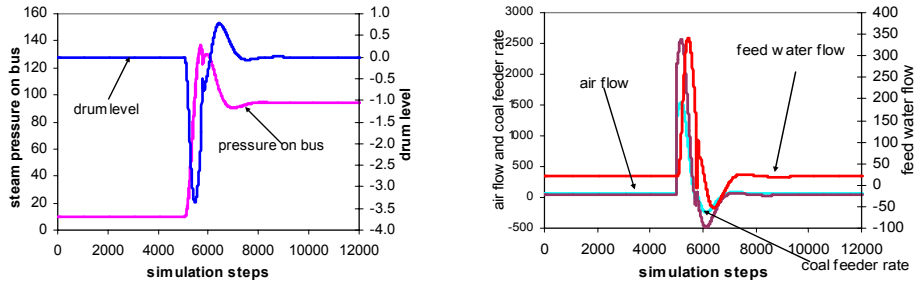



**Fig. 6.** Erroneous pressure set-point – boiler system with no wrapper. Drum level and feed water flow are outside permitted ranges: steam pressure rises to an excessive level.

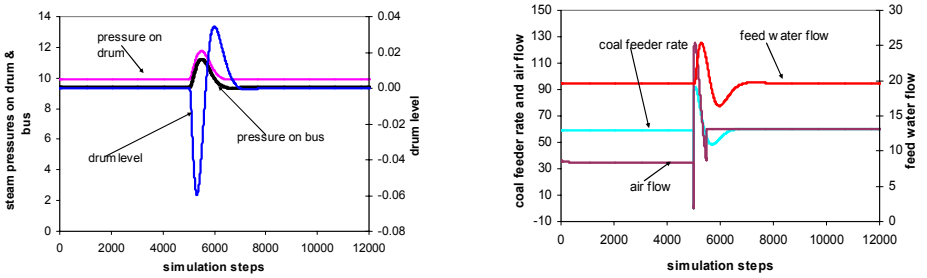



**Fig. 7.** Erroneous pressure set-point – error recovery by Handler1. Variables are corrected to be within permitted ranges

## 7 Discussion and Conclusion

### 7.1 Generic Error Recovery Strategy

Since an OTS item that has been integrated into a system is treated as a black box, only the inputs to and outputs from the OTS are available to be monitored for error detection. The inputs in a control system can be partitioned into two groups, where the first group consists of configuration variables (often under operator control as set-points), while the second group provides dynamic status information from the system under control. A wrapper providing protection in the system can monitor these two groups of inputs, and the outputs, and can attempt to distinguish different categories of erroneous behaviour. As three simple base categories we proposed: missing value, out of range value, and oscillating value.

In response to a detected error situation, over these groups of variables and categories of error, the protective wrapper can apply an exception handling framework to attempt to recover from the error. Simplistic generic recovery strategies that we have considered are: do nothing, alert the human operators, change variables to normal values, and stop the system. In our boiler system example we decided that erroneous outputs from the OTS could be over-ridden by the wrapper; in effect, the wrapper will take over the role of the OTS in erroneous situations, but can only provide a standard set of normative outputs. In the same way, the wrapper can over-ride configuration variable inputs to the OTS, particularly when these are input set-points. Our Handler1 is used to over-ride erroneous values.

We need to analyse in more detail the other input variables delivered to the OTS by the ROS, since forcing a change here could only have a very indirect influence on the variable itself, via the OTS and its outputs back to the controlled system. One important aspect is with respect to safety and we used this to apply a hierarchy of recovery. For variables with a direct impact on safety we "recover" by shutting the system down (Handler3); for other variables, not in the safety loop, we first wait to see if the error is transient, then to see if human intervention will achieve recovery, and if that too is unsuccessful then the system must be shut down in any case (Handler2).

Wrapper design in any specific case would proceed by analysis of the state space of variables, the errors that could be detected, the damage assessment that could be conducted, and the recovery strategies that could be devised, bearing in mind the implications on system operations from both a mission (economic) and a safety perspective.

### 7.2 Scope of the Wrapper

The essential characteristic of a protective wrapper is that *all* inputs to and outputs from the wrapped component are accessible to and modifiable by the wrapper. It might be argued that no other system variables should be accessible to the wrapper, since otherwise the intuitive image of "wrapping the component" would be distorted. We feel this is unnecessarily restrictive. If the system designer believes that improved performance of the wrapper can be achieved by utilising information from elsewhere in the system this should not be prohibited by an artificial limitation. When the wrapped component is an OTS item it may be very unlikely that the wrapper could make any effective use of internal state information within the OTS component, but valuable insight may perhaps be gleaned from variables in the ROS that are not visible to the component. An example of this is present in our case study. The drum steam pressure is not actually utilised by the PID controller (although it is made available), so it is debateable whether or not it constitutes an input. We gave the wrapper access to this variable without hesitation, since we suspect that any practising engineer would do the same.

### 7.3 Complexity of the Wrapper

A wrapper inserted into a system as a protective component performs an important role in improving the reliability of the integrated system. In discharging this role it is clearly essential that the wrapper does not itself contribute to an increase in failing behaviour from the overall system. Ideally, the wrapper should introduce no faulty behaviour itself, and should capture and rectify all faulty behaviour it encounters. Perhaps the best general guidance here is that the most reliable designs will usually be those that are simplest. The OTS PID controllers may, of necessity, involve highly complex algorithms to achieve the optimised boiler performance that is their goal. But in designing an effective wrapper we

are likely to find that a simple and effective recovery strategy will outperform something overly sophisticated. First, a simple design is more likely to be implemented correctly, and second, a more complex strategy may have unforeseen interactions with the control environment and these could detract from effectiveness.

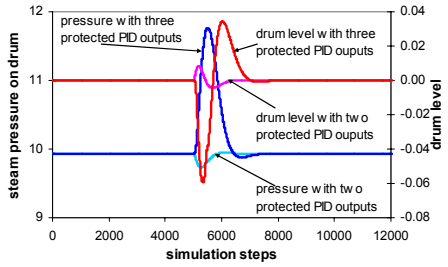



**Fig. 8**. Comparison of effects on performance of boiler system with two kinds of wrapper designs

Fig. 8 shows a comparison of the impact on two of the variables in the safety loop for the boiler system, with two different wrapper designs responding to the situation described in section 6 (the pressure set-point changed from 9.4 to 94 at step 5000). One wrapper applies recovery by using Handler1 to reset all three PID outputs to standard values, while the other wrapper only resets the feed water flow and air flow, leaving the coal feeder rate at its computed value. We can see from Fig. 8 that the more simplistic strategy of only changing two of the variables leads to a swifter and more stable recovery.

This paper has summarised our recent work in the development of protective wrappers as a structured approach to providing error detection and recovery in systems utilising OTS items. This approach embodies error classification and corresponding recovery strategies implemented within an exception handling framework, building on the structure and error detection issues considered in earlier papers [1,7].

**Acknowledgements**

This work is supported by the EPSRC/UK project DOTS: *Diversity with Off-The-Shelf Components*. (http://www.csr.ncl.ac.uk/dots). This work has benefit from interaction with colleagues at City University within the DOTS project.

**References**

1. T. Anderson, M. Feng, S. Riddle, A. Romanovsky. Protective Wrapper Development: A Case Study. *2nd International Conference on COTS-Based Software Systems (ICCBSS 2003)*. Ottawa, Canada, February, 2003. pp. 1-14
2. P. A. Lee, T. Anderson, *Fault Tolerance: Principles and Practice*, Wien - New York, Springer-Verlag, 1991.
3. F. Cristian. Exception Handling and Tolerance of Software Faults. In M. R. Lyu (Ed). *Software Fault Tolerance*. John Wiley and Sons, 1995, pp. 81-108
4. A. Romanovsky. Exception Handling in Component-Based System Development. *25th Int. Computer Software and Application Conference (COMPSAC 2001)*, Chicago, IL, October, 2001. pp. 580-586.
5. J.-C. Laprie. "Dependable Computing: Concepts, Limits, Challenges". *Special Issue of the 25th International Symposium On Fault-Tolerant Computing*. IEEE Computer Society Press. Pasadena, CA. June 1995. pp. 42-54
6. J. Voas. Certifying Off-The-Shelf Software Components. *IEEE Computer*, 31(6), 1998, pp. 53-59.
7. P. Popov, S. Riddle, A. Romanovsky, L. Strigini. On Systematic Design of Protectors for Employing OTS Items. In *Proc. of the 27th Euromicro conference*. Warsaw, Poland, September 2001 IEEE CS. pp. 22-29.
8. V. Havlena, Development of ACC Controller with MATLAB/SIMULINK. *MATLAB '99*. Praha: VSCHT - Ustav fyziky a merici techniky, 1999, pp. 52-59.
9. J-R. Abrial, E. Börger, H. Langmaack. *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*. LNCS 1165, Springer Verlag, October 1996.
10. Mathworks, Using Simulink: reference guide, http://www.mathworks.com
11. B. Meyer. Programming by Contract. In D. Mandrioli, B. Meyer (eds.), *Advances in Object-Oriented Software Engineering*. Prentice Hall, 1992.
12. Siemens, Boiler Control Overview, http://www.procidia.com
13. Minnesota Department of Labor and Industry, Your guide to safer boiler operation, http://www.doli.state.mn.us/pdf/guide2saferboiler.pdf.

# Exception handling in component-based systems : a first study

Frédéric Souchon* **, Christelle Urtado*, Sylvain Vauttier*, Christophe Dony **

* LGI2P - Ecole des Mines d'Alès - Parc scientifique G. Besse - 30 035 Nîmes - France - +33 4 66 38 70 00
{Frederic.Souchon, Christelle.Urtado, Sylvain.Vauttier}@site-eerie.ema.fr

** LIRMM - 161 rue Ada - 34 392 Montpellier - France - +33 4 67 41 85 33
dony@lirmm.fr

**Abstract** Reliability and fault-tolerance raise new issues in modern software architectures (such as component based architectures or multi-agent systems) as they aim at integrating separately developed software entities to build large-scale systems. In this paper, we study existing exception handling systems associated to the various component models that can be found in java-based platforms. This includes components having contract-based or event-based interaction schemes and synchronous or asynchronous communication schemes. We then focus on what we consider to be the most problematic issue: exception handling for asynchronously communicating components. We then present four qualities we think an EHS for a CBP should have and that we will consider as requirements for future work on designing an EHS that fulfill them.

## 1 Introduction

Modern ways of building applications require new means of software engineering: it is now easier to reuse existing pieces of software and integrate them to build complete applications. Component-based platforms (CBPs) [24,11,15] or multi-agents systems (MAS) [4] allow to build such applications with separately developed software entities. We believe that this reuse and integration process raises new reliability issues and that reliability becomes a more and more critical as systems get larger. In our previous work, we have studied exception handling in MASs and have proposed an exception handling system (EHS) dedicated to MASs [19]. In this paper, we address exception handling in CBPs and focus on what we consider to be the most problematic issue: exception handling for contract-based, asynchronously communicating components.

The remainder of this paper is organized as follows. Section 2 presents our study of exception handling in various existing CBPs. It introduces three categories of components and examines their particularities regarding exception handling. Section 3 focuses on a particular category of components (contract-based asynchronously communicating components) for which exception handling is particularly difficult and presents what we think to be the requirements that an EHS dedicated to such components should fulfill. Section 4 concludes and presents work in progress: how we are transposing and adapting our previous work on exception handling in MASs to provide a fully functional EHS for J2EE message-driven beans.

## 2 Exception handling in component-based platforms

Our study of exception handling is based on three kind of components that can be found in Java-based platforms (session or entity beans, message-driven beans and JavaBeans). They are well known, globally representative of the various kind of components available in existing operational platforms, and their open implementation makes it possible to freely experiment.

### 2.1 Component categories

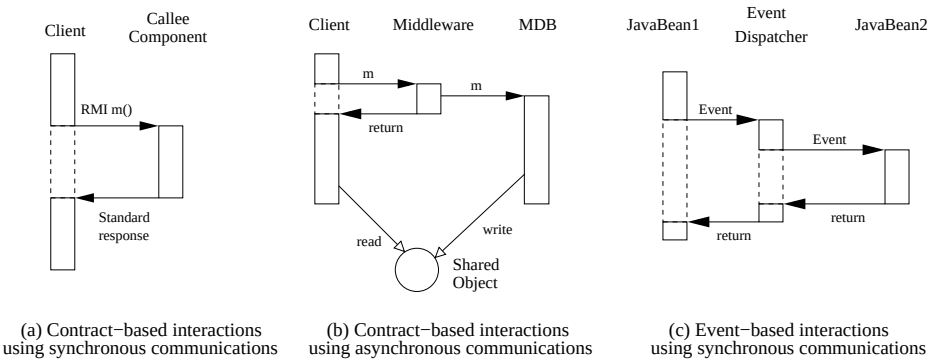
Components can be distinguished by (among other criteria non meaningful here) the way they interact and communicate.

**Interaction schemes:** main interaction modes are "Contract-based" and "Event-based".

- With Contract-based interactions (cf. Fig. 1a,b), components send requests to other components (acting as service providers) and expect answers in return as specified by component software contracts [13,12]. In this paper, we consider that requesting a **service** from a component triggers the execution of the corresponding **activity** in the concerned component.
- With Event-based interactions, components are more loosely coupled and interact via an event dispatcher that communicate events emitted by a component to those registered as its listeners. (cf. Fig. 1c). In this context, the emitting component does not know which components listen to its emitted events and does not expect any answer.

**Communication schemes:** components may communicate synchronously or asynchronously.

- With synchronous communications, service callers are blocked until callees reply. Only a single execution flow is executed at a given time: there is no concurrency (Fig. 1a,c).
- With asynchronous communications, request emission is non-blocking. Thus each service request triggers this creation of a new thread to run the corresponding activity (cf. Fig. 1b). This communication means is very flexible and suits particularly well applications in which QoS policies have to be implemented (eg. «provide a result once a certain QoS [6,26] level has been reached» or «providing the best result obtained before a specified delay»).




(a) Contract−based interactions using synchronous communications

(b) Contract−based interactions using asynchronous communications

(c) Event−based interactions using synchronous communications

**Figure 1.** Java components interaction and communication Schemes

Regarding this classification, we found three kinds of existing components:

- **CS components**: contract-based interactions with synchronous communications,
- **CA components**: contract-based interactions with asynchronous communications,
- **E components**: event-based interactions with synchronous communications.

### 2.2 Examples of Java components

Java Beans [23] are E components (cf. Fig. 1c) and are conceptually similar to Microsoft Active/X.

Other examples come with J2EE:

*Session* and *entity beans* are CS components which communicate via middlewares (RMI or CORBA). The middleware ensures packaging, transport and unpackaging of both method calls and results exchanged between remote components.

Finally, *message-driven beans (MDBs)* are CA components which communicate via messaging protocol called JMS (*Java Messaging Service*)[21,22]. A MDB which receives a JMS message executes the corresponding activity in a dedicated thread: this allows MDBs to concurrently process various activities. When a callee needs to return something to its caller, if the caller is not a component that can receive JMS messages, they have to use a shared object (cf. Fig. 1b) as a temporary repository. Otherwise, the callee can send a new JMS message to give the answer, thus implying the creation of a new thread.

JMS also allows to broadcast messages to a set of components thanks to the notion of *topic* (to be compared to *object groups* [14] which allow to gather a group of objects which can be seen and addressed as a unique entity). In JMS, a message sent to a topic is broadcasted to all MDBs which subscribed to the given topic but the sender is aware of neither the identity nor the number of receivers.

| | Contract-based interactions | Event-based execution interactions |
|---|---|---|
| Synchronous communications | J2EE (RMI, Corba) | JavaBeans, ActiveX |
| Asynchrnous communications | J2EE (JMS) | none identified |

**Figure 2.** Categories of components

### 2.3 Exception handling

This subsection describes how exceptions can today be signaled and handled for the three component models. Let us briefly recall that **exception signaling** [5], is a mechanism used to notify undesirable situations that hamper the standard execution of a program to continue. When an exception occurs, reliable software is able to react appropriately in order to continue its execution or, at least, to interrupt it properly while preserving data integrity as much as possible. An **exception handling system (EHS)** [3,5,10,13,27] offers control structures enabling developers to define program units (e.g. the source code of a procedure[10], a class definition [3]...) as protected by a set of **exception handlers** in order to capture exceptional situations that may be signaled in these protected regions. Each handler is defined to capture occurrences of a given exception type. The signal of an exception provokes the interruption of what is currently being executed and the search for an adequate handler. Handler search mechanisms are mostly based on Goodenough proposal [5] (eg. in C++ and Java) in which exceptions are propagated through the execution history. From a component point of view, such a mechanism would propagate exceptions signaled by component activities to their corresponding clients which would therefore be able to handle exceptions in the context of the failed service calls.

**Exception handling for CS components (eg. session beans):** their specification [20] and our experiments on the JOnAS platform [1] show that exception handling is achieved through standard Java control structures (*try/catch, throw...*). The middleware (RMI or CORBA) enables to transparently use server components as Java objects: the middleware abstracts distribution (cf. section 2.2) for standard execution as well as for exception handling as components use a proxy representation of remote components. This EHS seems to satisfy developers needs as it adopts the behavior of standard C++/Java languages EHSs which propagate exceptions through execution history. Thus, context-sensitive exception handling is possible as shown in figures 3 and 4a.

```
...
// A component invoking the buy method
through RMI
try {
  utx.begin(); // Starts a first transaction
  t1.buy(10); //request on the bean
  utx.commit();  //Commits the transaction
  }
catch (LimitedStockException exc) {
  int n =  exc.getMessage();
  println("Buying only" + n + "units");
  t1.buy(n);
  }
...
```

```
...
// Remote Business method implementation.
public void buy(int s) {
  if (stock>=s) {
    newtotal = newtotal + s;
    return;
    }
  else
    throw(new LimitedStockException(stock);)
}
...
```

**Figure 3.** Example of method invocation with RMI

**Exception handling for E components (eg. Javabeans):** If an event emitted by a component (*Javabean1*) leads to the signal of an exception in another component that cannot catch it, the exception is signaled to the event dispatcher (cf. Figure 4c). This event dispatcher can only generically handle the exception without propagating it to *Javabean1* (it just prints a stack trace) [23]. Thus, the exception cannot be propagated to the emitter's context. Such isolation of components concerning exception handling is coherent with the interaction scheme of these components: when notifying an event E components expect no response, either normal or exceptional (cf. subsection 2.1).

**Exception handling for CA components (eg. MDBs):** According to JMS specification [20] and to our experiments on the JOnAS platform, exception handling support in MDBs is limited: an error during a JMS message send can be notified to the sender but an exception signaled during the execution of the corresponding activity cannot be propagated to its caller. Indeed, an

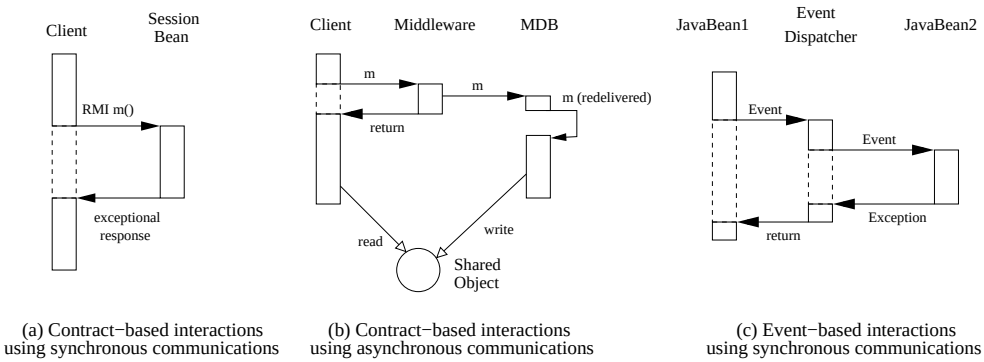



**Figure 4.** Communications in presence of exceptions

exception signaled in one the activity of a MDB without being handled locally is not propagated to the calling context. The only mechanism provided to allow the programmer to detect these situations is that the message that initiated the activity is redelivered (with a flag set to true) to the MDB in which it is executed (cf. Fig. 4b). Thus, to differentiate received messages from exception notifications, the developer has to test (for each message delivery) if the message is redelivered or not (cf. Figure 5). From our point of view, the handling of exceptions in the CA components we studied is too poor because:

1. Exceptions are not propagated through the execution history. It is therefore not possible to define context-sensitive treatments in handlers. Handler can only be generic and be used, for example, to display error messages (cf. Fig. 5). This issue will be developed in section 3.1.
2. There is no means to coordinate concurrent activities of components, and, thus, no exception coordination. For example, when an activity which requested services signals an exception, it should be possible to terminate the pending services it requested. This issue will be developed in section 3.2.
3. There is no means to concert exceptions that occur concurrently. This lack makes it impossible to correctly manage situations where multiple exception signals reveal a unique problem. It also hampers the programmer's capability to provide QoS policies (for example, he/she cannot distinguish under critical from critical exceptions). This issue will be developed in section 3.3.

```
public void onMessage(Message message) {

// Exceptional execution
  try {
  // The programmer tests if the message has been redelivered
  if (message.getJMSRedelivered()) {
    // The programmer handles the exception without information
    // about the execution context in which the exception has been signaled
    System.out.println("Error while handling" + message);
    return;
    }
  }
  catch(Exception ex) { System.err.println(ex.toString());}

//Standard execution
  try { ... }
  catch(HandledExceptionType ex) { System.err.println(ex.toString());}
}
```

**Figure 5.** Example of exception handling in a MDB

| | Contract-based interactions | Event-based execution interactions |
|---|---|---|
| Synchronous communications | Typical EHS : satisfying | Generic EHS: isolated components |
| Asynchrnous communications | dedicated EHS : unsatisfying | none identified |

**Figure 6.** The 3 components categories

## 3 Requirements for exception handling in CA components

As shown in the previous section (cf. Fig 6) CA components (such as MDBs) are the most problematic when exception handling is concerned because of asynchronous calls. This section aims at presenting the requirements concerning what we think should happen (and which extra concepts will have to be correctly managed) in situations in which an asynchronously invoked service cannot be fulfilled because of the occurrence of an exception.

### 3.1 Need for contextualization respect

When a called activity signals an exception it cannot handle, **its calling activity is the best place where to handle this exception properly**: it is the only place where the objective targeted by the corresponding service call is known and thus the best place to program what should be done in case of a defect. We thus believe that the software contract is to be respected in the case of exceptional responses as it is for normal ones in order to enable context-sensitive treatments of exceptions. Unfortunately, this behavior is not always adopted for CA components (cf. section 2.3) even if CBPs provide dedicated EHSs.

Alternative propositions[25,9] exist that do not address this problematic because of their centralized aspects. For example, [2] proposes a transposition of the *supervisor* model which its authors previously proposed in the framework of MASs [8,9]: it suggests the use of an EHS using *sentinel components* to detect and handle components failures. Propagating exceptions to such dedicated entities does not provide the capability to the write context-sensitive handlers and, thus, limits treatments in handlers to generic context-independent reactions (eg. error message display). Moreover, the centralization of the handling of exceptional events may affect QoS because it may cause bottlenecks which can slow down the whole system and decrease its reliability [19].

### 3.2 Need for coordination tools

Efficiently handling exceptions in systems that use asynchronous communications implies the definition of means to manage activities cooperation. [18,16] provides a classification of three types of concurrency and studies their impact on exception handling:

- Disjoint concurrency appears in systems that provide no coordination of concurrent activities. For example, there is no system level means to coordinate MDB activities when exceptions occur (no global activity is considered for such components).
- Competitive concurrency appears in systems that provide mechanisms to avoid inconsistencies caused by concurrent uses of system resources (generally thanks to lock-based mechanisms). Such mechanisms exist and can optionally be used with MDBs if *JTA (Java Transaction API)* or *JTS (Java Transaction Service)* is used to delimit distributed transactions.
- Cooperative concurrency is the kind of concurrency used by systems that provide some support to manage collaborations between active entities. For example, when an activity terminates, either normally or exceptionally, it must be possible to kill all non-terminated pending activities it initiated. [17] claims that such a cooperative concurrency management requires an execution model that enables collective activities to be explicitly represented. From our point of view, such a representation would make it possible **to associate handlers to the representants of such collective activities**. It would therefore be possible to globally manage the impact of the failure of either a single participant or a set of them.

To summarize, CBPs should provide support for cooperative concurrency in order to enable the coordination of components activities.

### 3.3 Need for exception concertation support

Once activity coordination is supported and integrates an exception handling mechanism, a means to collect exceptions occurring concurrently among participants of a collective activity must be provided. An activity which concurrently sends requests to a set of components and which receives one or more exceptional responses from them:

- must not react immediately when under-critical exceptions (which do not hamper its standard execution) are signaled,
- must be able to take into account that exceptions received concurrently may have no pertinence individually but may be meaningful collectively (result from an unique problem or a poor QoS level).

Based on these ideas, [7] suggests the use of a concertation mechanism: when entities which participate to a collective activity concurrently signal exceptions to their caller, a *resolution function* considers the set of signaled exceptions in order to evaluate a unique exception (a *concerted exception*) which reflects the global state of the collective activity. This ***concerted exception* is used in place of the whole set of individual exceptions signaled by requested services** to search for handlers in the collective activity.

### 3.4 Need for a specific exception handling policy for broadcasted requests

In section 3.2, we noticed the need to coordinate collective concurrent activities to properly handle exceptions. In JMS (cf. section 2.2), such a quality can partially be reached by activity coordination and must be completed by a proper handling of broadcasted requests. It must therefore be possible, for the programmer, to associate handlers to topics. As topics are not components (they are implicit notions hidden in the J2EE container), we propose to embody them in a dedicated component to which programmers can associate handlers and where they can configure concertation. This component is used to broadcast received requests to its registered components (just like topics did) and has the additional capability **to aggregate standard and exceptional responses in an unique meaningful response** transmitted to the service caller.

### 3.5 Synthesis

Figure 7 synthesizes what are the capabilities and the needs, in terms of exception handling, of the three categories of components we studied in this paper. It focuses on the four qualities presented in sections 3.1, 3.2, 3.3 and 3.4. It shows that exception handling in CS and E components is adequate as the EHS capabilities match the qualities needed. On the contrary, the analysis of exception handling in CA components shows that work must be done to match EHS capabilities and programmers expectations.

| | CS components | | CA components | | E components | |
|---|---|---|---|---|---|---|
| **Quality** | **Exist ?** | **Needed ?** | **Exist ?** | **Needed ?** | **Exist ?** | **Needed ?** |
| Contextualisation respect | yes | yes | no | yes | no | no |
| Coordination support | yes | yes | partial: transactions | yes | no | no |
| Exceptions concertation support | no | no | no | yes | no | no |
| Support for collective requests | no | no | partial: forwarding | yes | partial: event dispatcher | no |

**Figure 7.** Potential improvements

## 4 Conclusion and further work

In this paper, we studied available categories of components and their EHSs. On the one hand, we believe that two out of three categories of components have adequate EHSs. On the other hand, we highlighted lacks in EHSs provided with the third components category (components which interact using a contract-based scheme and asynchronous communications).

In particular, we think that an EHS for such components must have these four qualities:

- Contextualization must be respected in order to enable the writing of context-sensitive handlers because a service demander is the best entity to handle an exception that occurred in a requested service.

- Coordination of components activities must be achieved by enabling their explicit representation and by defining means to control their execution.
- Programmers must be able to configure the exception propagation policy by defining *resolution functions* to immediately handle exceptions that are really critical for the execution while logging under-critical exceptions until their conjunction enables to diagnose a unique problem represented by a *concerted exception*.
- The EHS must manage broadcasted requests in a pertinent way.

In previous work, we designed and implemented an EHS providing these qualities for MASs (the SaGE EHS[19]). The study presented in this paper is our first step towards the adaptation of the design and implementation of this work in the context of CBPs which we plan to integrate to the JOnAS platform.

## References


1. BullSoft. *Jonas*. http://www.objectweb.org/jonas/current/doc/JOnASWP.html.
2. Chrysantos Dellarocas. Toward exception handling infrastructures for component-based software. In *Proceedings of the International Workshop on Component-based Software Engineering, 20th International Conference on Software Engineering (ICSE), Kyoto, Japan, April 25-26, 1998*, 1998.
3. Christophe Dony. Exception handling and object-oriented programming : towards a synthesis. *ACM SIGPLAN Notices*, 25(10):322–330, October 1990. *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).
4. Jacques Ferber. *Les systemes multi-agents, vers une intelligence artificielle distribuée*. InterEditions, 1995.
5. John B. Goodenough. Exception handling: Issues and a proposed notation. *Communications of the ACM*, 18(12):683–696, December 1975.
6. Jun He, Matti A. Hiltunen, Mohan Rajagopalan, and Richard D. Schlichting. Providing qos customization in distributed object systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms (Middleware 2001), Lecture Notes in Computer Science 2218*, 2001.
7. Valerie Issarny. Concurrent exception handling. In A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, *Advances in Exception Handling Techniques*, LNCS (Lecture Notes in Computer Science) 2022. Springer-Verlag, 2001.
8. Mark Klein and Chrysanthos Dellarocas. Exception handling in agent systems. In Oren Etzioni, Jörg P. Müller, and Jeffrey M. Bradshaw, editors, *Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS-99)*, pages 62–68, New York, May 1–5 1999. ACM Press.
9. Mark Klein and Chrysanthos Dellarocas. Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. *Journal for Autonomous Agents and Multi-Agent Systems*, 7(1/2), 2003.
10. A. R. Koenig and B. Stroustrup. Exception handling for C++. In *Proceedings "C++ at Work" Conference*, pages 322–330, November 1989.
11. David H. Lorenz and Predrag Petkovic. Design-time assembly of runtime containment components. In Institute of Electrical and Electronics Engineers, editors, *Proceedings of the Technology of Object-Oriented Languages and Systems (TOOLS 34'00)*, 2000.
12. B. Meyer. Applying 'design by contract'. *IEEE Computer*, October 1992.
13. Bertrand Meyer. Disciplined exceptions. Technical report tr-ei-22/ex, Interactive Software Engineering, Goleta, CA, 1988.
14. M. Patino-Martinez, R. Jimenez-Peris, and S. Arevalo. Exception Handling in Transactional Object Groups. In *Advances in Exception Handling, LNCS-2022*, pages 165–180. Springer, 2001.
15. F. Peschanski. Comet: A reflective middleware architecture for adaptive component-based distributed systems, http://ads.computer.org/dsonline/0107/features/pes0107.htm, 2001.
16. A. Romanovsky, C. Dony, J. L. Knudsen, and A.Tripathi. *Advances in Exception Handling Techniques*. LNCS (Lecture Notes in Computer Science) 2022. Springer-Verlag, 2001.
17. Alexander Romanovsky. Exception handling in component-based system development. In *Proceedings of 25th Annual International Computer Software and Applications Conference (COMPSAC'01)*, 2001.
18. Alexander B. Romanovsky and Jorg Kienzle. Action-oriented exception handling in cooperative and competitive concurrent object-oriented systems. In *Advances in Exception Handling Techniques*, pages 147–164, 2000.
19. F. Souchon, C. Dony, C. Urtado, and S. Vauttier. A proposition for exception handling in multi-agent systems. In *SELMAS'03 proceedings*, 2003.

20. Sun Microsystems, Mountain View, Calif. *Enterprise JavaBeans (EJB).* http://java.sun.com/products/ejb/.
21. Sun Microsystems, Mountain View, Calif. *Java 2 Platform, Enterprise Edition (J2EE).* http://java.sun.com/j2ee.
22. Sun Microsystems, Mountain View, Calif. *Java 2 Platform, Standard Edition (J2SE).* http://java.sun.com/j2Se.
23. Sun Microsystems, Mountain View, Calif. *JavaSoft JavaBeans API Specification*, 1.01 edition, July 1997. http://javasoft.com/beans/docs/spec.htm.
24. Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming.* ACM Press and Addison-Wesley, New York, NY, 1998.
25. Anand Tripathi and Robert Miller. Exception handling in agent oriented systems. In A. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, *Advances in Exception Handling Techniques*, LNCS (Lecture Notes in Computer Science) 2022. Springer-Verlag, 2000.
26. N. Wang, D. Schmidt, K. Parameswaran, and M. Kircher. Towards a reflective middleware framework for qos-enabled corba component model applications, ieee distributed systems online special issue on reflective middleware, 2001.
27. Daniel L. Weinreb. Signalling and handling conditions. Technical report, Symbolics, Inc., Cambridge, MA, January 1983.