A Unified Meta-Level Software Architecture for Sequential and Concurrent Exception Handling

ALESSANDRO F. GARCIA¹, DELANO M. BEDER² AND CECÍLIA M.F. RUBIRA²

¹Department of Informatics, PUC/RJ, Brazil
²Institute of Computing, UNICAMP, Brazil
Email: afs Garcia@inf.puc-Rio.br, {delano, cmrubira}@ic.unicamp.br

Dependable object-oriented software systems are inherently complex and have to cope with an increasing number of exceptional conditions in order to meet the system’s dependability requirements. This work proposes a software architecture which integrates uniformly both concurrent and sequential exception handling. The exception handling architecture is independent of programming language or exception handling mechanism, and its use can minimize the complexity caused by the handling of abnormal behavior. Our architecture provides, during the architectural design stage, the context in which more detailed design decisions related to exception handling are made in later development stages. This work also presents a set of design patterns which describes the static and dynamic aspects of the components of our software architecture. The patterns allow a clear separation of concerns between the system’s functionality and the exception handling facilities, applying the computational reflection technique.

Keywords: Exception Handling, Fault Tolerance, Software Architecture, Design Patterns and Computational Reflection

1. INTRODUCTION

Complex and large software systems are common in many sectors, such as manufacturing, aerospace, transportation, communication, energy and healthcare. In general these systems have to cope with an increasing number of exceptional conditions, and are driven by demanding quality requirements on fault tolerance, security, performance, adaptability and so on. Dependable object-oriented systems detect errors caused by residual faults and employ error recovery techniques to restore normal computation [Lee and Anderson, 1990]. Exception handling provides a suitable scheme for incorporating fault tolerance activities into software systems. The inclusion of exceptional behaviour into a system is one of the major sources of software complexity. Hence, many ongoing investigations are concerned with mastering this software complexity by means of effective exception handling techniques for in order to enhance the system’s robustness.

The idea of exception handling is not new, and various modern object-oriented languages (e.g. Java, C++, Modula-3) incorporate exception handling facilities. Nevertheless, the exceptional activity of a component is amalgamated to its normal activity. Ideally software components should incorporate their abnormal activity in a structured and transparent manner and, during the development of OO software, a systematic method for incorporating exceptional behaviour should be adopted in order to produce a well-organized robust software. The detection of an error will result in an exception being raised, followed by the search of an appropriate handler (called sequential exception handling) which will automatically execute fault tolerance measures [Lee and Anderson, 1990]. The presence of an exception handling system can reduce software development efforts since they support: (i) representation of errors as exceptions, (ii) definition of handlers to deal with exceptions, and (iii) employment of an adequate strategy for handling an exception upon its occurrence.

Object-oriented systems may consist of various threads (or processes) executing methods concurrently on objects. Exceptions are more difficult to handle in concurrent object-oriented systems than in sequential ones due to the presence of cooperative concurrency [Campbell and Randell, 1986]. That is, several concurrent threads can cooperate in a complex interactive pattern in order to perform some system’s activity. In this context, erroneous information may be spread directly or indirectly through inter-thread communication during a cooperative activity. A general approach for structuring cooperative activities and employing exception handling in concurrent systems extends the well-known atomic ac-
tion notion [Campbell and Randell, 1986]. An atomic action is formed by a group of participants which are executed by cooperating threads. The group co-operates within the scope of an action which manages activities related to concurrent exception handling. Participants may join an action asynchronously but they have to leave it synchronously to guarantee that no information is smuggled to or from the action. When an exception is raised inside an action, all action participants should participate in the error handling [Campbell and Randell, 1986]. In general, different exception handlers for the same exception have to be invoked in different participants. These handlers are executed concurrently in order to handle the exception in a coordinated way. An additional difficulty is that several different exceptions can be raised concurrently by participants during a cooperative activity. In this case, an exception resolution algorithm will decide which exception should be notified to all participants.

Very few modern languages give direct support to concurrent exception handling (e.g., Arche [Issam 1993]). More recently other 'ad hoc' solutions have been proposed for the provision of concurrent exception handling that extend programming languages, such as Ada and Java [Romanovsky, 1997; Zorzo et al., 1999]. These approaches are largely complex, language-dependent and error prone; they are also intrusive since application code is amalgamated with the exceptional activity responsible for exception resolution and final synchronization of action participants. Consequently, the use of these solutions may produce software products which are non-reliable and difficult to understand, maintain and reuse.

The present interest in software architectures and design reuse motivated us to develop an exception handling software architecture for building dependable software. The degree to which quality requirements (e.g., fault tolerance) are met are largely dependent on the software system's architecture. Hence, if an architecture that includes design policies for error handling is chosen from the outset, then it is more likely that a proper use of exception handling throughout the development of the system will be achieved. Our approach provides, early in the architectural design stage, the context in which more detailed decisions related to error handling can be made in later stages of the development. The proposed architecture provides a generic infrastructure which supports uniformly both concurrent and sequential exception handling. Moreover, the architecture is independent of a programming language or exception handling mechanism, and its use can minimize the complexity caused by the handling of abnormal activities. The architecture is composed of four components: (i) the Exception component, (ii) the Handler component, (iii) the Exception Handling Strategy component, and (iv) the Concurrent Exception Handling Action component. The static and dynamic aspects of these components are described by a set of exception handling design patterns. These patterns allow a neat and transparent separation of concerns between the application's functionality and the exception handling facilities, applying the computational reflection technique.

The remainder of this paper is organized as follows. Section 2 presents the terminology used throughout this paper and gives an introductory discussion of exception handling. Section 3 describes the object-oriented techniques for design reuse and software structuring used in this work. Section 4 shows the proposed software architecture for exception handling. Section 5 presents a set of design patterns which refine the components of our architecture. Section 6 discusses some implementation issues. Section 7 presents some related work, and finally, Section 8 summarizes our conclusions and suggests directions for future work.

2. EXCEPTION HANDLING

2.1. Exception Handling in Sequential Systems

Dependable software developers usually refer to errors as exceptions because they are expected to occur rarely during a system's normal activity. These exceptions should be specified internally to the system and an instance of an exception raised at run-time is termed an exception occurrence. Extra-information about an exception occurrence, such as its name, description, location, and severity, is usually necessary for handling that occurrence. This extra-information is passed either explicitly by the application component that has raised the exception, or implicitly by an exception handling service.

Dependable applications need to incorporate exception handling activities in order to behave suitably in a great number of exceptional situations. Exception handling activities are structured by a set of exception handlers (or simply handlers). A handler is the part of an application code that implements the fault tolerance measures for recovering the system from a detected exception. A handler may be valid for one or more exceptions. Handlers are attached to a particular region of normal code called a protected region. Each protected region can have a set of attached handlers, and one of them is invoked when a corresponding exception is raised. Handlers can be attached to blocks of statements, methods, objects, classes, or exception classes. Handlers attached to exception classes, called default handlers, are the most general handlers, and must be valid in any part of the program, independently of any execution context and object state. For the purpose of improving the writeability and structuring of the software systems, it is desirable to allow some flexibility concerning the attachment of handlers. It should be possible the multi-level attachment of handlers i.e., the attachment of handlers to several levels of protected regions such as classes, objects, methods and so on.
An exception handling strategy should be followed after an exception occurrence is detected. In general, the normal control flow of the computation is deviated to the exceptional control flow. The deviation of the control flow is followed by the search for a suitable handler to deal with the exception occurrence. The handler search is performed according to a search algorithm. When a handler is found, it is invoked and the computation is returned to its normal control flow. The returning point where the normal flow continues also depends on the chosen model for the continuation, namely, the termination model, or the resumption model. In the termination model, execution continues from the point at which the exception was handled. In the resumption model, the execution has the capability to resume the internal activity of the component after the point at which the exception was raised. The semantic of the termination model is simpler and more suitable for construction of dependable software [Cristian, 1982].

2.2. Exception Handling in Concurrent Systems

In this work, cooperative activities of a dependable concurrent object-oriented system are structured as a set of atomic actions. We refer to these activities as concurrent cooperative actions (or simply actions). An action provides a mechanism for performing a group of methods on a collection of objects concurrently. The interface of an action includes its participants and methods (and their respective objects) that are manipulated by the participants. In order to perform an action, a group of threads should execute each participant in the action concurrently (one thread per participant). Threads participating in an action cooperate within the scope of the action by executing methods on objects, and exchange information only among those who are participants of that action. Threads cooperate and communicate with each other by means of shared objects. Participants may enter the action asynchronously but they have to exit the action synchronously to guarantee that no information is smuggled to or from the action.

We introduce a banking service example based in [Campbell and Randell, 1986] that illustrates the concepts of concurrent exception handling. Figure 1(a) shows the structuring of concurrent cooperative actions in the banking service example. Threads participating in the action are represented by solid lines, inter-thread communication by dotted lines, and actions by rectangles. Action participants are activated by threads which cooperate within the action's scope for performing the banking service. The participants of the action Service are Client, Client's Agency and Payer's Agency. Consider a Client that presents a check (i.e., an object of the type Check) to his/her bank and receives a Receipt that certifies the operation. To clear the check, the Client’s Agency sends the Check to Payer’s Agency which has the payer's account. Once Client’s Agency receives the Cash for the check, it sends to Client a new Statement of his/her account. Actions can be nested and exceptions may be propagated over nesting levels. In any moment, participants can start nested actions. Figure 1(a) shows two nested actions for the action Service. The participants Client and Client’s Agency perform the nested action BankMoney, and the participants Client’s Agency and Payer’s Agency perform the nested action ClearCheck.

Exception occurrences can be raised by participants during an action. Some of them can be handled internally by a local handler attached to the participant that raised that exception. We refer to these exceptions as local exceptions. Traditional exception handling strategies address this kind of exception. If an exception occurrence is not handled internally by a participant, then it should be handled cooperatively by all action participants. This kind of exception is called a cooperating exception, for which, a new concurrent exception handling strategy is required. A set of cooperating exceptions is associated with each action. Each participant has a set of handlers for (all or part of) these exceptions. Participants are synchronized and probably different handlers for the same exception have to be invoked in all participants [Campbell and Randell, 1986]. These handlers are executed concurrently, and cooperate in handling the exception in a coordinated way. Moreover, vari-

![A Unified Meta-Level Software Architecture for Sequential and Concurrent Exception Handling](image-url)
ous cooperating exceptions may be raised concurrently while participants are cooperating in the action. So, an algorithm for *exception resolution* is necessary in order to decide which cooperating exception will be notified to all participants of the action.

The work in [Campbell and Randell, 1986] describes a model for exception resolution called *exception tree*, which includes an exception hierarchy. If several cooperating exceptions are raised concurrently, the resolved exception is the root of the smallest subtree containing all raised exceptions. Cooperating exceptions can be of two different kinds in the exception tree: (i) *simple exceptions*, or (ii) *structured exceptions*. Simple exceptions are leaves of the tree and correspond to cooperating exceptions which are raised one at a time. Structured exceptions are non-leaf nodes and correspond to two or more exceptions being raised concurrently. An exception tree should be specified for each action of the application. In Figure 1(a), during the action *ClearCheck*, two cooperating exceptions are raised concurrently, namely *WrongDateException* and *InsufficientFundsException*. Figure 1(b) presents the exception tree specified for the action *ClearCheck*. The structured exception *BouncedCheckException* represents the concurrent raising of the simple exceptions *WrongDateException* and *InsufficientFundsException*.

Participants of an action can exit an action on three occasions: (i) when no exceptions were raised; (ii) when cooperating exceptions have been raised, and handlers have successfully dealt with them; and (iii) signaling a *failure* exception to the containing action if a cooperating exception has been raised and no proper handlers were found or the handling of that exception was not possible. There are at least two distinct approaches for concurrent exception handling: (i) the *blocking* approach, and (ii) the *pre-emptive* approach. In blocking schemes, each participant terminates by reaching the end of an action or fails by raising a cooperating exception. Participants are informed of an exception occurrence only when they are completed (or detect a cooperating exception); that is, when they are ready to accept information about the state of other participants. In contrast, pre-emptive schemes do not wait but require some language feature to interrupt all participants when cooperating exceptions are raised [Romanovsky, 2000]. In blocking systems, exception handling and resolution are easier to provide than in pre-emptive ones because each participant is ready for handling when handlers are invoked. Moreover, there is no need to perform the abortion of nested actions because they have either been completed successfully or have had exceptions dealt by nested action’s handlers.

2.3. Integration of Sequential and Concurrent Exception Handling

Figure 2 illustrates the integration of sequential and concurrent exception handling. Sequential exception handling facilities include: (i) exceptions - the definition and raising of local exceptions, and management of extra-information about exception occurrences. (ii) handlers - the definition and invocation of handlers, and (iii) exception handling strategy - the specification of an algorithm for handler search, and a model for continuation of the control flow. As discussed earlier, concurrent exception handling requires some extra support not required by sequential systems. So, an integrated approach to exception handling should support both local and cooperating exceptions, and also a concurrent exception handling strategy. Ideally the concurrent exception handling strategy should be consistent with the exception handling strategy (of the sequential exception handling). In this work, the strategy for concurrent exception handling extends the atomic action paradigm described previously.
3. DESIGN REUSE AND SOFTWARE STRUCTURING TECHNIQUES

3.1. Software Architecture and Patterns

A system's software architecture is a high-level description of the system's organization in terms of components and their interrelationships [Shaw and Garlan, 1996]. Components are physical and replaceable parts of an architecture; to each component are assigned a set of responsibilities. The components must interact with each other using pre-described rules, and must fulfill their responsibilities to other components as imposed by the architecture. Each component conforms to and provides the realization of a set of interfaces which make available services implemented by the component.

Software patterns are an important vehicle for constructing high-quality architectures. Patterns are abstracted from recurring experiences rather than invented, and exist at different levels of abstraction. Architectural patterns define the basic structure of an architecture and systems which implement that architecture [Buschmann et al., 1996]; while design patterns are more problem-oriented than architectural patterns, and are applied in later design stages. Usually, the selection of a design pattern is influenced by the architectural pattern that was previously chosen. A design pattern should balance a set of opposing forces. Design patterns can refine general components of an architecture, providing detailed design solutions.

In this work, components of the proposed architecture are refined by a set of design patterns which follows the overall structure of the Reflection architectural pattern [Buschmann et al., 1996]. This pattern is based on computational reflection and generate meta-level architectures, a concept discussed in the next section.

3.2. Computational Reflection and Meta-Level Architectures

Computational reflection [Maes, 1987] is defined as the ability of observing and manipulating the computational behavior of a system through a process called reflection. This technique allows a system to maintain information about itself (meta-information) and use this information to change its behavior. It defines a meta-level architecture which is composed of at least two dimensions: (i) a base level and (ii) a meta-level. A meta-object protocol (MOP) establishes an interface among the base-level and the meta-level components. A MOP provides a high-level interface to the programming language implementation in order to reveal the program information normally hidden by the compiler and/or run-time environment. As a consequence, programmers can develop language extensions, and adapt component behavior or even make changes to the systems more transparently.

The extensions of the behavior of base-level objects can be implemented at the meta-level. Reflection can be used to intercept and modify the effects of operations of the object model. For the purpose of illustration, suppose that for each base-level object o there exists a corresponding meta-object mo that represents the behavioral and structural aspects of o. As illustrated in Figure 3, if an object x sends a message m1 to an object o, the meta-object mo intercepts the message service, reifies the base-level computation and takes over the execution; later mo returns (reflects) the response to x. From the point of view of object x, computational reflection is transparent: x sends a message requesting a service to o, and receives the response with no knowledge that the message has been intercepted and redirected to the meta-level.

![FIGURE 3. A Meta-Level Software Architecture.](image)

4. A SOFTWARE ARCHITECTURE FOR EXCEPTION HANDLING

This section presents a generic software architecture that integrates sequential and concurrent exception handling. Applications reuse our architecture to handle their exceptional situations by using the exception handling facilities provided by the architecture's components. The architecture has four components: (i) the Exception component, (ii) the Handler component, (iii) the Exception Handling Strategy component, and (iv) the Concurrent Exception Handling Action component. Figure 4 summarizes the components and their responsibilities. The responsibilities are classified in two groups: (i) application-dependent responsibilities (ADR), and (ii) application-independent responsibilities (AIR).

![FIGURE 4. Components' responsibilities](image)

Application-dependent responsibilities are directly related to the application's functionality and include...
for instance, facilities for specification of exceptions and handlers, raising of application exceptions, and specification of concurrent cooperative actions. Software developers either invoke services provided by the architecture's component interfaces (Section 4.1) or else refine the design of architecture's components according to their needs (Section 4.2). For instance, application's components invoke the service provided by the Exception component in order to raise an application exception. So software designers can tailor the Exception, Handler and Concurrent Exception Handling Action components to the needs of their applications (Section 4.2).

Application-independent responsibilities include, for instance, facilities for extra-information management, handler invocation, deviation of control flow, handler search, participant synchronization and exception resolution. These responsibilities are related to management activities of exception handling. The components of our proposed architecture perform their management activities in a way that is transparent to the application. The architecture's components interact with each other as prescribed by the architecture in order to fulfill their application-independent responsibilities.

Figure 5 depicts the components and their interrelationships. The Exception component works as an extra-information holder, keeping information about application exceptions which are used by the other components. They interact with the Exception component in order to get and update information about exception occurrences.

**FIGURE 5.** A software architecture for exception handling

The Exception Handling Strategy component implements services related to the general strategy for exception handling. Its responsibilities are the deviation of control flow and the search for handlers. This component plays a central role in the architecture and interacts with all other components. It interacts with the Exception component to get extra-information about an exception occurrence while searching for its corresponding handler. After a handler is found, it asks the Handler component to invoke the exception handler. The Exception Handling Strategy component also interacts with the Concurrent Exception Handling Action component. The latter uses the services provided by the former in order to carry out the strategy for concurrent exception handling. For example, when cooperating exceptions are raised during an action, the exception resolution is accomplished by the Concurrent Exception Handling Action component, and then the Exception Handling Strategy component is responsible for locating the different handlers for the resolved exception. In this work, the strategy for concurrent exception handling extends the atomic action paradigm described previously in Section 2.2.

### 4.1. Interfaces of the Components

The interfaces are used either by the architecture's components themselves, or else by the application while using the exception handling services. Figure 5 illustrates the architecture's components and their interfaces. The interfaces are categorized in two groups: (i) private interfaces, or (ii) public interfaces. Private interfaces define the services that are only visible by the components of the architecture. Public interfaces define the services that are visible by both the application and architecture. Figure 6 depicts all of the interfaces provided by the architectural components.

**FIGURE 6.** The Detailed Interfaces

The Exception component implements three public interfaces: (i) the interface IRaising, (ii) the interface IGetInformation, and (iii) the interface IUpdateInformation. The interface IRaising allows the application to
raise exceptions by invoking the method `raise`. The interface `IGetInformation` allows the application and other architecture’s components to obtain extra-information about the exception occurrences. Finally, the interface `IUpdateInformation` allows its clients to update extra-information about exceptions. The `Handler` component implements the private interface `IInvocation`. This interface allows the `Exception Handling Strategy` component to invoke a handler when the appropriate handler has been found. The `Exception Handling Strategy` component conforms to the private interface `ISearch` that provides the `Concurrent Exception Handling Action` component with the service for handler search. The `Concurrent Exception Handling Action` component implements the public interface `ICooporation` which is accessible by the application to create concurrent cooperative actions.

The components collaborate to realize the set of scenarios. Figure 7 illustrates the interactions between the application and the interfaces of the architecture’s components in order to handle a cooperating exception that was raised during a concurrent cooperative action. A thread of the application invokes the method `join` in order to take part in the the action and perform a specific action participant (1). The `Concurrent Exception Handling Action` component invokes the application’s method to be executed by the application’s thread (2). While this method is being carried out by the participating thread, it obtains the shared objects used for inter-thread communication (3), and passes explicitly extra-information concerning the exception occurrence (4). During its execution, the application’s method raises a cooperating exception (5).

After an exception is raised, the architecture’s components interact with each other to accomplish the management activities. Extra-information about that exception occurrence is updated implicitly by the components (6). The action participants are synchronized and exception resolution process is executed within the `Concurrent Exception Handling Action` component.

---

**FIGURE 7.** A Scenario of the Proposed Software Architecture.

**FIGURE 8.** The architecture refinement
During the resolution, this component communicates with the Exception component in order to obtain extra-
information about the raised cooperating exceptions (7). The Concurrent Exception Handling Action compo-
nent asks the Exception Handling Strategy component to search the handler for the resolved exception
(8), and the later asks the Handler component to in-
voke it (9). The Handler component invokes the han-
der defined in the application (10). The handler ob-
tains extra-information useful for handling the resolved
exception (11-12), and shared objects that are used in
the cooperation with the other handlers being executed
concurrently (13).

4.2. The Architecture Refinement

Separation of Concerns. As stated previously, soft-
ware designers should tailor the components of the ar-
chitecture in order to add the functionality related to
specific applications. Note that each architectural com-
ponent include application’s functionality and also ma-
agement activities for exception handling. In order to
obtain a clear separation of concerns between the
application’s functionality and the exception handling
services, the architecture and their components incor-
porate a meta-level architecture, following the over-
all structure of the Reflection pattern (Section 3.1).
Figure 8 presents the proposed meta-level architecture
which is composed of two dimensions: the base level,
and the meta-level. The architecture’s base level en-
compases the application-dependent elements, such as
exceptions, handlers, normal activities, and concurrent
cooperative actions. The architecture’s meta-level con-
sists of meta-objects which perform the management
activities for exception handling.

Transparency. The Reflection pattern also captures
the benefit of transparency provided by computational
reflection. For the purposes of this work, object states,
results and invocations of methods of the application
(base-level) are intercepted and reified by the MOP,
and potentially checked and altered by the meta-objects
(meta-level) in order to carry out the management ac-
tivities for exception handling. For instance: results of
methods are checked transparently by the meta-objects
to verify if such methods have raised any exception.
MOP intercepts at run-time the exceptional results and
deviates the normal control flow of the base-level
application to the exceptional one at the meta-level.
When the management activities are concluded, MOP
returns the computation to the application’s normal
flow. Therefore, the meta-objects execute their manage-
ment activities transparently from the viewpoint of
the base-level.

Refinement of the Components and Design Pat-
terns. Some questions arise in this context, such as: (i)
how to specify simple and cooperating exceptions and
to handle them uniformly, (ii) how to specify handlers,
and (iii) how to perform the synchronization of action
participants and other management activities in a way
that is transparent to the application. In this work,
design patterns are proposed in order to refine the gen-
eral components of our architecture. However, we would
like to emphasize that it is possible to apply the design
patterns for exception handling in the absence of reflec-
tion. In such a case, the benefits of transparency are
lost. Due to space limitations, in this work, we discuss
only the reflective version of the design patterns.

5. DESIGN PATTERNS FOR EXCEPTION
HANDLING

5.1. The Exception Pattern

Context. Dependable software designers should be
able to specify local and cooperating exceptions in their
applications. These exceptions may be raised at run-
time during the application’s normal activity. Extra-
information is required by the application in order to
handle an exception occurrence.

Problem. The software architecture should support
the definition and raising of local and cooperating ex-
ceptions. Moreover, a flexible and reusable software ar-
chitecture is required to make the exception specifi-
cation easier and to separate concerns between application
exceptions and extra-information management. Several
forces are associated with this design problem: (i) local
and cooperating exceptions should be defined uniformly.
(ii) software developers should be able to construct ex-
ception trees easily, and (iii) the exception occurrence
itself should carry the extra-information necessary for
its handling.

Solution. Use the Reflection architectural pattern
in order to separate classes responsible for manag-
ing extra-information (meta-level) from the ones used
to specify application exceptions (base level). Differe-
tent types of exceptions are organized hierarchically as
classes which are termed exception classes (Figure 9).
These classes derive from the root class Exception. Ex-
ception trees are defined by using the Composite design
pattern [Gamma et al., 1995]. Exception occurrences are
base-level objects created at run-time when an ex-
ception is raised, and are termed exception objects. Ex-
ceptions are raised by calling the method raise on excep-
tion objects. Meta-objects are associated transparently
with exception objects for keeping extra-information
about the exception occurrences. Extra-information is
reified as meta-objects which keep meta-information
collected at run-time about the corresponding exception
occurrence. Meta-objects alter transparently the state of
the exception objects in order to make this in-
f ormation available for the application. As a result, the
exception object keeps extra-information necessary for
its handling. The application obtain this information
by invoking methods on exception objects.

**Structure.** The Exception Pattern consists of exception classes, and meta-objects. Meta-objects of the type MetaException are associated with instances of base-level exception classes. Meta-objects are associated with exception objects. Application developers are provided with three main exception classes – LocalException, CooperatingException and StructuredException (Figure 9(a)). These classes derive from the root class Exception. The class LocalException defines the local exceptions of applications; it is subclassed by application designers in order to specify the local exceptions. Exception trees are easily specified – an application developer only needs to create a class for each simple exception by subclassing the class CooperatingException, and a new instance of the class StructuredException for each structured exception. An exception object of the type StructuredException stores the simple and/or structured exceptions which compose it. The method getSimpleExceptions returns the simple exceptions that compose a structured exception. Figure 9(b) shows an instance of the proposed pattern for defining an exception tree in the banking service application (Section 2.2). This scheme for definition of exception trees is similar to the structure of the Composite design pattern [Gamma et al., 1995].

**Dynamics.** Figure 10 presents the interaction diagram that illustrates a scenario for the banking service example (Section 2.2). A meta-object (of the type MetaSearcher), associated with the application object that has raised the exception InsufficientFundsException, relieves extra-information about the location where the exception was detected. The information includes the method, the action and the action participant where the exception was raised. This meta-object sends the extra-information to the meta-object associated with the application’s exception object that represents that exception occurrence. The meta-object updates extra-information about the exception occurrence by invoking transparently the method setLocation on the application’s exception object. The invocation and the update are transparent from the viewpoint of the application. Then the method getLocation is invoked by the application handler in order to receive the extra-information related to the location.

**Known Uses.** The representation of exceptions as classes is a design solution adopted by several systems and programming languages, such as Java, C++ and Arche.

**Consequences.** The Exception Pattern offers the following benefits:

- **Uniformity.** Both local and cooperating exceptions are uniformly defined as exception classes. Moreover, the Exception pattern allows application designers to treat simple exceptions and their compositions (structured exceptions) uniformly since it adopts the Composite design pattern [Gamma et al., 1995].

- **Simplicity.** Exception trees are easily defined. Application developers define exception trees without writing an exception resolution procedure for each concurrent cooperative action of the application. In addition, the exception resolution algorithm is performed transparently by the meta-level (Section 5.4).

- **Reusability and Extensibility.** The representation of local and cooperating exceptions as classes promotes the reusability and extensibility of the exception classes. In addition, the separation of concerns provided by the Exception pattern also promotes the reusability of the management services.

- **Readability and Maintainability.** Applications whose exceptions are represented as objects are easier to understand and maintain than applications where exceptions are mere symbols (numbers or strings) [Garcia, 2000].

5.2. The Handler Pattern

**Context.** Dependable software designers need to specify handlers for local and cooperating exceptions that are expected to occur during the normal activity of their applications. A handler is invoked when a corresponding exception is raised.

**Problem.** The infra-structure of the software architecture should be organized in order to allow application developers to define the exception handlers in a way that separates them from the application’s normal activity. In addition, this infra-structure should promote the separation between the application components containing the exception handlers and the architectural components responsible for invoking the eligible handler. The following forces shape this solution: (i) exception handlers for local and cooperating exceptions should be defined in an uniform manner, and (ii) the software architecture should include multi-level attachment of handlers (Section 2.1).

**Solution.** Use the Reflection architectural pattern in order to separate the class responsible for invoking handlers (meta-level) from the classes used to specify the application handlers (base level). The base-level defines the exceptional classes i.e., the application classes that implement the handlers for local and cooperating exceptions. The methods of exceptional classes are
FIGURE 9. Exception pattern class diagram

FIGURE 10. Exception Pattern Interaction Diagram
the handlers for the local and cooperating exceptions raised during the execution of normal classes’ methods. The **normal classes** are located at the base-level and implement the application’s normal activities (see Section 5.3). Therefore, exceptional classes implement the handlers of the application and they are attached to the corresponding normal classes. Exceptional classes can contain method, object and class handlers. The meta-level consist of meta-objects which are associated with exceptional classes, and are responsible for invoking the exception handlers transparently.

**Structure.** The **Handler Pattern** consists of two kinds of elements: (i) exceptional classes, and (ii) meta-objects of the type **MetaHandler** (Figure 11(a)). The **exceptional classes** are located at the base-level and define the error handling activities of a specific application. The methods of exceptional classes are the handlers for the local and cooperating exceptions raised during the execution of normal classes’ methods. The **normal classes** are located at the base-level and implement the application’s normal activities (see Section 5.3). Therefore, exceptional classes implement the handlers of the application and they are attached to the corresponding normal classes. Meta-objects of the type **MetaHandler** are associated with exceptional classes, and are responsible for invoking transparently the exception handlers. Exceptional classes can contain handlers attached to classes, objects and methods. Each exceptional class may contain handlers for coping with the local and cooperating exceptions; they are invoked when these exceptions are raised during the execution of methods of the corresponding normal class. Figure 11(b) shows an instance of this pattern for the banking service example. The methods of the **ExceptionalAccount** are the handlers for the simple and structured exceptions that can be raised while the methods of the corresponding normal class (the class **NormalAccount** — Section 5.3) are being executed during a concurrent cooperative action.

**Dynamics.** Suppose the method **withdraw** is being executed concurrently during a concurrent cooperative action and raises the exception **InsufficientFundsException**; another method is being executed concurrently during this concurrent cooperative action and also raises an exception, the exception **WrongDateException**. The concurrent raising of these simple exceptions means the occurrence of the structured exception **BouncedCheckException**, and the subsequent invocation of the handlers to deal with this structured exception. Figure 12 illustrates the transparent invocation of the appropriate handler by the meta-object associated with the exceptional class. During the execution of the handler, it gets extra-information about the location where the exception **InsufficientFundsException** was raised.

**Known Uses.** The work [Hof et al., 1997] also uses the computational reflection technique in order to obtain meta-information about the application and invoke the suitable handler when an exception is raised. Meta-level structures implement the exception handling mechanism while at the base level resides the application. Finally, handlers also are implemented as ordinary methods. The approach presented in [Mitchell et al., 1998] uses a variant of the **Handler pattern**. This variant transfers the handler methods from the exceptional classes to the meta-level. The meta-objects associated with the normal classes contain application’s methods responsible for performing the exception handling. Instead of utilizing reflective principles to complete the separation between application and management mechanisms, this variant explores reflection to separate normal and exceptional code of the application.

**Consequences.** **Handler Pattern** has the following consequences:

- **Uniformity.** Handlers for both local and cooperating exceptions are defined uniformly as methods on exceptional classes.
- **Readability and Maintainability.** This pattern provides explicit separation between normal and error-handling activities, which in turn promotes readability and maintainability.
- **Flexibility.** The multi-level attachment of handlers allows developers to attach handlers to the respective levels of classes, objects and methods.
- **Reusability.** The use of normal and exceptional classes allows application designers to compose an **exceptional class hierarchy** that is orthogonal to the normal class hierarchy of the application. The exceptional classes are organized hierarchically so that the resultant hierarchy is orthogonal to the **normal class hierarchy**. Exceptional class hierarchies allow exceptional subclasses to inherit handlers from their superclasses and, consequently, they allow exceptional code reuse. When reuse is not desired, the handler method can be redefined at the subclasses.
- **Lack of Static Checking.** A possible disadvantage of this pattern is that it may not be easy to check statically if handlers have been defined for all specified exceptions. However, alternative solutions may be applied (Section 6).

### 5.3. The Exception Handling Strategy Pattern

**Context.** Exception occurrences can be detected during execution of a protected region of the application’s normal activity. The normal control flow is deviated to the exceptional one and an appropriate handler is searched.

**Problem.** The software architecture should be organized in a disciplined manner: the components responsible for the deviation of the normal control flow and for the handler search should perform their management...
Figure 11. Handler Pattern Class Diagram

Figure 12. Interaction Diagram for the Handler Pattern.
activities in a non-intrusive way to the application.

The following force arises when dealing with such a problem: the chosen model for continuation of the control flow should be termination since it is more suitable for developing dependable systems (Section 2.1).

**Solution.** Use the Reflection architectural pattern in order to separate classes responsible for the management activities (meta-level) from the ones that implement the normal activities of the application (base level). The base-level defines the application’s logic where normal classes implement the normal activities. The meta-level consists of meta-objects which search transparently for the exception handlers. Meta-objects are associated with instances of the normal classes, and maintain meta-information concerning the protected regions defined at the base-level. A protected region can be a method, an object, and a class. The MOP is responsible for intercepting method results and switching normal control flow to exceptional one when exceptions are detected, by transferring control to the meta-level. With the available meta-information, meta-objects find the handler that should be executed when an exception occurrence is detected in a given protected region. When the execution of the handlers is concluded successfully, the MOP returns control flow to the application’s normal computation according to the termination model.

**Structure.** The Exception Handling Strategy Pattern introduces two types of elements: (i) normal classes, and (ii) meta-objects of the type MetaSearcher (Figure 13(a)). The normal classes are located at the base-level and define the normal activities of a specific application. They are associated with the corresponding exceptional classes. Figure 13(b) pictures an instance of this pattern for the banking service application. This figure shows the normal class NormalAccount; it is attached to the exceptional class ExceptionalAccount (Section 5.2). Meta-objects of the type MetaSearcher are associated with instances of normal classes, and are responsible for the interruption of the normal control flow and the handler search.

**Dynamics.** Figure 14 presents the interaction diagram for the banking service example. The method withdraw is being performed concurrently during a concurrent cooperative action. The exception InsufficientFundsException is returned as the result of withdraw since it has raised this exception during its execution. MOP intercepts and refines the result of withdraw, and notifies the meta-object about the exceptional result by means of the method handleResult. The meta-object checks if the exception occurrence is a local or a cooperation exception. If it is a local exception, the meta-object searches immediately for the exception handler based on the available meta-information. Otherwise, the cooperating exception is firstly delegated to the meta-object responsible for the participant synchronization and exception resolution. InsufficientFundsException is then delegated since it is a cooperating exception. After exception resolution is accomplished, the meta-object is required to find a handler for the resolved exception, the exception BouncedCheckException. The handler is found and the invocation of it is delegated to the appropriate meta-object (Section 5.2). Since the exception handler is executed successfully, the control is passed to the meta-object responsible for the participant synchronization, which in turn will deviate the exceptional control flow to the normal one.

**Variants.** **Reflective Handlers.** This variant [Mitchell et al. 1998] transfers the handler methods from the exceptional classes to the meta-level. The meta-objects associated with the normal classes contain application’s methods responsible for performing error recovery. Instead of utilizing reflective principles to complete the separation between application and management mechanisms, this variant explores reflection to separate normal and abnormal code.

**Reified Exceptions.** In this variant [Mitchell et al. 1998], the exception itself is the reified entity instead of operations result. Such an alternative allows the exception to control the handling. Consequently, it is possible implement the resumption of the method where the exception occurred since the control flow was stopped exactly at the point of the exception raising.

**Known Uses.** The work [Mitchell et al. 1998] presents a variant of the Exception Handling Strategy pattern. In this variant, the exception itself is the reified entity instead of a method result. This alternative design solution allows the exception itself to control the handling. Consequently, it is possible to implement the resumption model since the control flow is stopped exactly at the point of the exception raising. The work [Hof et al., 1997] uses the reflection technique in order to obtain at compile-time information concerning protected regions and the handlers that are attached to them.

**Consequences.** Exception Handling Strategy Pattern has the following consequences:

- **Transparency.** The meta-level objects bind transparently the normal activity and corresponding handlers without requiring programmers to use new features to specify protected regions.
- **Readability and Maintainability.** The normal code is not amalgamated with the exceptional code. As a consequence, both normal and exception code are easier to read and maintain.
- **Compatibility.** The Exception Handling Strategy pattern can be used together with an exception handling strategy implemented in the underlying
FIGURE 13. Class Diagram of the Exception Handling Strategy Pattern.

5.4. The Concurrent Exception Handling Action Pattern

Context. Dependable software designers should be able to specify concurrent cooperative actions. These actions must be controlled at run-time and their participants have to exit the action synchronously. During the execution of an action, a number of cooperating exceptions can be raised. As a consequence, a service of exception resolution is necessary to determine the cooperating exception which is to be handled by all participants of the action.

Problem. The software architecture should support the definition of concurrent cooperative actions. Moreover, a disciplined approach is required in order to separate concerns and minimize dependencies between the concurrent cooperative actions of the application and the strategy for concurrent exception handling (i.e., the management mechanisms for synchronization and exception resolution). Some forces are associated with this design problem: (i) the definition of concurrent cooperative actions should be done in a structured manner to avoid an increase in the software's complexity, (ii) the strategy for concurrent exception handling should be a consistent extension of the general strategy for exception handling, and (iii) the blocking approach should be used for concurrent exception handling since it is simpler and easier to implement (Section 2.2).

Solution. Use the Reflection architectural pattern for segregating classes responsible for the management mechanisms (meta-level) from the classes which must be derived for defining the concurrent cooperative actions of the application (base-level). Based on a metalevel architecture, the Concurrent Exception Handling Action pattern separates objects into well-defined levels (Figure 15(a)). The base-level provides developers with classes for creating the concurrent cooperative actions of their applications; the definition of nested actions is also supported in order to control the system's complexity and allow better organization of both normal and error handling activities of the enclosing action. The MOP itself intercepts and relays invocations of methods and their results. The meta-level implements the management mechanisms based on refined invocations and results, and on the available meta-information. So, meta-objects are responsible for synchronizing the action participants and perform the exception resolution process. Figure 15(b) shows an instance of the proposed pattern for defining the actions, the participants, and threads for the banking service application (Section 2.2).

Structure. The Concurrent Exception Handling Action Pattern introduces five types of objects: (i) Action, (ii) Participant, (iii) Thread, (iv) MetaParticipant, and (v) MetaAction (Figure 15(a)). The class Thread represents the threads which intend to participate in a concurrent cooperative action. Developers create their threads, and extend the classes Action and Participant by subclassing them to implement their concurrent cooperative actions. Instances of these subclasses represent at run-time a specific action and their participants respectively. Developers should redefine the method ConfigureSharedObject while subclassing the class Action. The method ConfigureSharedObject implements the application-dependent activity which consists of creating shared objects used for purpose of inter-participant communication (inter-thread communication). In order to access these objects, each participant have to ask to its corresponding action references to these objects by means of the method getSharedObject. If an action is composed of one or more nested actions, developers should also redefine the method ConfigureNestedActions in order to create the objects that represent the nested actions. In order to access these objects, each participant have to ask to its corresponding action references to these objects by invoking the method getNestedAction. Each object of the type Participant holds references to: (i) its action, and (ii) an object and its method that will be executed during the action by a thread. Instances of the class Action have references to: (i) action participants, (ii) internal and failure exceptions, (iii) its parent (enclosing action), (iv) its nested actions, and (v) shared objects. Internal exceptions are the exceptions that should be handled within action by all action participants, while external exceptions are the exceptions that should be signaled to the enclosing action. Figure 15(b) shows an instance of the proposed pattern for defining the actions, the participants, and the threads for the banking service application. Instances of the class MetaParticipant are associated with instances of subclasses of Participant. These meta-objects are responsible for: (i) execute the application's method which is held by its associated participant, (ii) inform to its corresponding MetaAction about the end of this method execution, and (iii) ask the appropriate meta-object to invoke the handler associated with a resolved exception. Instances of the class MetaAction are associated with instances of subclasses of Action. These meta-objects are responsible for: (i) perform the exception resolution, and (ii) synchronize the action participants.

Dynamics. Figure 16 presents the interaction diagram for the banking service example. The diagram illustrates the application's thread performing the participant PayerAgency within the action ClearCheck. This thread intending to participate in the action, calls the method join on the object ClearCheck corresponding to that action. The thread informs to the action what participant (the participant PayerAgency) it intends to execute during the concurrent cooperative action MOP.
FIGURE 15. Class Diagram of the Concurrent Exception Handling Action Pattern.

intercepts and reifies the invocation of join, and notifies the meta-object MetaAction about this invocation by means of the method handleOperation. This meta-object checks to see if it is allowed to play that participant in this action, and if so, the meta-object MetaParticipant executes the method withdraw that is attached to that participant. While this method is being carried out by the thread, it obtains the shared objects used for inter-thread communication. The exception InsufficientFundsException is returned as the result of withdraw since it has raised this exception during its execution (Section 5.2). MOP reifies the result of withdraw and notifies the meta-object MetaParticipant about the exception result by means of the method handleResult.

The action participants are synchronized and exception resolution process is accomplished based on the available meta-information. For instance, the meta-object MetaAction communicates with the meta-object responsible for maintain extra-information about the raised cooperating exceptions (Section 5.1). The meta-object MetaParticipant receives the resolved exception and then asks the eligible meta-object to search the handler for the resolved exception. Note that after the thread asks to start its activity within the action, all management activities are performed by the meta-object in a way that is transparent to the application.

Variants. Distributed Reflective Management. In this variant, the management of the cooperating thread groups is distributed. In this way, all information about the group must be held by each participant. Consequently, the MetaAction and Action classes are not necessary. The process of resolution and synchronization is performed in a distributed way. The work [Romanovsky et al., 1996] proposes an algorithm for coordinated exception handling in distributed object systems. Each participant must keep a copy of the algorithm and the management is performed by means of message exchange.

Non-Reflective Management. This variant transfers all management activities (exception resolution, synchronization, search and invocation of handlers) to the base-level Action and Participant classes. Even though such a variant does not need of a meta-object protocol to be implemented within a object-oriented language, application code is intermingled with explicit invocations of handlers, and procedures for exception resolution and synchronization. Consequently, the use of such a variant reduces reusability and readability of applications.

Known Uses. The work [Romanovsky et al., 1996] proposes a non-reflective and distributed variant of this pattern. This works proposes an algorithm for concurrent exception handling in distributed object systems. Exception resolution and the final synchronization is performed in a distributed way, and the information concerning the action must be held by each participant. Each participant must keep a copy of the algorithm and the management is performed by means of message exchange. However, the class Action is not necessary. Zorzo's coordinated atomic action framework [Zorzo et al., 1997] also provides software developers with a number of classes to structure their concurrent applications. However, it uses a non-reflective variant of the Concurrent Exception Handling Action Pattern. Programmers extend two classes of the framework in order to implement their concurrent cooperative actions. Both classes are similar to the classes Action and Participant regarding their responsibilities.

Consequences. Using Concurrent Exception Handling Action Pattern has the following consequences:

- **Uniformity.** The strategy for concurrent exception handling is a consistent extension of the general strategy for exception handling.
- **Transparency and Simplicity.** Management mechanisms for exception handling are performed transparently to the application. Programmers focus their attention on the identification of concurrent cooperative actions of their application.
- **Complexity Control.** This pattern allows programmers to define nested actions.
- **Readability. Reusability and Maintainability.** The application code is not intermingled with invocations of methods responsible for synchronization and exception resolution. As a consequence, it improves readability, which in turn improves reusability and maintainability.

6. ARCHITECTURE EMPLOYMENT AND IMPLEMENTATION ISSUES

The software architecture and the patterns proposed in this paper have been developed based on our study of a number of exception handling proposals [Garcia, 2000], and on our extensive work implementing dependable object-oriented systems [Garcia et al., 1999, Xu et al., 1995]. Since our software architecture is independent of programming language or exception handling mechanism, a wide range of dependable applications designers can employ it. A possible limitation of our approach is that it may not be easy to perform checks statically (Section 5.3). However, existing tools can help programmers to avoid mistakes: post- and pre-processors, parsers, syntax-oriented editors and macro-processing.

A prototype of our software architecture was built using the Java programming language without any changes to the language itself by means of a meta-object protocol called Guarana [Oliva and Buzato, 1998] developed at the Institute of Computing, UNICAMP, Brazil. Guarana like other meta-object protocols, supports the composition of meta-objects. That is, the definition of a special kind of meta-object, usually called composer, that delegates operations and results to
other meta-objects. Thus, base-level objects can be directly associated with a meta-object that encapsulates (composes) several quality requirements. This composition support and how our proposed software architecture has been designed allows us integrate easily the proposed software architecture components with other components responsible for other quality requirements such as security [Braga et al., 1998] and software fault tolerance [Ferreira and Rubira, 1998] into a meta-level software architecture [Beder and Rubira, 2000] that provides support for implementing general dependable collaboration-based designs.

This meta-level software architecture has been used to implement some experiments such as the Station Case Study [Beder et al., 2000]. This case study is a subsystem of the railway control system that deals with train control and coordination in the vicinity of a station. Trains transport passengers from a source to a destination station. Stations usually have several platforms on which trains can stop (no more than one train on each platform at a time). Trains can execute some join cooperative activity when they stop at the same station together; for example, passengers can change trains during this stop to make their journey faster. We assume that errors caused by faults (for example, a train do not stop) are detected and fault tolerance measures employed to restore normal computation. Sensors check whether trains stopped at this station, and whether they leave the station after the cooperative activity has finished.

We have used the Concurrent Exception Handling Action pattern to design the Station action (and its nested actions) that coordinates the execution of an activity concerned with cooperation between two trains calling at a particular station. In addition, we use the Handler pattern to structure the ExceptionalTrain class which implements handlers for the exceptions that can be raised by Station action participants, and the Exception pattern to define simple and concurrent (structured) exceptions.

7. RELATED WORK

Even though various object-oriented languages support exception handling facilities, very few (such as the Arche language [Issarny, 1993]) provide effective support for concurrent exception handling. Arche’s exception handling mechanism allows user-defined resolution of multiple exceptions amongst a group of objects that belong to different implementations of a given type; however, this approach is not generally applicable to the concurrent exception handling of multiple interacting objects of different types. The work in [Romanovsky, 1997] describes a scheme for concurrent exception handling based on atomic action structures for the Ada95 language. In this approach, application programmers have to implement an exception resolution function for each concurrent cooperative action. Programmers are responsible for implementing this resolution function. Application code is also intermingled with invocations of head processes which are procedures for synchronizing action participants exiting from the action.

Zorzo et al. [Zorzo et al., 1999] have developed an object-oriented scheme for implementing coordinated atomic actions [Xu et al., 1995] that provides software designers with a number of classes to structure their concurrent applications. Zorzo’s approach components are similar to the proposed software architecture components regarding their responsibilities. However, in this approach the management mechanism for exception handling is not performed transparently to the application and the application code is intermingled with invocations of methods responsible for synchronization and exception resolution. Besides, the normal code is amalgamated with the exceptional code.

Zorzo [Zorzo, 1999] has proposed an object-oriented scheme for implementing Dependable Multiparty Interactions (DMI). Each DMI is a group of multiparty interactions: one interaction for the DMI when there is no failures, i.e. basic interaction, and several interactions for dealing with exceptions that may be raised during the execution of the DMI (either during the basic interaction or during an exception handling interaction). The key idea for handling exceptions is to build DMIs out of unreliable multiparty interactions by chaining them together, where each multiparty interaction in the chain is the exception handler for the previous multiparty interaction in the chain. Even though such approach separates the normal code and exceptional code, the management mechanism for exception handling is not yet totally transparent.

The work of Hof et al. [Hof et al., 1997] describes an approach for exception handling based on meta-programming and computational reflection. Their implementation was carried out in a specific system but it could be implemented in other systems that support meta-programming. However, this approach does not support concurrent exception handling and its exception handling model is not designed in an object-oriented fashion.

8. CONCLUSIONS AND ONGOING WORK

The current lack of effective exception-handling techniques for constructing dependable object-oriented software motivated us to develop an exception handling software architecture. In this context, this paper makes the following contributions: (i) it presents a generic software architecture for incorporating exception handling into dependable object-oriented software; the architecture supports uniformly concurrent and sequential exception handling; specific applications reuse the exception handling facilities provided by the architecture’s components, allowing developers to focus their attention on the application-dependent functionality;
and (ii) it proposes a set of design patterns which refine the components of the architecture. The design patterns incorporate well-proved solutions and their micro-architecture achieves a clear and transparent separation of concerns between the application’s functionality and the exception handling services. In this way, designers of dependable systems can concentrate on the complexity inherent to their systems rather than on the intricacies of the exception handling facilities, easing the task of building dependable software with high assurance. We are currently applying our approach to incorporate abnormal code into third-party software components using wrappers and computational reflection. This extra code adapts the components to different exceptional situations which arise in the several contexts that they are reused.

Acknowledgments. This work has been supported by CNPq/Brazil (under grant 131945/98-0 for Alessandro Garcia. 141425/97-0 for Delano Beder and 351592/97-0 for Cecilia Rubira), and by CAPES/Brazil (under grant BEX553/99-0 for Delano Beder). Alessandro Garcia is also supported by the PRONEX project under grant 7697102900, and Cecilia Rubira is also supported by the FINEP and “Advanced Information Systems” Project (PRONEX-SAI-7697102200).

REFERENCES


