ELSEVIER

The Journal of Systems and Software 59 (2001) 197–222

The Journal of Systems and Software

www.elsevier.com/locate/jss

A comparative study of exception handling mechanisms for building dependable object-oriented software

Alessandro F. Garcia [a], Cecília M.F. Rubira [a], Alexander Romanovsky [b,*], Jie Xu [c]

[a] *Institute of Computing, University of Campinas (UNICAMP), Brazil*
[b] *Department of Computing Science, University of Newcastle, Newcastle upon Tyne, NE1 7RU, UK*
[c] *Department of Computing Science, University of Durham, UK*

Received 17 October 2000; received in revised form 31 December 2000; accepted 7 March 2001

Abstract

Modern object-oriented systems have to cope with an increasing number of exceptional conditions and incorporate fault tolerance into systems' activities in order to meet dependability-related requirements. An exception handling mechanism is one of the most important schemes for detecting and recovering errors, and for structuring fault-tolerant activities in a system. The mechanisms that were ill designed can make an application unreliable and difficult to understand, maintain and reuse in the presence of faults. This paper surveys various exception mechanisms implemented in different object-oriented languages, evaluates and compares different designs. A taxonomy is developed to help address 10 basic technical aspects for a given exception handling proposal, including exception representation, external exceptions in signatures, separation between internal and external exceptions, attachment of handlers, handler binding, propagation of exceptions, continuation of the control flow, clean-up actions, reliability checks, and concurrent exception handling. Practical issues and difficulties are summarized, major trends in actual languages are identified, and directions for future work are suggested.

Keywords: Exception handling; Exception mechanisms; Object-oriented languages; Object-oriented programming; Software fault tolerance

1. Introduction

Modern object-oriented systems often involve complex concurrent and interacting activities. Such systems have to cope with an increasing number of exceptional conditions. With such systems growing in size and complexity, employing exception handling techniques while satisfying dependability-related requirements, such as maintainability and reusability, are still deep concerns to designers of dependable object-oriented systems. An exception handling mechanism provides support for detecting and recovering errors, and structuring fault-tolerant activities in a system. The mechanism must be designed properly and an ill designed scheme can make an application unreliable and difficult to understand, maintain and reuse in the presence of faults.

Engineers who develop complex object-oriented systems often use exception mechanisms to implement exception handling activities for dealing with erroneous situations. In these systems, the code devoted to error detection and handling is usually both numerous and complex. As a consequence, up to two-thirds of a program can be for error handling (Cristian, 1989; Gehani, 1992). In this context, the design of an exception mechanism should be simple and easy to use, and provide explicit separation between the normal and exceptional code. Ideally, dependable object-oriented systems using the exception mechanism should be therefore easy to understand, maintain and reuse. A number of exception mechanisms exist in both experimental and actual object-oriented programming languages. Realistic examples of object-oriented languages include Java (Arnold and Gosling, 1998), C++ (Koening and Stroustrup, 1990), Modula-3 (Nelson, 1991) and Eiffel (Meyer, 1988).

The purpose of our paper is to investigate the applicability of the existing exception mechanisms of object-oriented programming languages for developing dependable object-oriented software with effective

* Corresponding author. Tel.: +44-191-2228135; fax: +44-191-22282232.

E-mail addresses: afgarcia@inf.puc-rio.br (A.F. Garcia), cmrubira@ic.unicamp.br (C.M.F. Rubira), alexander.romanovsky@newcastle.ac.uk (A. Romanovsky), jie.xu@durham.ac.uk (J. Xu).

0164-1212/01/$ - see front matter
PII: S0164-1212(01)00062-0

quality attributes. The major contributions of this paper are:

(i) the definition of a taxonomy which is used to discuss 10 functional aspects of an exception mechanism and to distinguish one mechanism from another;
(ii) the presentation of a comprehensive survey of existing exception mechanisms implemented in object-oriented languages;
(iii) the comparison and evaluation of the investigated mechanisms as well as the identification of primary limitations in applying them in practice to develop dependable object-oriented systems;
(iv) the definition of a set of adequate design solutions and an ideal exception handling model for exception mechanisms suitable for dependable object-oriented software; and
(v) the identification of current trends related to exception handling and dependable object-oriented software.

The remaining part of this paper is organized as follows. Section 2 gives a brief description of exception handling within a framework for facilitating software fault tolerance. This section also introduces exception mechanisms as well as difficulties related to concurrent exception handling. Section 3 describes our proposed taxonomy for classifying different design approaches to exception handling mechanisms. Section 4 discusses in more detail exception handling mechanisms implemented in various object-oriented languages. Section 5 assesses the relative advantages, disadvantages and general limitations of these mechanisms based on our established taxonomy. Section 6 presents a set of important design criteria and an ideal exception handling model for building dependable object-oriented systems. Section 7 discusses open problems and directions for future work. Finally, Section 8 presents some concluding remarks.

2. Exception handling and fault tolerance

2.1. Terminology

A software system consists of a number of components, which cooperate under the control of a design to service the demands of the system environment (Lee and Anderson, 1990). The components and the system environment may be also viewed as systems. The design can be considered as a special component that defines the interactions between components and establishes connections between components and the system environment. In principle, the dynamic behaviour of a software system is characterised by the series of internal states which the system adopts during its processing. Certain elements of an internal state will coincide with the interface between the system and its environment, and these form the external state of the system by which the external behaviour of the system can be realised. Under normal processing conditions, the system will advance from one valid internal state to the next by means of a valid transition. However, an erroneous transition can be caused by one or more components or by the ill design of the system.

A system *failure* occurs when the service delivered by the system deviates from what the system is aimed at (e.g., specification). An *error* is that part of the system internal state which is liable to lead to subsequent failure. A *fault* is the (hypothesised) cause of an error. A fault can be a physical defect, imperfection or flaw that occurs within a hardware or software component. In particular, a fault which occurs in the environment of a system is called an environmental fault; it may cause an error in the system.

A dependable software system often uses supplementary techniques for tolerating the manifestations of faults in its components and thereby avoids failures of the entire system. These techniques are usually based on the provision of software redundancy. However, software redundancy increases the size of the systems and introduces additional complexity into them. Dependable software systems should therefore be well-structured in order to control such additional complexity. Each component within a system should be able to return well-defined responses, and incorporate a clear separation between normal and exceptional activities. In this sense, exceptions and exception handling provide a desirable framework for structuring fault-tolerant activities in such a system.

A software component receives service requests and produces responses. If the component cannot satisfy a request, it will return an *exception*. Thus, responses from the component can be separated into two distinct categories: *normal* and *exceptional responses*. Exceptions can be further classified into three different categories: (i) *interface exceptions* which are signalled in response to a request which did not conform to the component's interface; (ii) *failure exceptions* which are signalled if a component determines that for some reason it cannot provide its specified service; (iii) *internal exceptions* which are exceptions raised by the component in order to invoke its own internal fault tolerance measure. It is important to notice that an exception is *raised* within a component, but *signalled* between components. The interface and failure exceptions can be regarded as *external exceptions* since the actions that handle such exceptions are external to the component that signalled them.

The activity of a component can be also divided into two parts: *normal activity* and *abnormal* (or *exceptional*) *activity* (Fig. 1). The normal activity implements the component's normal services while the exceptional

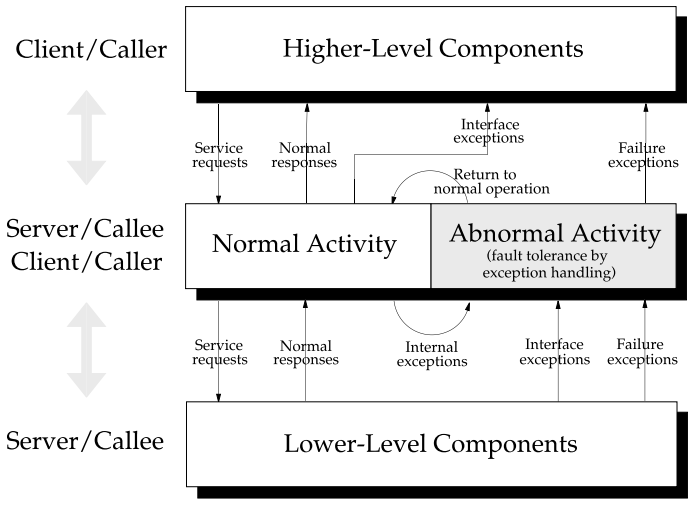

Fig. 1. Idealised fault-tolerant component.

activity provides measures that cope with the faults that cause the exceptions.

Ideally, a dependable system should be composed of a set of *idealised fault-tolerant components*. An idealised component is responsible for handling the exceptions raised during its normal activity and any exceptions signalled by lower-level components. Whenever a (server/callee) component cannot handle an exception it raises, the exception will be signalled to the (client/caller) higher-level component. After the exception is handled, the system may return to its normal activity.

Developers of dependable systems usually refer to faults as exceptions because they are expected to manifest themselves rarely during the component's normal activity. Exception handling mechanisms have been developed for many high-level programming languages and allow software developers to define exceptions and to structure the exceptional activity of software components by means of handlers. The handlers of a program constitute its exceptional activity part. The exception handling mechanism is responsible for changing the *normal control flow* of a program to the *exceptional control flow* when an exception is raised during its execution. Exception mechanisms are either built as an inherent part of the language with its own syntax, or as a feature added on the language through library calls (Drew and Gough, 1994).

In the context of programming languages, exceptions can be classified into two types (Goodenough, 1975; Lang and Stewart, 1998): (i) *user-defined*, and (ii) *predefined*. User-defined exceptions are defined and detected at the application level. Predefined exceptions are declared implicitly and are associated with conditions that are detected by the language's run-time support, the underlying hardware or operating system. Exceptional events pre-defined by a particular exception mechanism may differ from one language to another and depend on general decisions made by the language designers.

An exception can be raised at run-time (an *exception occurrence*) during the normal execution of an operation (or method). A *signalling statement* is the statement being executed when an exception occurrence is detected. The code block containing the signalling statement is referred to as the exception *signaller* (Fig. 2). When an exception occurrence is detected by an assertion statement or by the underlying virtual machine, the exception mechanism will search and invoke an *exception handler* (or simply *handler*) to deal with the raised exception. The handler is the part of application's code that provides application-specific measures for handling the raised exception. Some extra information about an exception occurrence, such as its name, description, location and severity (Lang and Stewart, 1998), is usually required by the corresponding handler. Extra information is passed either explicitly by the signaller or implicitly by the exception handling mechanism.

Handlers are attached to a particular region of the normal code which is termed *protected region* or *handling context*. Fig. 2 illustrates three protected regions. Each protected region can have a set of attached handlers. If an exception is raised in a protected region, the normal control flow is deviated to the exceptional control flow. When the raised exception is an internal one, the exception mechanism seeks for a *local handler* which is attached to the protected region (the signaller). Ideally, all internal exceptions should to have local handlers. When an external exception is raised, this exception is signalled to the operation caller to inform it about the impossibility to produce the required result. After a suitable handler is found, invoked and executed by the mechanism, the computation will be returned to the normal control flow.

Fig. 2 pictures one of the possible scenarios for the operation of an exception handling mechanism. Internal exceptions are represented by `e1`, `e2`, `e3`, ..., while external ones are represented by `E1`, `E2`, `E3`, ... The

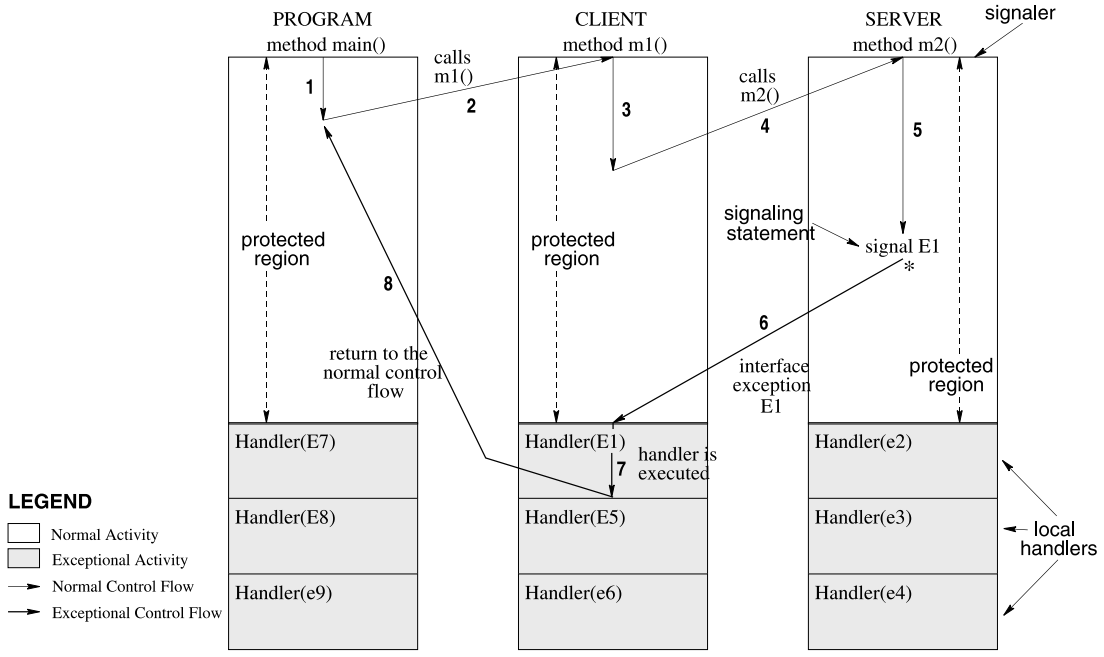

Fig. 2. A possible scenario for the operation of an exception mechanism.

interface exception E1 is detected during the normal execution of m2. The exception mechanism signals E1 to the caller, i.e. the method m1 (arrow 6). The exception mechanism then finds and invokes the appropriate handler at the context of the caller (arrow 7), and returns the system to the normal control flow (arrow 8).

2.2. Exception handling in concurrent OO systems

In many object-oriented software systems, there may be a number of processes (threads) running concurrently. There are different ways of dealing with concurrency in object-oriented systems. In this paper, we define a clear distinction between *objects* and *threads*: threads are agents of computation that execute methods on objects (which are the subjects of computation). However, exception handling, and consequently the provision of fault tolerance, is much more difficult in concurrent object-oriented systems than in sequential ones. Exception mechanisms used in sequential programs cannot be applied to concurrent software without appropriate adjustment due to new difficulties introduced by concurrent exception handling.

From the standpoint of fault tolerance, the implementation of an exception mechanism for concurrent object-oriented systems is an interesting challenge due to cooperative concurrency (Campbell and Randell, 1986; Lee and Anderson, 1990). Different threads of a system can be cooperating for executing some system's task. Threads are said to be *cooperating* when they are designed collectively and have shared access to common objects that are used directly for communication between the threads (Lee and Anderson, 1990). Erroneous information may spread directly or indirectly through inter-thread communication. As a consequence, the handling of an exception should involve all concurrent threads participating in a cooperative activity. Therefore, the cooperating threads of a concurrent system must be designed and controlled very carefully in order to avoid that erroneous information spreads unexpectedly through the whole system (Campbell and Randell, 1986).

An approach to using exception handling in such systems was introduced in (Campbell and Randell, 1986). It extends the well-known atomic action paradigm. Atomic actions are a comprehensive way of structuring the behaviour of concurrent systems. These actions are units of inter-thread cooperation and their execution is indivisible and invisible from the outside. The activity of a group of threads participating in a cooperation constitutes an atomic action if there are no interactions between that group and the rest of the system for the duration of the activity (Lee and Anderson, 1990). Complex interactions between the participating threads of an atomic action can be coordinated within that action, including necessary activities for concurrent exception handling (Campbell and Randell, 1986). When one of the cooperating threads has raised an exception, error recovery should proceed in a coordinated way by triggering handlers for the same exception within all the threads (Campbell and Randell, 1986; Xu et al., 1995a).

In principle, a group of interacting threads, the *action participants*, cooperate within the scope of an atomic action. A set of exceptions is associated with the action.

Each participant in the action has a set of handlers for (all or part of) these exceptions. The entries of participants in the action may be asynchronous but they have to leave the action synchronously in order to guarantee that no information is smuggled into or from the action. When an exception has been raised in any of the participants inside an action, all action participants have to participate in the error recovery. Handlers for the same exception in all of the participants will then be called and cooperate to recover the action. The participants can leave the action on three occasions. First of all, this happens if there have been no exceptions raised. Second, if an exception has been raised, and the called handlers recovered the action. Third, they can signal a *failure* exception to the containing action if an exception has been raised and it has been found that there are no appropriate handlers for the exception or that recovery is not possible.

Furthermore, due to nature of concurrent systems, it is possible that various exceptions may be raised concurrently by cooperating threads. In this way, a mechanism for *exception resolution* is an essential part of concurrent exception handling. The paper (Campbell and Randell, 1986) describes a model for exception resolution called *exception tree* which includes an exception hierarchy. This model allows finding the exception that represents all exceptions raised concurrently. This tree includes all exceptions associated with an atomic action and imposes a partial order on them in such a way that a higher-level exception has a handler capable of handling any lower-level exception. If several exceptions are raised concurrently, the resolved exception is the root of the smallest subtree containing all of the exceptions.

The need for concurrent exception handling in object-oriented systems has been recognised by many researchers because it would make the error handling simpler, uniform and less error prone. In particular, the need for concurrent exception handling has been identified in a number of practical examples and systems in different application domains such as banking (Garcia et al., 2000; Xu et al., 1995a), office automation (Xu et al., 1995a), sales control systems (Xu et al., 1995a), software development environments (Xu et al., 1995a), and production cell control systems (Randell and Zorzo, 1999; Romanovsky et al., 1998; Zorzo et al., 1999).

3. A taxonomy for object-oriented exception mechanisms

There is a number of design issues needed to be considered while building exception mechanisms for object-oriented software. However, the chosen solution for designing each of them varies from mechanism to mechanism. This section presents a taxonomy which identifies the common design issues of exception mechanisms, and classifies different design solutions. The taxonomy was developed based on the set of analysed exception mechanisms (Section 4), and on previous studies reviewed (Drew and Gough, 1994; Lang and Stewart, 1998).

We classify the design issues of an exception handling scheme into 10 aspects of interest: (i) exception representation, (ii) external exceptions in signatures, (iii) separation between internal and external exceptions, (iv) attachment of handlers, (v) handler binding, (vi) propagation of exceptions, (vii) continuation of the control flow, (viii) clean-up actions, (ix) reliability checks, and (x) concurrent exception handling. In the following, we discuss each aspect in turn.

A1. *Exception representation*. Exceptions that can be raised during a system's execution must be represented internally within this system. Exceptions can be represented as: (i) *symbols*, (ii) *data objects* or (iii) *full objects*. The representation of exceptions as symbols is a classical approach in which exceptions are strings or numbers. Raising an exception sets the corresponding string variable (or integer variable) and returns the control to the caller that is in charge of testing the variable.

In the second and third solutions, exceptions are organised hierarchically as classes; when an exception is raised, an instance of an *exception class* (an *exception object*) is created and passed as a parameter to the corresponding handler. Therefore, exceptions are first-class objects. However, such solutions differ in how exceptions are raised. In the second solution, exceptions are raised by calling a keyword of the language. In the third solution, exceptions are raised through invoking a method raise on the exception object. In this last case, the exception is a standard object that receives messages since the behaviour specific to exception raising is defined as a method on an exception class (Lacourte, 1991). In addition, specific behaviour to the continuation of the control flow (see below) can also be defined as methods on exception classes. Note that in the second solution the aim of exception objects is just to hold data (despite the possible definition of methods on them).

A2. *External exceptions in signatures*. A method may either terminate normally or exceptionally by signalling an external exception. However, it should exit from the points defined in the method's signature that explicitly declares the list of external exceptions that might be signalled by the method (Lang and Stewart, 1998) (Fig. 3).

There are different design solutions for this issue in existing exception mechanism proposals. In some exception mechanisms, the declaration of external exceptions in the signature is *compulsory* – an attempt to propagate to the invoker an exception that is not in the exception list causes either a compiling error or the raising of a predefined exception at run-time. In other

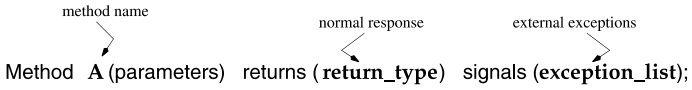

Fig. 3. A Method's signature with external exceptions.

mechanisms, the declaration is either *optional* or *unsupported*. There is also a *hybrid* approach – some exceptions must be listed in signatures while others may not be listed.

A3. *Separation between internal and external exceptions*. The execution of a component's service can be completed either normally (i.e., without any external exceptions signalled, although an internal exception might have been raised and successfully handled) or exceptionally, with an external exception signalled (either from the normal code or from an internal exception handler). Some exception handling mechanisms allow a clear separation of internal and external exceptions, while others do not make any distinction between them. This separation should be enabled by means of raise and signal operations. Superficially, both operations may appear to perform similar functions since both are used to initiate fault tolerance activities by means of exception handling. However, signal allows a component to signal an external exception to the system of which it is a part, invoking the handler within that system; raise enables a component to raise an internal exception in order to invoke a handler within that component.

A4. *Attachment of handlers*. Handlers can be attached to different protected regions, such as: (i) a *statement*, (ii) a *block*, (iii) a *method*, (iv) an *object*, (v) a *class* or (vi) an *exception*.

Statement (or block) handlers are attached to a statement (or a statement block), allowing context-dependent responses to an exception. The block is usually defined by means of keywords of the language; the protected region starts with a specific keyword and ends with another keyword of the language. Method handlers are associated with a method; when an exception is raised in one of the statements of the method, the method handler attached to this exception is executed. Exception mechanisms with handlers attached to blocks consequently also provide support for method handlers since a block may be defined as a method. Object handlers are valid for particular instances of a class; that is, each instance has its own set of handlers. Object handlers are usually attached to object variables in their declarations. Class handlers are attached to classes, allowing the software developers to define the common exceptional behaviour for a class of objects. Handlers attached to exceptions themselves will be always invoked if no more specific handlers can be found. They are the most general handlers, and must be valid in any part of the program, independent of the execution context and object states. For instance, such a handler could print an error message or make a general correction action.

A5. *Handler binding*. There are three different design solutions for binding handlers to exception occurrences: (i) the *static* approach, (ii) the *dynamic* approach, and (iii) the *semi-dynamic* approach. In the static approach, a handler is statically attached to a protected region, and this handler is used for all possible occurrences of the corresponding exception during the execution of that protected region. The handler binding is independent of the control flow of the program, and hence there is no run-time search for binding handlers to exception occurrences.

In the dynamic approach, the binding depends upon the control flow of the program. As a consequence, this approach determines at run-time which handler should be used for a given exception occurrence. The handler cannot be determined at compile-time. Exception handlers are defined dynamically in the executable statements of programs by executing a statement making a handler available for a particular exception. In PL/I, the binding operation is performed dynamically by means of the statement ON. A statement ON specifies the binding of a handler to a specific exception, and it stays in effect until either a new statement ON for that exception is executed or the block in which it occurs exits.

The semi-dynamic binding is a hybrid model that combines the two previous approaches. Local handlers can be bound statically to the signaller. If a handler is not attached to the raised exception in the context of the signaller, a dynamic approach is employed to find a suitable handler. First, handlers attached to enclosing protected regions are searched dynamically. If none is found, the exception mechanism then signals the exception to the caller. The call chain of method invocations and protected regions is therefore traversed backwards until a statement or another protected region is found in which a handler for that exception is attached. The example in the Fig. 2 illustrates the semi-dynamic approach. Assume that the exception mechanism does not find a local handler attached statically to the signaller, i.e. the method m2. The exception mechanism then proceeds the search by taking the invocation history into account. A handler is then found at the context of m1.

A6. *Propagation of exceptions*. If no local handler is defined for an exception which has been raised, the exception can be propagated to the caller of the method raising the exception. In fact, the caller often knows

what effect the operation has to achieve and how best to respond to exceptions (Cui and Gannon, 1992; Schwille, 1993). If no handler is found for the exception within the caller, the exception can be propagated to higher-level components other than its immediate caller. There are two design solutions for exception propagation: (i) *explicit* propagation, and (ii) *automatic* (or *implicit*) propagation.

In the first case, the handling of signalled exceptions is limited to the immediate caller; however, the raised exception or a new exception can be signalled explicitly within a handler (attached to the caller) to a higher-level component. For this reason, the exception mechanisms that adopt such an approach are called *single-level* (Liskov and Snyder, 1979). If a signalled exception is not handled at the caller's context, either a predefined general exception is further propagated automatically, or the program is terminated. In the second case, if no handler is found for the exception within the caller, the exception is propagated automatically to higher-level components until a handler can be found; that is, an exception can be handled by components other than its immediate caller. As a consequence, the exception mechanisms that incorporate this design solution are termed *multi-level* (Liskov and Snyder, 1979).

Exception propagation is closely related to the issue of handler binding. Exception mechanisms that implement static binding allow no exception propagation since the binding is done at compile time and the chain of invokers is ignored. Semi-dynamic and dynamic binding operations are performed at run-time and thereby allowing exception propagation.

A7. *Continuation of the control flow*. After an exception is raised and the corresponding handler is executed, the system should continue its normal execution. There is an issue concerning the semantics which determine the continuation of the control flow, i.e., where normal execution proceeds. There are at least two possible design solutions, which correspond to different styles of continuation of the control flow: (i) the *resumption* model, and (ii) the *termination* model. In the resumption model, the execution has the capability to resume the internal activity of the signaller after the point at which the exception was raised. In the termination model, the activity of the component which raised the exception cannot be resumed; conceptually, this means that its activity is terminated.

There are some variations of the termination model. Such variations can be classified into at least three different ways with their respective semantics: (i) *return* – terminate the signaller and direct control flow to the statement following the protected region where the exception was handled; (ii) *strict termination* – terminate the program and return control to the operating environment; and (iii) *retry the signaller* – terminate the signaller and retry it to attempt to complete the required service in the normal way.

Fig. 2 pictures an example of termination with return semantics. The execution of m2 (the signaller) is terminated, the handler is executed, and the normal control flow is directed to the statement following the protected region where the exception is handled. Then, execution continues at the method main since the method m1 is the protected region where the exception is handled. If the specified model of the continuation of control flow were resumption, the handler would be executed and normal execution would resume the internal activity of the signaller after the point at which the exception was raised. This return point is indicated as $*$ in Fig. 2.

A8. *Clean-up actions*. Components of a program should be kept in a consistent state, regardless of whether the code completes normally or is interrupted by an exception. In this sense, it is required to do some clean-up action to keep the program in a consistent state before the termination of the component. The clean-up code may either restore the component to a possible consistent state, undo some undesirable effect or release allocated resources. Clean-up actions may be supported by the design of particular exception mechanisms in three different ways: (i) *use of explicit propagation*, (ii) *specific construct*, and (iii) *automatic clean-up*.

In the first approach, the explicit propagation is used for performing some clean-up actions before the component is terminated. When an exception occurrence is detected, if it cannot be handled by the signaller and has to be propagated, then the clean-up action should be specified within the corresponding handler before the statement that propagates the exception.

The second method provides a construct which is executed whenever the protected program unit exits. The clean-up code is attached to the protected program unit and this code is executed whether an exception is raised or not. If no exceptions are raised in the protected region, the attached clean-up code is executed after the protected region ends. However, if an exception is raised in the protected region, control is transferred immediately to the statements devoted to cleanup.

The third solution is based on the premise that the exception mechanism knows what should be cleaned up before the component is terminated. The exception mechanism itself performs automatically necessary clean-up actions and the application developer is not responsible for such actions.

A9. *Reliability checks*. Reliability checks test for possible errors introduced by the use of an exception mechanism. A number of issues can be checked by the exception mechanism itself, such as (Yemini and Berry, 1985): (i) checking that each exception is signalled with

the correct set of actual parameters; (ii) checking that each handler for an exception is defined with the correct set of formal parameters; (iii) checking that only those exceptions that are defined by a signaller are signalled by the signaller, in effect forcing the explicit propagation; and (iv) checking that all exceptions that can be raised in a given scope are handled in that scope. We classify the design approaches regarding reliability checks into two design solutions: (i) *static checks*, and (ii) *dynamic checks*. Static checks are performed by the compiler while dynamic checks are performed by the run-time system. Static checking depends on the use of exception interface, static binding and representation of exceptions as objects. When exceptions are not declared in the external interface of their signallers or are not typed, there is very little that can be checked at compile time. Some exception mechanisms do not provide any support for static checks, while other ones perform both static and dynamic checks.

A10. *Concurrent exception handling*. If concurrent programming is supported by the underlying programming language, one or more exceptions can be raised concurrently during a cooperative activity (Section 2.2). In this way, exception mechanisms should provide some support for concurrent exception handling. The design approaches to concurrent exception handling can be classified into at least three possible design solutions, which correspond to different support levels: (i) *unsupported*, (ii) *limited*, and (iii) *complete*.

Exception mechanisms that implement the second approach provide only limited support for concurrent exception handling. A special exception (signal) is used to notify the threads involved in a cooperation when an exception is raised in one of the cooperating threads. In this way, exceptions can be handled by more than one thread in a coordinated manner. However, facilities that allow using atomic actions with exception handling are not provided, and the exception resolution process is also left to application programmers.

The third approach provides comprehensive support for concurrent exception handling. Explicit facilities are provided for using atomic actions with concurrent exception handling. Software designers can concentrate on defining cooperative activities, exceptions and handlers, which are application-specific issues. The exception mechanism provides: (i) final synchronisation of the action participants, (ii) support for exception resolution, and (iii) invocation of the different handlers attached to the action participants.

4. Exception handling in various OO languages

In the 1970s, exception mechanisms were developed specifically for procedural programming languages. PL/I (Maclaren, 1977) was the first widely used language which provided application programmers with linguistic constructs for exception handling. Unfortunately, this mechanism was rather complex and difficult to use. The CLU exception mechanism (Liskov and Snyder, 1979) overcame some of the problems of the PL/I exception mechanism and introduced an exception handling model more suitable for implementing dependable software. In the 1980s, object-oriented languages opened to developers a new way of system design based on novel approaches to system modularisation and reuse. Since then, exception handling mechanisms have been integrated into mainstream object-oriented languages such as Java (Arnold and Gosling, 1998), Modula-3 (Nelson, 1991) and Eiffel (Meyer, 1988). In this section, we will briefly overview a series of existing exception mechanisms found in different object-oriented languages. The taxonomy proposed in Section 3 is used to evaluate and compare these mechanisms. Table 1 in Section 5 summarises the characteristics of each exception mechanism presented in this section.

CLU (Liskov and Snyder, 1979) is the first language to offer an exception mechanism suitable for implementing fault-tolerant software. In fact, its primary purpose is to support construction of software modules which are capable of behaving reasonably in a wide variety of circumstances (including the abnormal ones). This exception mechanism is based on a simple exception handling model which supports the development of well-structured programs. Consequently, we view this model as fundamental for our study.

The exception handling model of CLU. In the CLU exception handling model, exceptions are defined as symbols. However, a set of typed parameters can be used to pass information about the exception from the signaller to the handler. CLU clearly separates internal and external exceptions by means of the signal and exit operations. The CLU exception mechanism is described as single-level since an exception raised by a procedure has to be handled by its immediate caller. However, the caller may explicitly resignal the exception to its caller. The CLU procedure definition must include external exceptions. Handlers can be attached to any statement in a CLU procedure by clause except, which has the following syntactic form:

```
...
statement except
when E1: ...
when E2: ...
...
```

If possible, handlers are declared at the end of the procedure because the placement of handlers in an individual statement can reduce the code readability. However, exception handlers can be interleaved with the normal code on a per statement basis. If a statement

calls a procedure which signals an exception, but the statement has no attached handler for the exception, then the exception is propagated automatically to a progressively larger static scope within the procedure. If a handler is found in the procedure, it is executed. Otherwise, the predefined default exception failure is raised and control returns to the caller. This exception is the only one which can be implicitly propagated in this model. CLU procedures that raise an exception are normally terminated. CLU adheres to the termination model with return semantics: when handler execution is completed, the control is passed to the statement following the statement to which the handler is attached. The CLU exception mechanism neither provides any specific construct for defining clean-up actions nor supports concurrent exception handling.

4.1. Exception handling in Ada 95

Ada 95 (Taft and Duff, 1997) is a fully object-oriented language which has upward compatibility with Ada. The exception mechanism of Ada 95 is basically the same as in Ada; it was not revised to make it suitable for object-oriented programming. Ada exceptions are originally symbols and are not declared in the external interface of procedures (methods). There is no distinction between internal and external exceptions. However, some new features make using Ada 95 exceptions much simpler: (i) an exception can be raised with a message which can be analysed during handling; (ii) a new type ExceptionOccurrence is introduced to represent each occurrence of the exception together with a function converting the variable of this type into a string; (iii) another function, of the string type, is provided to return the name of the raised exception.

Handlers can be attached to blocks, procedures or packages. Handlers are placed together in the exception clause, which must be located at the end of the protected region, as below:

```
begin
... -- protected region -- ...
exception when E1 ⇒ ... -- handler -- ...
          when E2 ⇒ ... -- handler -- ...
          when others ⇒ ... -- all-encompass-
          ing handler -- ...
end;
```

Ada allows a special default handler to be defined using the when others construct. This handler catches all exceptions for which no specific handlers are provided.

The Ada 95 exception handling model supports semi-dynamic binding and automatic propagation of all unhandled exceptions. Therefore, this mechanism is multi-level as opposed to the single-level mechanism of CLU. It adopts the termination model with return semantics. An exception can also be propagated explicitly by a component reraising the exception within the handler. Therefore, explicit propagation can be used to perform any final clean-up actions before signalling the exception. All-encompassing default handlers can also be used to do it. However, no explicit support is provided for clean-up actions. Since exceptions are not typed or declared in the external interface of their signallers, there is very little that can be checked by compilers. Obviously, programmers cannot raise or handle an exception if it is not declared since the compiler detects this. In fact, Ada 95 exceptions are very unsafe.

Ada 95 allows attaching handlers to tasks. In some situations handlers may be called in several concurrent tasks when an exception has been raised in one of them. However, this language has a limited form of concurrent-specific exception propagation: an exception is propagated to both the caller and callee tasks only if it is raised during the Ada rendezvous. Such exception mechanism is not directly applicable for systems that contain complex cooperative concurrency.

4.2. Exception handling in Lore

An object-oriented exception mechanism has been designed by Dony (1988, 1990); it has been implemented in Lore, an object-oriented language dedicated to knowledge representation. An important characteristic of this mechanism is the object-oriented representation of exceptions: exceptions are full objects. External exceptions may be defined in the method signature using the signals clause. However, there is not any distinction between propagation or handling of internal and external exceptions.

The exception mechanism ensures explicit separation between normal and exceptional code since handlers are ordinary methods of a specific class named protected-handler. Handlers can be attached to statements, classes and exception classes. Handlers attached to classes are called default handlers. When an exception is raised, the handler search proceeds as follows: (i) first handlers attached to statements that dynamically include the signalling one are searched, and the search stops as soon as a handler, whose parameter type is a supertype of the signalled exception, is found; (ii) if none is found, the system tries to find a default handler attached to the class or upper classes of the signalling active object; (iii) if none is found, the system looks for a default handler attached to the signalled exception itself; (iv) if none is found, the exception is propagated automatically to the operation caller, and the sequence of steps is again repeated. Exceptions can also be propagated explicitly.

This exception mechanism is more flexible than the CLU one. Two policies for continuation of control flow are provided: resumption and termination (with return semantics). The chosen policy is defined as a method of an exception class from which all exception classes

inherit. The methods that define such policies are invoked within handlers. When handlers do not explicitly choose one of these options, the predefined exception ExceptionNotHandled is signalled. This approach is interesting because it integrates exceptions into the standard invocation mechanism. The Lore exception mechanism has explicit support for specifying clean-up actions. It provides the when-exit construct allowing programmers to attach clean-up actions to expressions. Lore provides no support for concurrent programming or concurrent exception handling.

4.3. Exception handling in Smalltalk-80

As one of the earliest widely available object-oriented languages, Smalltalk (Goldberg and Robson, 1983) attracted much attention in 1980s. An exception in Smalltalk-80 is a selector but not a first-class object. Each selector specifies the operation name. Thus a exception selector cannot own any characteristics, cannot be inspected, modified or upgraded (Dony, 1990). In order to signal run-time exceptions, the Smalltalk evaluator sends a message corresponding to the current exception to the current object. Therefore handlers are methods referenced by exception selectors, and they only can be attached to classes. Thus, exceptions raised by methods defined on a class are handled within that class. In Smalltalk, exceptions cannot be propagated to operation callers, this is why external exceptions are not part of a method signature.

Smalltalk-80 has no static type checking. A method is compiled successfully if it has no syntax errors or undeclared variables; all type errors occur at run-time. For this reason, most run-time errors (exceptions) in the Smalltalk environment occurs when a message for which no method exists arrives to an object. In this case, the run-time system sends a special message doesNotUnderstand: to the receiver, this message carries the message selector and arguments of the original message as arguments. The search for the doesNotUnderstand: selector thus follows the same path as the search for the first selector. In this way, a user is given an opportunity to define methods doesNotUnderstand: in the visited classes, in order to customize exception handling or catch errors. If the programmer does not do this, an error is signalled anyway by the standard method doesNotUnderstand: of the class Object (all application classes are derived from Object). In this case a notifier containing some information about the error appears on the screen; this provides a feature for invoking the debugger. Unlike CLU, Smalltalk supports both resumption and termination. Two variants of the termination model are supported: return and retry. The programmer may specify clean-up actions within the ensure block which is always executed after the protected block, no matter if an exception has occurred or not. The mechanism does not support concurrent programming and concurrent exception handling.

4.4. Exception handling in Eiffel

The exception mechanism of the sequential language Eiffel (Meyer, 1988, 1992) is integrated with the notion of design by contract. The main purpose of this feature is to support this methodology. Classes (and their methods) establish contracts with their clients by specifying assertions: pre- and post-conditions, and invariants. The loop invariants and check instructions can be specified as well. Exceptions are defined as the violation of assertions during the execution of the associated code, and they are always raised implicitly. An Eiffel exception is a typed entity which has an integer value and a string tag. External exceptions cannot be described in method signatures, and there is not any distinction between internal and external exceptions. Handlers can be attached to a method body or to a class by means of the rescue clauses. Application programmers use if or case to separate handlers of different exceptions in this clause.

When an exception occurs during the execution of a method, its execution is stopped and the corresponding handler is executed. Within a handler, the exception can be determined by checking a predefined variable called exception. If no handler is defined, the method is said to fail and the exception is propagated implicitly to the caller (this is called *organized panic*). Therefore, Eiffel design adopts automatic exception propagation as the default behaviour.

The Eiffel exception mechanism supports retry, which is a variation of the termination model. Handlers try to restore the class invariant by retrying the method if the pre-condition still holds. In this way, a routine may either succeed or fail (intermediate results are not possible). So the raising of an exception means the failure of a software component which has been unable to terminate in a normal manner. If a routine fails then the rescue clause executed is executed (the retry clause is not called). The rescue clause has to guarantee that the invariant holds. No support is provided in Eiffel for concurrent programming or concurrent exception handling.

4.5. Exception handling in Modula-3

In Modula-3 (Nelson, 1991) exceptions are defined as symbols. However, exceptions can have parameters. The external exceptions are declared in the method signature using the raises clause; this facilitates both static checking of raising unlisted exceptions and dynamic checking of exception occurrences that were not explicitly raised. Both, modules and interfaces can import interfaces, but only modules can export interfaces. A procedure in an exported interface can be re-declared in

a module only to supply a body. There is a simple rule for procedure re-declaration with respect to exceptions: the raises clause of the exported declaration must include all exceptions from the raises clause of the re-declared procedure: one can delete some/all of the exceptions but not add new ones. However, there is not any further distinction between internal and external exceptions with respect to their raising and handling.

Interface exception specification is compulsory in Modula-3. Any attempt to propagate an exception which is not in the raises list of the procedure is treaded as a run-time error. In particular, no exception can be propagated outside the procedure if the raises clause is omitted. Handlers can be attached to any block by means of the try... except construct:

```
try ... -- protected region -- ...
except ... -- handlers -- ...
else ... -- all-encompassing handler -- ...
```

In Modula-3 handler binding is semi-dynamic. If during the execution of the try block (the protected region), an exception occurrence is detected, the execution stops and control is passed to the corresponding handler. If no handler is found, and the else part is present, the control flow is deviated to this part. The else part implements an all-encompassing handler, i.e., a single default handler attached to the protected region to handle any exception for which no specific handler is provided (similar to the when others construct in Ada 95). If there is no else part, a handler is sought in the statically enclosing protected region (which might be defined by the try... except construct nested in the try block). If no handler is found there, the exception is automatically propagated, and the search continues in the context of the calling procedure. If no handler can be found, the Modula-3 run-time system halts the program with a warning error message. To summarise, Modula-3 adopts the automatic propagation as the default behaviour. However, exceptions can be propagated explicitly as well.

After the found handler has finished its execution, the control is passed to the statement following the protected region where the exception was handled: the termination model is adopted with return semantics (as in CLU). Modula-3 provides programmers with the try... finally construct for defining clean-up actions. No support is provided for concurrent exception handling in this language.

4.6. Exception handling in C++

Exceptions in C++ (Koening and Stroustrup, 1990) are data objects. External exceptions may be optionally included as part of the method signature. However, there is no distinction between internal and external exceptions. Like Ada 95 and Modula-3, C++ also introduces an exception mechanism that is sensitive to contexts. The handling context is the try block: handlers are declared at the end of such block using the keyword catch. An exception is handled by invoking an appropriate handler selected from a list of handlers found immediately after the try block:

```
try { ... -- protected region -- ...
    }
catch (E1) { ... -- handler -- ... }
catch (E2) { ... -- handler -- ... }
catch (...) { ... -- all-encompassing
              handler -- ...}
```

A handler catches an exception object of the specified type. Each handler declares a class as a parameter, so it can catch an exception object of any subclass of this class as well. C++ also allows the definition of all-encompassing default handlers by means of construct catch (...). As in Ada 95 and Modula-3, this handler catches only exceptions for which the programmer does not provide specific handlers.

Unlike CLU, the C++ exception mechanism implements semi-dynamic binding and supports both automatic and explicit exception propagation. However, automatic propagation is the default behaviour. With respect to continuation of the control flow, only the termination model with return semantics is supported. If the execution of clause catch terminates without raising any exception, it continues normally from the first statement after the try block to which the executed handler is attached. The model does not provide any specific support for clean-up actions. As exceptions are typed entities, static checks can be performed by the compiler. However, because the use of exception interface is not compulsory, dynamic checks are performed by the run-time system. C++ provides no support for concurrent exception handling.

4.7. Exception handling in Java

Java (Arnold and Gosling, 1998; Papurt, 1998) is considered to be a language from the C++ family which relies on many similar design decisions. For instance, Java supports representation of exceptions as data objects, semi-dynamic binding, and the termination model with return semantics. Java allows software developers to define protected regions using the try block. However, its exception handling is much safer and clearer than that of its ancestor. As for the main aspects of exception handling, Java has more powerful features than C++, because it allows better static checking and provides specific support for programming clean-up actions. However, there is not any distinction between internal and external exceptions.

Java adopts a hybrid solution for exception interface. All exceptions must be throwable, that is they must be inherited (directly or indirectly) from class Throwable.

This class has two predefined subclasses Error and Exception. Subclass Exception has a predefined subclass RunTimeException. All subclasses of Throwable can be categorised into two groups: classes, that inherit from either Error or RunTimeException, these exceptions are *unchecked*, and other classes inheriting from either the Exception class or directly from class Throwable, which are called *checked*. The first group includes exceptions after which ordinary programs are not expected to recover (for example, loading and linkage errors, virtual machine errors), and exceptions that occur within the Java run-time system (for example, arithmetic, pointer, indexing exceptions). The compiler does not require that programmers either catch the unchecked exceptions or specify them in the method signature. Any checked exception that can be thrown directly or indirectly within the scope of the method must be either caught or specified.

As opposed to C++, Java provides programmers with construct try... finally which defines the clean-up actions. The finally block is always executed at the end of the try block, whether an exception is raised or not, except for the situation when the try block raises an exception that is not caught by its handlers, in which case the exception is propagated.

Although Java describes clearly the semantics of exception handling in concurrent programs, it does not offer a more general support for concurrent exception handling. The Java exception mechanism is integrated with the Java thread/synchronisation model: all locks of the callee object are released when a synchronised method signals an exception to the caller. An asynchronous exception (signal) can be raised in a thread of concurrent program by invoking the stop method of class Thread.

4.8. Exception handling in Object Pascal/Delphi

Object Pascal (Teixeira and Pacheco, 1999) is the underlying programming language of Delphi, a tool for rapid application development. Exceptions in Object Pascal are data objects. External exceptions cannot be declared in method signatures and there is not any distinction between internal and external exceptions. Like in Modula-3 and Java, handlers can be attached to a statement or a block. A protected region starts with keyword try and ends with keyword end. As Modula-3 and Ada 95, Object Pascal allows all-encompassing default handlers to be defined. After an exception is handled, execution continues from the end of the block in which the exception has been handled. Therefore, Object Pascal implements the termination model with return semantics.

Unlike CLU, exceptions in Object Pascal are propagated automatically and handlers may be associated semi-dynamically. If a block does not handle a particular exception, the execution continues by raising an exception in the block that contains this block or in the code that called it. This process repeats with an increasingly wider scope until either execution reaches the outermost scope of the application or a block at some level handles the exception. However, a handler may explicitly reraise the same exception by executing the raise statement without the argument. Like in Modula-3 and Java, the clean-up actions may be specified by using construct try... finally. The application always executes any statements in this block, even if an exception occurs in the protected block. Because this model adopts the semi-dynamic binding and does not support exception interface, there is very little that can be checked statically. This model does not support concurrent programming or, for that matter, concurrent exception handling.

4.9. Exception handling in Guide

Exception mechanism in Guide (Balter et al., 1994; Lacourte, 1991) is similar to the CLU one. Guide exceptions are symbols by their nature. However, complementary information can be passed to handlers while raising exceptions. Each handler can learn the names of the class and of the method that signalled the exception. In Guide, all exceptions that can be raised must be declared in the class interface. However, there is not any distinction between internal and external exceptions. In this model handlers can be attached to method invocations (statements), methods and classes.

If a statement raises an exception, the method containing it always signals the exception to its caller. Local handlers are not possible. This exception mechanism adopts the termination model with return semantics. After the handler has been normally completed the execution continues from the next statement after the invocation of the method that raised the exception. The binding is semi-dynamic in this model, and all exceptions propagated from a method should be either handled or resignalled, or another exception should be explicitly propagated further. Exceptions are not propagated automatically in Guide. When an exception is not handled by the caller, a special exception called Uncaught_Exception is propagated. The retry policy can be implemented using the retry statement (this can be used only inside a handler).

The Guide approach allows programmers to guarantee the consistency of objects. Using the restore block one can define a block with the clean-up actions. The block is executed just after an exception has been raised but before the execution of the handler. Recursively, if the handler propagates an exception, then the restore block of the caller object is executed before the search for a new handler starts. Because Guide uses static binding and the use of exception interface is obligatory most checks are performed statically. Although Guide

provides constructs for concurrent programming, its exception mechanism does not support concurrent exception handling.

4.10. Exception handling in extended Ada

Cui's and Gannon's approach, called data-oriented exception handling (Cui and Gannon, 1992), allows associating handlers with the objects declarations. This concept has been implemented as an Ada pre-processor and empirical studies (Cui and Gannon, 1992) have shown that the programs that are produced this approach are smaller and better structured then the programs produced using the Ada exception handling. Handlers in Ada can be attached to blocks, which can create many problems as it is an error-prone solution and very often these blocks are introduced to associate different handlers with different objects. This approach inserts exception handling code in the middle of the main algorithm and prevents a clear separation of the normal and abnormal code. The data-oriented exception handling removes exception handling code from the algorithmic code improving both code writeability and readability.

In this model exceptions are declared in the type declaration specified as a generic package. Handlers are statically attached to object variables in these declarations. Each object declaration contains a set of (exception, handler) pairs. Three auxiliary language features are defined to support this model: #exception, #when, and #raise. Exceptions are declared by attaching clause #exception to the type exported from the package specification. Handlers are associated with the object declaration by attaching clause #when to the declaration that specifies the handler procedures for the exceptions defined on the object type. Exceptions are signalled by the #raise statement containing parameters which indicate the object with failure. Default handlers for exceptions can be specified in a type declaration, so that all variables of this type will use them. Like in CLU, only the termination model is supported. Although this approach has bottom up compatibility with Ada, it does not support exception propagation.

4.11. Exception handling in BETA

BETA (Madsen et al., 1993) is an object-oriented language that generalises some of the concepts introduced by Simula67. BETA has no special constructs for exception handling, but uses a general language construct instead. The abstraction mechanism called *pattern* replaces classes, procedures, functions, types and exceptions. Instances of patterns are called objects and can be used as variables, data structures, procedures, functions, etc. Inheritance is implemented using the supper-pattern mechanism, which allows explicit control of overriding using the inner construct. Exceptions are represented as *virtual patterns*, a variant of a pattern, which describes the construction of classes or of individual objects.

Signalling an exception amounts to direct call of a handler by its name. A handler can be attached to class, object, method or statement. The mechanism is based on the static binding approach, i.e., there is no run-time search for handlers. The default continuation of control flow is strict termination of the program, although BETA supports resumption as well.

Propagation of an exception to the caller is not supported. For this reason, the concept of dynamic search for a handler does not exist in BETA. The inner construct and virtual patterns together provide a way for the invoker of an operation to affect the handling of an exception inside the object. The code extension (called binding) given by the invoker at the time of an invocation is executed substituting the operation code at the place where inner is declared. The inner mechanism also allows a subclass to extend or augment the exception handling of the parent class. Using the inner mechanism for this purpose requires a good understanding of the pattern code (Miller and Tripathi, 1997).

4.12. Exception handling in Arche

Arche (Issarny, 1993) is a concurrent object-oriented language which makes a clear distinction between type (a description of an interface) and class (an implementation of a type). Exceptions are data objects in Arche. Exception interface is supported, but there is not any distinction between internal and external exceptions. The signals clause can be used in any method signature to define the external exceptions that the method may signal. There are strict rules of subtyping for procedures: any exceptions of the set of the clause signals in a subtype procedure should be a subtype of an exception belonging to the exception set of the parent. In an extreme case you can have an empty list in the child even though the parent has several exceptions in the signals clause.

Handlers can be attached to blocks and they are declared by means of a construct similar to Modula-3. Like CLU, Arche adopts the termination model with explicit propagation of exceptions. The signal command is used by handler to explicitly propagate the failure. Thus, the handler is searched using explicit propagation of exceptions. If the search fails, predefined exception failure is signalled. Handler binding is semi-dynamic in Arche. However, the absence of handlers can be detected at compile time due to explicit propagation.

Among the object-oriented languages surveyed in this study, Arche provides the best scheme for concurrent exception handling. Cooperating threads can be dealt with using the concept of object groups. An object group is declared as a sequence of objects using the type

constructor seq of. Methods of objects from a group are executed concurrently and synchronously within the group scope, this is called a *multi-operation*. In Arche, the notion of multi-operation is therefore the basic structuring mechanism for providing fault tolerance. Multi-operations can be designed as atomic actions (Section 2.2).

An object group can issue a coordinated call (a natural extension of the method-call mechanism allowing multiple callers and multiple callees), in which case several components join together to call a multi-operation (the run-time support synchronises them at this point). This operation is executed concurrently by each of the multiple callees. When the call is completed, the results – if any – are made available to all calling components before their parallel activities are resumed. During execution of a multi-operation, several callee components may concurrently signal different exceptions. To deal with them Arche provides a resolution function, which is declared within the class. A resolution function takes a set of exceptions as an input and returns a resolved exception called *concerted exception* which will be signalled to all callers. The resolution function is implicitly invoked each time when an execution of a multi-operation results in signalling of at least one exception.

4.13. Exception handling in other languages

Trellis/Owl (Schaffert et al., 1986) is an important landmark in the history of developing object-oriented languages. When an operation is invoked in Trellis/Owl but cannot be completed, one of the exceptions associated with the interface of this operation is signalled. The message not found error (a run-time error which is similar to errors found in untyped languages like Smalltalk) cannot occur due to the strict compile-time type checking. In this language, exceptions are created using the signal keyword; exception handlers are defined using the exception clause, similar to CLU exception handling.

Act1 (Lieberman, 1987) is an object-oriented language designed on the basis of the actor model. Any actor can delegate a message to which it is not able to respond, to another actor called proxy. When the proxy takes charge of the task which is delegated to it, the delegating actor becomes ready to process another message. The proxy knows the delegating actor, and it can use it when it need additional information. Delegation, like class inheritance, is a means of sharing behaviour (i.e., operations). However, it is more flexible than inheritance: an actor can dynamically choose new proxies, whereas the inheritance graph is statically defined at compile time. A specific actor called Object is the universal proxy and the root of the ''delegation tree''. Error handling in Act1 is distributed: a message can be distributedly transferred to one of the actors which will handle the exception. These error handling actors can, for example, activate an interactive debugger. Users can dynamically define new actors which will be handling exceptions in a way which suits best their requirements.

5. Evaluation and discussion

Table 1 provides a summary of the main features of the exception mechanisms presented in Section 4, showing the different approaches for each design aspect of our taxonomy (Section 3). In order to compare the exception mechanisms, we assign weights -1, 0 or $+1$ to each design approach, which reflect the contribution (negative, neutral or positive) of that approach in facilitating the construction of fault-tolerant applications. Adding up the weights of a mechanism produces a final score that is an indication of the suitability of that mechanism for the development of fault-tolerant software.

The reader should note that some design aspects have mutually exclusive design approaches, while for others the approaches have an additive character (these are marked with '(+)' in Table 1). Thus, the highest score attainable by a mechanism is 15, obtained by adding the maximum possible scores for each aspect A1–A10: $1+1+1+5+1+1+1+1+2+1$. Section 5.1 evaluates the approaches for each exception handling aspect of Table 1. Section 5.2 summarises our findings.

5.1. Evaluation of design approaches

A1. *Exception representation*. We observe that half of the mechanisms have represented exceptions as symbols and half as objects. Of these, four used data object representation and two used full object representation. Adopting symbols to represent exceptions in object-oriented languages produces a undesirable non-uniform programming environment. This justifies the negative weight assigned to this approach and positive weights assigned to the two object approaches.

A second advantage of representing exceptions as objects is that it allows the inclusion of context-related information in the state of the exception object, facilitating the handling of the exceptional situation. When, otherwise, exceptions are mere symbols, context-related information should be passed via global variables, which decreases the system's modularity.

A2. *External exceptions in signatures*. According to our study, (i) five exception mechanisms do not support explicit declaration of external exceptions in signatures; (ii) four of them support this feature but do not make it compulsory; (iii) two explicitly force its use; and (iv) only Java's mechanism adopts a hybrid solution, in which runtime exceptions are not required to be part of

Table 1
Summary of design aspects of exception mechanisms

Taxonomy Aspects	Design Decisions \ Exception Mechanisms	Ada 95	Lore	Smalltalk	Eiffel	Modula3	C++	Java	Delphi	Guide	Ext. Ada	BETA	Arche
A1. Exception Representation	Symbols	-1		-1	-1	-1				-1	-1		
	Data Objects						+1	+1	+1				+1
	Full Objects		+1									+1	
A2. External Exceptions in Signatures	Unsupported	-1		-1	-1				-1			-1	
	Optional		0				0						0
	Compulsory					+1				+1	+1		
	Hybrid							+1					
A3. Separation between Internal and External Exceptions	Unsupported	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1	-1
	Supported												
A4. Attachment of Handlers (+)	Statement	+1	+1			+1	+1	+1	+1	+1		+1	+1
	Block	-1				-1	-1	-1	-1				-1
	Method				+1					+1		+1	
	Object										+1	+1	
	Class	+1	+1	+1	+1					+1	+1	+1	
	Exception		+1									+1	
A5. Handler Binding	Static			-1							-1	-1	
	Dynamic												
	Semi-Dynamic	+1	+1		+1	+1	+1	+1	+1	+1			+1
A6. Propagation of Exceptions (+)	Unsupported			-1							-1	-1	
	Automatic	-1	-1		-1	-1	-1	-1	-1				
	Explicit	+1	+1			+1	+1	+1	+1	+1			+1
A7. Continuation of Control Flow (+)	Resumption		-1	-1								-1	
	Termination	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1	+1
A8. Clean-up Actions	Unsupported										-1		
	Use of Explicit Propagation	0			0		0						0
	Specific Construct		+1	+1		+1		+1	+1	+1		+1	
	Automatic Clean-up												
A9. Reliability Checks (+)	Dynamic Checks	+1	+1	+1	+1	+1	+1	+1	+1	+1			+1
	Static Checks				+1	+1	+1	+1	+1	+1	+1	+1	+1
A10. Concurrent Exception Handling	Unsupported		-1	-1	-1	-1	-1		-1	-1	-1	-1	
	Limited	0						0					
	Complete												+1
	Final Score	1	5	-3	1	3	3	6	3	7	-1	3	6

a method's signature, while all checked exceptions must be specified in the signature; so, by requiring the catching of checked exceptions, Java forces programs to handle conditions that the developer can anticipate. We conclude that, regrettably, a large majority of designers of exception mechanisms have opted for a more flexible approach, not requiring the specification of exceptions in the methods' signatures. This specification explicitly shows the abnormal responses of a method, allowing a client to guard against exceptional situations by providing appropriate handlers (Ghezzi and Jazayeri, 1997). This approach is well integrated with the idealised fault-tolerant component model, described in Section 2.1. In order to reflect these qualities, we have assigned weight -1 to mechanisms which do not support external exceptions in signatures, whereas weight $+1$ was assigned to those which support compulsory and hybrid solutions; the optional approach was given weight 0.

A3. *Separation between internal and external exceptions*. We believe the separation between internal and external exceptions is important. However, none of the studied mechanisms has implemented this separation; accordingly, we have assigned them weight -1 in this approach in Table 1. Moreover, it is also important to assure that an internal exception raised by a specific component is not propagated to higher-level components; that is, only exceptions categorised as external can be propagated outwards (see Section 2.1). Ideally, the following restrictions should be imposed (Romanovsky, 2000a): given a component X, (i) all its internal exceptions have to have handlers; and (ii) all external exceptions have to be signalled to the containing exception context, in order to inform that X was not able to produce the required result. These external exceptions have to be internal to each component Y that uses the services of component X. These rules of safe

exception handling can be easily checked during compiling time.

A4. *Attachment of handlers*. Among the twelve studied mechanisms, (i) nine have included statement handlers, (ii) six implemented block handlers, (iii) three included method handlers, (iv) only two (extended Ada and BETA) have included handlers at the level of objects, (v) seven implemented class handlers, and (vi) only two (Lore and BETA) have included exception handlers. For the purpose of enhancing the structuring of dependable systems, it is desirable to allow the attachment of handlers to as many levels of protected regions (exceptions, classes, objects, methods and statements) as possible.

We believe that the use of block handlers violates explicit separation of concerns, since exceptional code is intermingled with normal code, albeit moved to the end of the block. Another disadvantage of defining blocks of commands is that nested blocks are usually declared for the sole purpose of attaching an exception handler (Lang and Stewart, 1998; Papurt, 1998). As a result, it leads to the development of dependable software which is difficult to read, maintain and test. In addition, block handlers are not absolutely necessary; instead, statement handlers should be used since they treat exceptional conditions one at a time and do not violate separation of concerns. Correspondingly, block handlers are assigned weight -1 while the other approaches received $+1$. The exception handling model of Guide is an example of design which favours statement over block handlers, achieving explicit separation between normal and exceptional activities. Note, also that BETA's choices for this aspect achieved the highest possible score.

A5. *Handler binding*. Three mechanisms have implemented the static approach to handler binding, none has implemented the dynamic approach, and nine mechanisms have implemented the semi-dynamic approach. Static binding leads to better readability since it is easier to verify statically which handler would be activated in a given exception occurrence. With dynamic and semi-dynamic binding, this verification is more difficult since exceptions can be propagated dynamically and the binding depends on the control flow at run-time. However, the static approach does not allow exception propagation and there is no run-time search for handlers (Section 3). This is a limitation since the callers of an operation generally have better solutions for handling exceptional situations than statically bound handlers, which are unaware of the computation history (Dony et al., 1991). The semi-dynamic and dynamic approaches, however, take the invocation history into account while looking for a handler. In view of the discussion in Section 3, we believe the semi-dynamic approach is the most suitable choice. These reasons justify our choice of weights for this aspect in Table 1.

A6. *Propagation of exceptions*. Three mechanisms have not provided any support for exception propagation, seven have supported automatic propagation and eight mechanisms have implemented explicit propagation. The exception mechanisms of Smalltalk, extended Ada and BETA adopt static binding; therefore, they do not support any kind of propagation. Although most mechanisms allow both explicit and automatic propagation of exceptions, automatic propagation is usually adopted as the default behaviour. However, automatic propagation leads to an unsafe exception handling model: it may allow an exception occurrence to be inadvertently bound to an inappropriate handler. In addition, automatic propagation of unhandled exceptions through different levels of abstraction may compromise encapsulation and decrease modularity (Yemini and Berry, 1985), because the exception object can reveal information about the original signaller to other components than its immediate caller. Explicit propagation solves this problem since the handling of an exception occurrence is limited to the immediate caller. Our strong preference for explicit propagation is made clear by our choice of weights in Table 1.

A7. *Continuation of the control flow*. All mechanisms have adopted the termination model. BETA, Smalltalk and Lore provide both the termination and resumption models. Mechanisms that support resumption are very flexible but also very difficult to use, as well as error-prone. As far as fault tolerance is concerned the termination model is considered to be most adequate due to its clearer semantics (Liskov and Snyder, 1979). Our choice of weights in Table 1 expresses this preference. The interested reader will find in Cristian (1989) a formal treatment of the termination model within the framework of software fault tolerance.

A8. *Clean-up actions*. Only one mechanism (extended Ada) has not provided any support for clean-up actions. Among those providing such support, four have allowed the specification of clean-up actions by means of explicit propagation only, seven mechanisms have provided specific constructs, and none has provided automatic facilities. Ideally the exception mechanism should be responsible for performing clean-up actions automatically, thus making the programmer's task easier and less error prone. However, the effective implementation of automatic clean-up is still a challenging problem and further investigation of alternative techniques is required (Lang and Stewart, 1998). The weights assigned to this aspect in Table 1 favour those approaches which explicitly support clean-up actions, either automatically or not.

A9. *Reliability checks*. Seven mechanisms perform static checks combined with some degree of dynamic checking. Three mechanisms have implemented only dynamic checking and two implemented only static

checking. Ideally, a mechanism should support both approaches, with most checking done statically and the rest only at runtime, when the information needed is available. Thus, each approach gets a +1 weight in Table 1.

A10. *Concurrent exception handling*. Nine mechanisms do not support concurrent exception handling in any way, two provide some very limited form of support and only one, Arche, completely supports it. It is, thus, the only mechanism which gets a +1 weight in Table 1 for this aspect. Arche's exception 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 co-ordinated recovery of multiple interacting objects of different types.

5.2. Summary of discussions

As much as possible, exception mechanisms should be simple and reliable schemes for developing robust programs. In spite of this aim, the previous discussions revealed that several decisions taken in the design of the studied exception mechanisms resulted in solutions which are too flexible and complex, leading, paradoxically, to the construction of dependable object-oriented software which is error prone.

Ranking the studied mechanisms according to their final score, out of the maximum of 15, we have: Guide (7), Java and Arche (6), Lore (5), Modula-3, C++, Delphi and BETA (3), Ada 95 and Eiffel (1), Extended Ada (−1), and Smalltalk (−3). A word of caution should be said about this ranking: from the perspective of fault tolerance, Guide's, Java's and Arche's exception mechanisms possess the highest numbers of positive features; however, (i) they are still too far from what we believe to be the ideal solution represented by a score of 15; and (ii) when other practical considerations also have to be taken into account, such as performance and widespread use of the language, a trade-off may be necessary when choosing a particular mechanism.

Table 1 presents, in a summarised fashion, both the positive and negative features of each mechanism, allowing the developer to compare different mechanisms and evaluate potential difficulties and the impact of a given choice in the construction of dependable software. Therefore, Table 1 is meant primarily as a guide for decision-making rather than an absolute measurement of the suitability of a given mechanism.

6. General design criteria

The taxonomy developed in Section 3 identifies several design issues, which should be taken into account while designing an exception mechanism, and classified possible approaches to tackle them. The design decisions should be taken according to the demanding quality requirements (Fig. 4). Some of these requirements are stated for the dependable object-oriented applications which use the exception mechanism, while the others are imposed on the exception mechanism itself. Our analysis in Section 5 shows that the existing exception handling mechanisms do not satisfied these requirements.

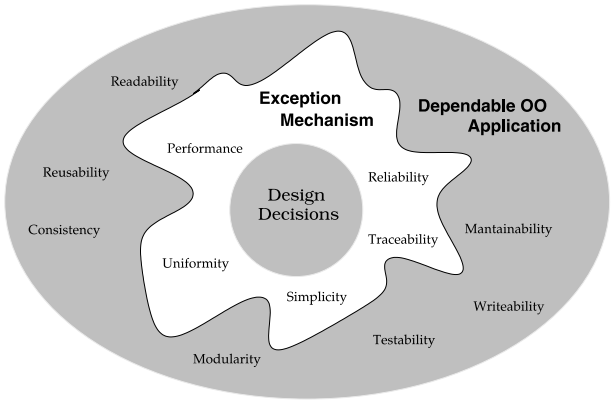

Fig. 4. Quality requirements of exception mechanisms.

This section outlines the main criteria which the design of an effective exception mechanism for developing dependable object-oriented systems should follow. Based on these criteria, we identify the choices for designing an ideal exception handling model suitable for developing this kind of systems. The criteria and the proposed exception handling model have been developed using our extensive experience in designing both dependable object-oriented systems (Ferreira, 1999; Romanovsky et al., 1997; Rubira, 1994; Xu et al., 1995a,, 1998, 1999) and the exception mechanisms suitable for this kind of systems (Garcia et al., 1999, 2000; Garcia, 2000; Romanovsky, 1997; Romanovsky et al., 1996; Xu et al., 1995a, 2000; Zorzo et al., 1999).

6.1. Quality requirements

Q1. *Readability*. One of the main reasons for using exception mechanisms is improving the program readability (Burns and Wellings, 1996; Mitchell et al., 1998). The importance of readability is even higher for modern dependable object-oriented software in which a number of possible exceptions and the amount of exceptional activity dealing with such exceptions are considerable. The exception mechanism should promote explicit and clear separation between the exceptional and normal execution code, following the overall structure of the idealised-fault-tolerant-component model (Section 2.1). Application code developed using such mechanism is easier to read and to understand because it highlights the main purposes and the boundaries of both the protected

region and the abnormal code located in the exception handler section.

Q2. *Modularity*. An exception mechanism should yield dependable object-oriented applications in which the effect of an abnormal condition occurring at run-time in a component is confined by this component, or, more generally, is kept under control (e.g., it is propagated to few neighbouring component only) (Meyer, 1988). In this way, the exception mechanism ensures that each component of a dependable object-oriented application practices information hiding (Yemini and Berry, 1985).

Q3. *Maintainability*. It is widely accepted that up to 70% of software cost is devoted to maintenance (Meyer, 1988). An effective exception mechanism should not neglect this aspect and should make the program maintenance easier. If dependable object-oriented software cannot be easily changed, additional errors can be introduced during the maintenance phase. This is why, the design of an exception mechanism for dependable object-oriented systems should specifically focus on the mechanism simplicity and program readability.

Q4. *Reusability*. Designing for reusability allows the third-party components to be used in building new systems (usually with some customisation or adjustment). It should be possible to reuse the exceptional code of a software component as well as its normal code. Ideally, exceptions and handlers should be defined independently of the normal activity of the component, so that exceptions, handlers and normal code can be reused separately. Note that separation of different handlers from each other also improves reusability.

Q5. *Testability*. Software testability refers to the ease with faults can be revealed through (typically, execution-based) testing. In general, system testability relates to several structural issues (Bass et al., 1998): separation of concerns, level of documentation, and the degree to which the system uses information hiding. Object-oriented software testing is still an evolving area. Furthermore, adding exceptions and exceptional behaviour significantly complicates the testing activity. In the ideal world it should not be difficult to verify analytically that every possible error has a known handler, and to test every exceptional scenario in a systematic manner. It is clear that dependable object-oriented software should be extensively tested to decrease the possibility of manifestation of residual faults at run-time. The design decisions about an exception mechanism should be taken without undermining the testability of dependable object-oriented software.

Q6. *Writeability*. Dependable object-oriented systems should incorporate error recovery activities at various levels of the system. In this way, the complexity of such systems can be controlled in a flexible and systematic way, and redundancy can be added at the levels where its needed. However, care should be taken while improving the expressive power of exception handling. Unnecessary expressive power may complicate both using the exception mechanism and understanding the code (Section 5).

Q7. *Consistency*. Components of a software system should be kept in a consistent state regardless of whether their execution completes normally or is interrupted by an exception. Consistency of components of dependable object-oriented systems should always be maintained, because such systems should be able to continue execution even in the presence of errors to prevent catastrophic failures. A very useful approach for providing consistency is to apply the abort semantics to guarantee that when an exception happens a component is left in a state that it had when the execution started.

Q8. *Reliability*. The features of an exception mechanism should help in developing reliable programs. The designers of a dependable object-oriented software have to deal with faults of different types, and the exception mechanism itself should neither introduce nor cause additional design faults. Therefore, the exception mechanism must be tested (and, generally speaking, verified) exhaustively. If the exception mechanism itself fails then a predefined failure exception should be reported to the higher level passing the responsibility for recovery there. Moreover, additional faults should not be introduced by the use of the exception mechanism. The exception handling system should prevent common programmer errors from happening by performing reliability checks.

Q9. *Simplicity*. As with all language features, the exception mechanism must be easy to understand and to use. Meyer (Meyer, 1988) emphasises that a good exception mechanism should be simple and modest. Therefore, the exception mechanism should not contain unnecessary complexity. It should rely on a consistent semantics that uses a minimum number of underlying concepts. In other words, the number of the new concepts introduced by an exception mechanism should be as small as possible and consistent with the needs of the dependable object-oriented applications. It should have few special cases and should be composed from features with a simple and clear semantics. The programmers of dependable systems should have confident understanding of the features which the exception mechanism provides, so that they can concentrate on the complexity of their systems rather than on the intricacies of the exception mechanism.

Q10. *Uniformity*. The exception mechanism should follow uniform syntactic conventions and should not provide several notations for the same concept. In addition, the solutions followed while designing an ex-

ception mechanism for object-oriented systems should correspond well to the object-oriented paradigm. Object-oriented solutions should not be intermingled with conventional solutions. Violation of these rules undermine reusability, modularity, testability of the dependable object-oriented software employing the exception mechanism.

Q11. *Traceability*. Dependable object-oriented software often needs auxiliary information which can be used for accessing damage and for providing following error recovery. The information should be passed by the exception mechanism together with the notification of the exception; it may include name, description, location and severity of the exception, as well as, the propagation chain and other data (Section 2.1).

Q12. *Performance.* Performance is always an important concern in developing modern systems. There are two major issues in the exception mechanism design with respect to performance: (i) minimising the time for searching a suitable handler after an exception has been raised – ideally, the complexity of the search algorithm should be O(1); (ii) minimising the run-time overheads caused by the exception mechanism under normal operation conditions – ideally, the mechanism should be designed in such a way that run-time overheads are incurred only when an exception has been raised. However, in the case of dependable systems, where providing fast error recovery is the prime concern, an application may be ready to live with small overhead during the normal error-free operation (Burns and Wellings, 1996). Providing higher performance often contradicts achieving all other qualities. Some performance penalty should be acceptable when a higher quality exception mechanism is used. This is why all qualities should be prioritised and taken into account while designing an exception mechanism for dependable object-oriented software. Performance is the critical issue for developing exception mechanisms for hard real-time systems, which is not the aim of this study. We refer to (Lang and Stewart, 1998) for deeper discussions regarding exception handling in the context of real-time systems.

6.2. An ideal exception handling model

After analysing the quality requirements in the previous section, we proceed by discussing each individual design decision and present an ideal exception handling model suitable for developing dependable object-oriented software. We show how the quality requirements affect each choice to be made while designing exception handling mechanisms (Table 2 gives a summary of our findings). Tradeoffs are also discussed in this Section since the quality requirements are often conflicting.

D1. *Exceptions represented as objects*. Exceptions should be represented as (full or data) objects. This design decision has a number of advantages. It leads to a better traceability (Q11) and modularity (Q2) since extra information can be easily passed with exceptions when exception occurrences are objects (Section 5). Moreover, this representation is integrated smoothly (uniformly – Q10) with the object paradigm and has a number of advantages when compared to the classical approach; some of which are (Dony, 1988; Dony et al., 1991): (i) exceptions are organised into an inheritance hierarchy which makes the system easier to reuse (Q4), read (Q1), test (Q5), maintain (Q3) and extend; (ii) general handlers can be designed (Q6), since each handler does not only handle one kind of exceptions but all exceptions that are its subclasses, as a result of this, less handler bindings are needed, the program is shorter, which improves readability (Q2) and makes the mechanism simpler and easier to use (Q9); (iii) handlers can be attached to the exception classes, if they are independent of any execution context (Q6); (iv) handlers attached to classes can be inherited by subclasses (Q6); (v) the exception mechanism becomes more reliable (Q8) since representing exceptions as mere symbols may be error prone.

D2. *Compulsory definition of external exceptions in signatures*. According to the model of idealised fault-tolerant component (Section 2), each system component should be able to return well-defined results. The normal and exceptional results of a component of a dependable system should be rigorously specified. One should not have to examine the object implementation to understand which exceptional results (or exceptions) a method can return. In fact, the presence of external exceptions in the method signature leads to better readability (Q1) (Section 1). This feature also promotes the construction of modular software systems (Q2) (Dony et al., 1991), which in turn improves maintainability (Q3), reusability (Q4) and testability (Q5). Finally, it makes the conformance rules, which can be checked by the compiler, more stringent and the exception mechanism itself more reliable (Q8). Therefore, the definition of external exceptions in the method signature should be compulsory. However, a hybrid approach also could be adopted since it is sometimes difficult for designers to anticipate all exceptions, because exceptions of some types are unpredictable by their nature.

D3. *Separation between internal and external exceptions*. As we explained before (Section 5), we believe that is vitally important to separate internal and external exceptions and not to allow an internal exception to be propagated from the component. From our point of view these exceptions serve completely different purposes. The external exceptions are part of the component specification, they are intended for informing the environment about the fact that the component is not able to produce the correct result. Usually some additional information can be associated with such excep-

Table 2
Design decisions × quality requirements

Taxonomy Aspects		Design Decisions \ Quality Requirements	Application							Exception Mechanism				
			Q1.Readability	Q2.Modularity	Q3.Maintainability	Q4.Reusability	Q5.Testability	Q6.Writeability	Q7.Consistency	Q8.Reliability	Q9.Simplicity	Q10.Uniformity	Q11.Traceability	Q12.Performance
A1.	Exception Representation	D1. Full Objects/Data Objects	+	+	+	+	+	+		+	+	+	+	-
		Symbols	-	-	-	-	-	-		-	-	-	-	+
A2.	External Exceptions in Signatures	Unsupported/Optional	-	-	-	-	-	+		-	+	-		+
		D2. Compulsory	+	+	+	+	+	-		+	-	+		-
		D2. Hybrid	+	+	+	+	+	-		+	-	+		-
A3.	Separation between Internal and External Exceptions	Unsupported	-	-	-	-	-	+		-	+			+
		D3. Supported	+	+	+	+	+	-		+	-			-
A4.	Attachment of Handlers	D4. Statement	+	+	+	+	+	+			+	+		
		Block	-	-	-	-	-	+			-			
		D4. Method/Object/Class/Exception	+	+	+	+	+	-			+	+		
A5.	Handler Binding	Static	+	+		-	+			+	+			+
		Dynamic	-	-		+				-	-			-
		D5. Semi-Dynamic	-	-		+					-			-
A6.	Propagation of Exceptions	Unsupported	-	+	+	-	+	-			-			+
		Automatic	-	-	-	-	-	+		-	+			+
		D6. Explicit	+	+	+	+	+	-		+	-			-
A7.	Continuation of Control Flow	Resumption								-	-			-
		D7. Termination								+	+			+
A8.	Clean-up Actions	Unsupported	-						-	-				
		Use of Explicit Propagation	-					-	-	-	-			+
		D8. Specific Construct	+					+	+		+			+
		Automatic Clean-up	+		+	+	+	+	+	+	+			-
A9.	Reliability Checks	Unsupported								-				
		D9. Dynamic Checks								+				-
		D9. Static Checks								+				+
A10.	Concurrent Exception Handling	Unsupported	-		-	-	-	-	-	-				
		Limited	-	-	-	-	-	-	-	-	-			-
		D10. Complete	+	+	+	+	+	+	+	+	+			+

tions to inform the environment (e.g., the caller) about the abnormal state in which the component has been left and about possible causes of the exception. Internal exceptions are implementation-dependant; they are introduced by the component developer and hidden inside component. This design decision improves readability (Q1), modularity (Q2), maintainability (Q3), reusability (Q4), testability (Q5), and reliability (Q8).

D4. *Multi-level attachment of handlers*. Allowing the multi-level attachment of handlers (Section 5) improves writeability (Q6) as well as structuring of dependable object-oriented software. It is important for programmers to be able to attach handlers at different levels of system structure. When an exception is related to an operation, a handler for this exception may be locally associated with the operation. Alternatively, handlers can be associated with a class, in which case they are associated with all operations of that class. It is also possible to attach handlers to objects and exceptions themselves. Such flexible attachment has many advantages: (i) it provides a clear separation of the object abnormal behaviour from the normal one following the concept of an idealised fault-tolerant component (Q1); (ii) protected regions can be factored out at the respective levels of classes, objects and operations (Q6); (iii) software layering facilitates the design of fault-tolerant systems; (iv) a closer integration between the language as a whole and its exception mechanism could be obtained through the uniform use of the object paradigm (Q9). However, in our opinion block handlers should not be supported. Separation of concerns is of a high importance for dependable object-oriented systems but the use of block handlers usually intermingles the exception handling code with the normal flow of control. This often produces less readable (Q1) programs which

is difficult to reuse (Q4). The error handling code is usually very detailed and complex, which makes the normal code difficult to read (Q1) and maintain (Q3). Explicit separation of concerns supports achieving a number of software qualities: readability (Q1), modularity (Q2), maintainability (Q3), reusability (Q4), testability (Q5) and writeability (Q6).

D5. *Semi-dynamic binding*. Although the static approach leads to better readability (Q1), there are still several reasons why more dynamic approaches should be supported for dependable systems (see Section 5). We believe the semi-dynamic binding is sufficient. This binding associates different handlers with an exception depending on the contexts in which this exception is raised during the program execution. In addition, the semi-dynamic binding can be used to achieve functionality similar to that of the dynamic method. A semi-dynamically bound handler can call different handlers depending on run-time conditions. This cannot be achieved with static binding because the run-time condition may not be valid in some contexts. Although this design solution has negative affect on readability (Q1) and simplicity of the exception mechanism (Q9), the adoption of explicit propagation (D6) can considerably minimise such negative impacts because it limits the handler binding to the local context related only to the signaller and to the immediate caller.

D6. *Explicit propagation of exceptions*. Explicit propagation along the chain of invokers should be the only way of propagating exceptions. According to the CLU designers, the caller of a method x should know nothing about the exceptions signalled by the methods which are called during the execution of x. Handling of an exception occurrence should be strictly limited to the immediate caller. Explicit propagation directly improves modularity (Q2) (Section 5), which in turn improves readability (Q1), maintainability (Q3), reusability (Q4), testability (Q5) and reliability (Q8). The exception mechanism should not use automatic propagation and should make the programmers to explicitly rename any exception they are going to propagate further. When automatic propagation is disallowed, the set of handlers that can deal with a particular exception can be statically determined, thus allowing additional compiler checks (Q8). However, this design choice can make writeability (Q6) and performance (Q12) worse.

D7. *Termination*. As discussed in Section 5, the termination model should be the only model used for continuation of the control flow after an exception has been handled in dependable systems. Mechanisms implementing only termination are easy to construct (and hence it is more reliable – Q8) since the signalling of an exception can be viewed as an abnormal return from the component (Lee and Anderson, 1990). From the viewpoint of fault tolerance, the resumption model introduces unnecessary expressive power (Liskov and Snyder, 1979), as well as, additional complexity into the exception mechanism (Q9) (Liskov and Snyder, 1979; Issarny, 1993; Cristian, 1995). Practical experience with exception mechanisms which use the resumption model has shown that it is error-prone (Ghezzi and Jazayeri, 1997). Furthermore, it can promote the unreliable programming (Q8) by removing the symptom of an error without removing the cause (Ghezzi and Jazayeri, 1997).

D8. *Explicit support for clean-up actions.* The use of an exception mechanism might lead to state inconsistencies (Q7) when exceptions are raised (Schwille, 1993). Components of a dependable object-oriented application cannot be left in inconsistent states, because the system should continue to operate even in a presence of errors to prevent catastrophic failures. In many practical situations the automatic features for clean-up actions are infeasible (Section 5) as they may cause high overheads at run-time (Q12). In fact, none of the exception mechanisms in realistic object-oriented languages have automatic clean-up. Therefore, the most suitable solution is to provide programmers with specific support for developing application-specific clean-up actions. One of the reasons for this is that using explicit propagation to perform clean-up actions is more error-prone (Q8), and the resulting code is more difficult to understand and to use (Q9). Furthermore, using the explicit support for clean-ups leads to improving both readability (Q1) and writeability (Q6) because it allows programmers not to replicate code responsible for clean-up. Note that it is impossible to avoid this problem while using explicit exception propagation since the clean-up actions are implemented within each handler attached to the protected region.

D9. *Static reliability checks*. The exception mechanism should be suitable for creating software which meets high dependability requirements. It should allow extensive static checking, followed by some level of dynamic checking. This design decision should produce exception mechanisms which help programmers of dependable systems to follow good practice in developing reliable software (Q8). The crucial decisions of adopting exception interface (D2) and explicit exception propagation as design principles make the static checking more powerful and exhaustive. For instance, the compiler can verify if an exception being raised at run-time will have a bound handler.

D10. *Complete support for concurrent exception handling*. We believe that using an adequate support for exception handling in concurrent programming is crucial because modern dependable object-oriented applications are concurrent and distributed by their nature. In practice, the approach classified as limited

(Section 2.2) can lead to developing software components which are difficult to read (Q1), maintain (Q3), reuse (Q4) and test (Q5). In addition, the responsibility related to handler invocation and exception resolution is left with the application programmers, this decision can decrease the overall reliability of object-oriented applications (Q8). From the viewpoint of the current research in fault tolerance, concurrent exception handling is a complex issue (Section 3) which can be solved properly and uniformly only together with applying the general concept of atomic actions as the main structuring techniques. In this case the amount of problems, which the developers of dependable object-oriented systems face, is minimised, so that they can concentrate on the application-specific issues.

7. Ongoing research and deployment of existing exception mechanisms

Section 6.2 has presented an ideal exception handling model which is relevant to the development of dependable object-oriented software. However, as we have concluded in Section 5, exception mechanisms in existing object-oriented languages have not fully addressed this model. As a consequence, new error-handling techniques and methods are necessary. The goal of this section is twofold: (i) to discuss recent advances on exception handling techniques which can be used to improve the robustness of software systems, and (ii) to provide guidance of how to cope with limitations of a particular exception handling mechanism (due to limitation of space, in this work we discuss only the use of exceptions to enhance robustness for Java programs).

7.1. Recent advances on exception handling techniques

Exception handling and computational reflection. Ideally, a new technique developed for a specific programming language should not introduce new language features; in practice, this is infeasible in existing languages (Romanovsky, 2000b). A current trend for extending object-oriented programming languages is to use computational reflection (Maes, 1987), which is a non-intrusive technique for the incorporation of additional mechanisms into the underlying language, by means of a meta-object protocol. This introduces a new dimension of modularity – the separation of the base-level computation from the meta-level computation.

The work of Hof et al. (1997) describes an exception mechanism based on meta-level programming and computational reflection. Its implementation was carried out in a specific system but the solution is generic and could be implemented in other systems that support a reflection mechanism. However, this mechanism does not support concurrent exception handling in cooperating participants and is not fully integrated with the object paradigm. Mitchell et al. (1998) also propose an exception handling model which ensures complete separation between error handling and normal code. However, their proposal applies the reflection technique in a different form. Instead of applying computational reflection to achieve separation between application and management activities related to exception handling, their work uses reflection to separate the application's normal code (meta-level) from the application's exceptional code (base-level). Their proposal, though, is not adequate for producing dependable software systems, since it is too flexible, allowing the use of the resumption model. The work of Garcia et al. (1999) proposes a new error-handling technique for developing dependable object-oriented software also based on a reflective approach. The meta-level implements the exception mechanism, and the base-level encompasses the application. This proposal was implemented in Java, without any changes to the language itself, by means of a meta-object protocol. The proposed object-oriented exception handling model is based on the idealized fault-tolerant component (Section 2.1) and establishes a clear separation between exceptional and normal code. In addition, this model supports: (i) exceptions represented as data objects; (ii) external exceptions in signatures; (iii) the attachment of handlers at different levels, such as methods, individual objects or groups of objects, classes, and exceptions; (iv) semi-dynamic binding; (v) explicit propagation; (vi) termination; and (vii) concurrent exception handling. Nevertheless, it does not support either statement handlers or else explicit separation between external and internal exceptions.

Exception handling and N-version programming. The work of Romanovsky (2000a) presents a scheme for introducing exception handling into object-oriented N-version programming, and shows the importance of using exceptions while applying diversely developed software. Internal and external exceptions are clearly separated in this proposal: each version has its own internal exceptions but the external exceptions of all versions have to be the same and identical to the interface exceptions of the diversely designed class. The exception handling model presents nice features: (i) external exceptions in signatures; (ii) the attachment of handlers at different levels, such as statements, methods and classes; (iii) explicit propagation; and (iv) termination. However, exceptions are not represented as data or full objects, and clean-up actions are not supported.

Exception handling, frameworks and design patterns. An object-oriented framework (Johnson, 1997) is a reusable software system that can be extended to produce customized applications. Framework designers specify variations within its design by means of extension points, which are those aspects of a domain that have to remain flexible; developers extend and customize the

framework design according to their specific application needs by filling in those extension points. An object-oriented framework for exception handling can be designed in such a way that its extension points represent the various design choices (see Table 1) for building different exception handling systems. For instance, propagation of exceptions could have two extension points associated with it: automatic or explicit. It is worth noting that computational reflection can be used to customize the exception handling framework for a particular application in a transparent and non-intrusive way for the framework users.

According to (Johnson, 1997) the use of design patterns is extremely useful as a guide during the framework development and also for providing a better understanding of its design to potential users (Buschmann et al., 1996). Thus, a system of patterns for exception handling could be developed to assist the building of an exception handling framework and document its design. The work of Garcia et al. (2000) proposes one such set of design patterns. These design patterns accommodate the ideal exception handling model (Section 6.2), and describe solutions that are independent of programming language or exception handling mechanism. These patterns follow the computational reflection notion in order to: (i) incorporate language extensions related to exception handling in a transparent way, i.e., without any changes to the language itself, and (ii) allow for a clear separation of concerns between the dependable system's functionality (base-level) and the exception handling features (meta-level). The base-level encompasses the application-dependent elements, such as exceptions, handlers, normal activities and cooperating threads. The meta-level consists of meta-objects which perform the management activities for exception handling, including invocation of handlers, deviation of the control flow, synchronization and exception resolution (Section 2.2), and so on. In general, meta-objects intercept and check method/operation results in order to perform these management activities. The *Error Detection* pattern (Renzel, 1997) proposes a design solution for detecting errors of an application at runtime. However, such a pattern only addresses error detection, without defining means for handling them.

Concurrent exception handling. As we have examined in this paper, very few exception handling mechanisms offer complete support for handling concurrent exceptions. Some works have been proposed to integrate concurrent exception handling with the atomic action concept. The work in Romanovsky (1997) describes a concurrent exception mechanism based on atomic action structures for the Ada 95 language. The coordinated atomic action (CA action) concept (Xu et al., 1995a) was introduced as a unified approach for structuring complex concurrent activities and supporting error handling between multiple interacting objects in a concurrent object-oriented system. CA actions provide a suitable framework for developing dependable object-oriented systems. In Romanovsky et al. (1996) the authors discuss the introduction of concurrent exception handling and CA action schemes into object-oriented systems and present a distributed exception–resolution algorithm.

Exception handling and OO software development. As stated previously, error handling activities play a crucial role in the development of dependable object-oriented software. Generally, traditional methods for developing object-oriented software deal with exceptions too late, only at the implementation phase. Better results can be achieved if exceptions and exception handling were incorporated in a systematic and disciplined way during all stages of software development, beginning with the requirement phase, going through analysis, architectural design, detailed design and implementation. The work of de Lemos and Romanovsky (1999) describes a systematic approach for defining exceptions and their respective handlers, thus eliminating the ad hoc character with which exception handling is usually dealt with. More recent work which deals with *obstacles* in a goal-driven approach for requirements engineering has provided systematic techniques for identifying failure behaviors in requirements specifications (van Lamsweerde and Letier, 2000).

Exception handling and CASE tools. It is desirable to have the extra, abnormal code dealing with exceptional conditions automatically generated as much as possible, with the help of CASE tools and standard templates (Lang and Stewart, 1998). Only a few researchers have dealt with this question in the existing literature. *Xept* (Vo et al., 1997) is a tool for providing source code with the ability to detect, mask, recover and propagate exceptions from library functions.

7.2. Deploying an existing exception mechanism

Although none of the existing object-oriented languages have implemented the exception handling model presented in Section 6.2, our study can be used by developers of dependable systems and exception mechanisms in order to understand the advantages and disadvantages of a given exception mechanism and deploy it in order to use exceptions effectively. In this section, we provide: (i) dependable software engineers with design guidelines to develop robust Java programs, and (ii) exception mechanism developers with design directives to adapt transparently the Java's exception mechanism by applying computational reflection and meta-level programming (Section 7.1).

Guidelines to design robust Java programs. The works of (Papurt, 1998; Robillard and Murphy, 2000) show a set of guidelines that help the proper use of Java's

exception handling mechanism, improving the robustness and maintainability of Java programs. One can apply these guidelines in order to adhere to the ideal exception model summarised in Table 2. Papurt (1998) demonstrates how Java programmers can avoid the use of block handlers. His solution consists of defining each protected region (i.e., a try block) of a Java program as a statement (following the design decision D4 of Table 2). Moreover, Robillard and Murphy (2000) propose an additional set of guidelines: (i) to map internal exceptions to external exceptions when propagating them (design decision D3 of Table 2), (ii) to use explicit exception propagation only (design decision D6 of Table 2), and (iii) to use external exception in signatures (design decision D2 of Table 2).

Guidelines to design a reflective exception mechanism for Java. Java does not separate internal and external exceptions, since it only supports a clause for raising exceptions, the throw clause. Meta-level programming and the approach implemented in CLU (Section 4) can be adopted to tackle this problem. CLU clearly distinguishes internal and external exceptions by supporting two different clauses, one to raise internal exceptions and another to signal external exceptions. In this way, a new operation for exception signalling should be created (for instance, a signal operation). This operation must be devoted only to signal external exceptions, and the throw clause of Java must be used only to raise internal exceptions. Meta-level programming is used for assuring that an internal exception is not propagated to the caller.

Java does not support handlers at the level of methods, objects, classes and exceptions. *Exceptional classes* and meta-level programming should be used in order to address this issue. The exceptional classes are application classes that implement the method, object and class handlers; i.e., the methods of exceptional classes are the handlers (at the level of methods, objects, classes) for the exceptions raised during the execution of normal classes' methods. In addition, since exceptions are represented as classes in Java, handlers at the level of exceptions can be defined as methods on exception classes. Meta-objects are associated with exceptional classes, and are responsible for invoking the suitable handler transparently.

Java adopts automatic propagation of exceptions in spite of the pitfalls of this approach. Meta-objects can be implemented to enforce the explicit exception propagation. When a method result is intercepted by a meta-object and this result is an exceptional one, meta-objects keep information in order to avoid automatic propagation of this exception.

Java does not provide explicit facilities for using atomic actions with concurrent exception handling. Computational reflection can be used to separate concerns and minimize dependencies between the application's cooperating threads and the management activites for concurrent exception handling (i.e., synchronization, exception resolution, and invocation of the different handlers). The base-level provides developers with classes for creating the cooperating threads of their applications, and the meta-level programming deal with management mechanisms based on reified invocations and results. So, meta-objects are responsible for synchronizing the cooperating threads, performing the exception resolution process, and invoking the handlers attached to the cooperation participants.

8. Concluding remarks

Exception handling mechanisms are becoming important features of object-oriented programming languages. They provide support at the programming language level for structuring fault tolerance activities in dependable object-oriented applications. We have reviewed and evaluated the exception handling models used in twelve existing exception mechanisms for object-oriented languages. We have also defined a set of demanding quality requirements which should be satisfied while developing an exception mechanism for dependable object-oriented software. These requirements constitute the major criteria that we have used to determine the design solutions for an ideal exception handling model.

Language features and their corresponding mechanisms for exception handling continue to evolve in both experimental and commercial object-oriented languages. However, our evaluation shows that none of the existing exception mechanisms has so far followed appropriate design criteria. Most of the existing mechanisms have adopted a number of classical design solutions for the implementation of exception handling models. Design decisions in many languages were made based on too general and complex solutions, which makes it an extremely difficult task to construct dependable object-oriented software in a well-structured fashion. Furthermore, designers of a new language do not pay enough attention to the language part that provides support for exception handling; in most cases, they usually attempt to add exception handling facilities to an existing language rather than to keep exception handling in mind at the very beginning of the process of language design. We believe that an ideal object-oriented exception model is urgently needed to guide the design of effective exception handling mechanisms.

It is worthwhile to highlight the design of the exception mechanism of the Guide language which, according to our evaluation, has reached the highest punctuation in our ranking. The exception mechanism of Eiffel is also one of the best state of practice since it is complemented with a broad range of techniques as discussed in the

Section 5. The design of exception handling mechanisms used in both languages is quite suitable for building dependable object-oriented software with effective quality attributes, but provides no support for concurrent exception handling. In fact, the main drawback of the current exception handling techniques is the lack of explicit support for concurrent exception handling. The Arche language is the only language which has contributed a lot to this area, although it has some limitations.

Acknowledgements

This work has been supported by CNPq/Brazil under grant No. 131945/98-0 for Alessandro F. Garcia, and grant No. 351592/97-0 for Cecília M.F. Rubira. Cecília is also supported by the Brazilian FINEP "Advanced Information Systems" Project (PRONEX-SAI-7697102200). Alexander Romanovsky is supported by European IST DSoS project (IST-1999-11585) and by EPSRC/UK DOTS project (GR/N24056). Dr. Jie Xu was supported by the EPSRC Flexx Project on Dependable and Flexible Software.

References

Arnold, K., Gosling, J., 1998. The Java Programming Language, 2nd Ed. Addison-Wesley, Reading, MA.

Balter, R., Lacourte, S., Riveill, M., 1994. The guide language. The Computer Journal 7 (6), 519–530.

Bass, L., Clements, P., Kazman, R., 1998. Software Architecture in Practice. Addison-Wesley, Reading, MA.

Burns, A., Wellings, A., 1996. Real-Time Systems and Their Programming Languages. Addison-Wesley, Reading, MA.

Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M., 1996. A System of Patterns: Patterns-Oriented Software. Wiley, New York.

Campbell, R., Randell, B., 1986. Error recovery in asynchronous systems. IEEE Transactions on Software Engineering SE-12 (8), 811–826.

Cristian, F., 1989. Exception handling. In: Anderson, T. (Ed.), Dependability of Resilient Computers. Blackwell Scientific Publications, Oxford, pp. 68–97.

Cristian, F., 1995. Exception handling and tolerance of software faults. In: Lyu, M. (Ed.), Software Fault Tolerance. Wiley, New York, pp. 81–107.

Cui, Q., Gannon, J., 1992. Data-oriented exception handling. IEEE Transactions on Software Engineering 18 (5), 393–401.

de Lemos, R., Romanovsky, A., 1999. Exception handling in a cooperative object-oriented approach. In: Proceedings of the 2nd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing, France.

Dony, C., 1988. An object-oriented exception handling system for an object-oriented language. In: Lectures Notes in Computer Science, vol. 322, pp. 146–161.

Dony, C., 1990. Exception handling and object-oriented programming: towards a synthesys. SIGPLAN Notices 25 (10), 322–330.

Dony, C., Purchase, J., Winder, R., 1991. Exception Handling in Object-Oriented Systems. In: ECOOP'91. Report on ECOOP'91 Workshop W4.

Drew, S., Gough, K., 1994. Exception handling: expecting the unexpected. Computer Languages 32 (8), 69–87.

Ferreira, L., 1999. An Object-Oriented Framework for Train Controllers. Master's thesis, Institute of Computing, University of Campinas, Brazil.

Garcia, A., 2000. Exception Handling in Concurrent Object-Oriented Software. Master's thesis, Institute of Computing – University of Campinas, Brazil (in English).

Garcia, A., Beder, D., Rubira, C., 1999. An exception handling mechanism for developing dependable object-oriented software based on a meta-level approach. In: Proceedings of the 10th IEEE International Symposium on Software Reliability Engineering – ISSRE'99. IEEE Computer Society Press, Boca Raton, USA.

Garcia, A., Beder, D., Rubira, C., 2000. An exception handling software architecture for developing fault-tolerant software. In: Prroceedings of the 5th IEEE High Assurance Systems Engineering Symposium – HASE 2000. IEEE Computer Society Press, Albuquerque, USA.

Gehani, N., 1992. Exceptional C or C with exceptions. Software: Practice and Experience 22 (10), 827–848.

Ghezzi, C., Jazayeri, M., 1997. Programming Languages Concepts, 3rd Ed. Wiley, New York.

Goldberg, A., Robson, D., 1983. Smalltalk-80, the Language and its Implementation. Addison-Wesley, Reading, MA.

Goodenough, J., 1975. Exception handling: issues and a proposed notation. Communications of the ACM 18 (12), 683–696.

Hof, M., Mossenbock, H., Pirkelbauer, P. 1997. Zero-overhead exception handling using meta-programming. In: Lectures Notes in Computer Science, vol. 1338, pp. 423–431.

Issarny, V., 1993. An exception-handling mechanism for parallel-object-oriented programming: toward reusable, robust distributed software. Journal of Object-Oriented Programming 6 (6), 29–40.

Johnson, R., 1997. Frameworks = Components + patterns. Communications of the ACM – Object-Oriented Application Frameworks 40 (10), 39–42.

Koening, A., Stroustrup, B., 1990. Exception handling for C++. Journal of Object-Oriented Programming 3 (2), 16–33.

Lacourte, S., 1991. Exceptions in guide, an object-oriented language for distributed applications. In: Lectures Notes in Computer Science, vol. 512, pp. 268–287.

Lang, J., Stewart, D., 1998. A study of the applicability of existing exception-handling techniques to component-based real-time software technology. ACM Computing Surveys 20 (2), 274–301.

Lee, P., Anderson, T., 1990. Fault Tolerance: Principles and Practice, 2nd Ed. Springer, Berlin.

Lieberman, H., 1987. Concurrent object-oriented programming in Act1. In: Yonezawa, A., Tokoro, M. (Eds.), Object-Oriented Concurrent Programming. MIT Press, Cambridge, MA, pp. 9–36.

Liskov, B., Snyder, A., 1979. Exception Handling in CLU. IEEE Transaction on Software Engineering SE-5 (6), 546–558.

Maclaren, M., 1977. Exception handling in PL/I. SIGPLAN Notices 12 (3), 101–104.

Madsen, O., Moller-Pedersen, B., Nygaard, K., 1993. Object-Oriented Programming in the BETA Programming Language. Addison-Wesley, Reading, MA.

Maes, P., 1987. Concepts and experiments in computacional reflection. ACM SIGPLAN Notices 22 (12), 147–155.

Meyer, B., 1988. Object-Oriented Software Construction. Prentice-Hall, Englewood Cliffs, NJ.

Meyer, B., 1992. Eiffel – The Language. Prentice-Hall, Englewood Cliffs, NJ.

Miller, R., Tripathi, A., 1997. Issues with Exception Handling in Object-Oriented Systems. In: Lectures Notes in Computer Science – ECOOP'97, vol. 1241. Springer, Berlin.

Mitchell, S., Burns, A., Wellings, A., 1998. MOPping up Exceptions. In: ECOOP'98 Workshop on Reflective Object-Oriented Programming and Systems.

Nelson, G., 1991. Systems Programming with Modula-3. Prentice-Hall, Englewood Cliffs, NJ.

Papurt, D., 1998. The use of exceptions. Journal of Object-Oriented Programming 13–17, 32.

Randell, B., Zorzo, A., 1999. Exception handling in multiparty interactions. In: Proceedings of VIII Brazilian Symposium of Fault-Tolerant Computing. Campinas, Brazil.

Renzel, K., 1997. Error Detection. In: EuroPLOP'97.

Robillard, M., Murphy, G., 2000. Designing robust java programs with exceptions. In: Proceedings of the 8th ACM International Symposium on the Foundations of Software Engineering, San Diego, USA.

Romanovsky, A., 1997. Practical exception handling and resolution in concurrent programs. Computer Languages 23 (7), 43–58.

Romanovsky, A., 2000a. An exception handling framework for n-version programming in object-oriented systems. In: Proceedings of the 3rd IEEE International Symposium on Object-Oriented Real-Time Distributed Computing – ISORC'2000. IEEE Computer Society Press, Newport Beach, USA.

Romanovsky, A., 2000b. Extending conventional languages by distributed/concurrent exception resolution. Journal of Systems Architecture 46 (1), 79–95.

Romanovsky, A., Xu, J., Randell, B., 1996. Exception handling and resolution in distributed object-oriented Systems. In: Proceedings of the 16th International Conference on Distributed Computing Systems. Hong Kong.

Romanovsky, A., Randell, B., Stroud, R., Xu, J., Zorzo, A., 1997. Implementation of blocking coordinated atomic actions based on forward error recovery. Journal of System Architecture 43 (10), 687–699.

Romanovsky, A., Xu, J., Randell, B., 1998. Exception handling in object-oriented real-time distributed systems. In: Proceedings of the International Symposium on Object-Oriented Real-time Distributed Computing. Kyoto, Japan.

Rubira, C., 1994. Structuring fault-tolerant object-oriented systems using inheritance and delegation. Ph.D. thesis, University of Computing Science.

Schaffert, C., Cooper, T., Bullis, B., Kilian, M., Wilport, C., 1986. An Introduction to Trellis-Owl. In: OOPSLA'86.

Schwille, J., 1993. Use and abuse of exceptions – 12 Guidelines for proper exception handling. In: Lectures Notes in Computer Science – Ada-Europe'93, vol. 688, pp. 142–152.

Taft, S., Duff, R., 1997. Ada 95 Reference Manual: Language and Standard Libraries, International Standard ISO/IEC 8652:1995(e). In: Lectures Notes in Computer Science, vol. 1246. Springer, Berlin.

Teixeira, S., Pacheco, X., 1999. Delphi Developer's Guide. SAMS Publishing.

van Lamsweerde, A., Letier, E., 2000. Handling obstacles in goal-oriented requirements engineering. IEEE Transactions on Software Engineering SE-26, 978–1005.

Vo, K., Wang, Y., Chung, P., Huang, Y., 1997. Xept: A software instrumentation method for exception handling. In: Proceedings of the 8th International Symposium on Software Reliability Engineering – ISSRE'97. IEEE Press, Albuquerque, NM, USA.

Xu, J., Randell, B., Romanovsky, A., Rubira, C., Stroud, R., Wu, Z., 1995a. Fault tolerance in concurrent object-oriented software through coordinated error recovery. In: FTCS-25: 25th International Symposium on Fault Tolerant Computing. Pasadena, CA.

Xu, J., Randell, B., Rubira-Calsavara, C., Stroud, R., 1995. Towards an object-oriented approach to software fault tolerance. In: Pradham, D., Avresky, D. (Eds.), Fault-Tolerant Parallel and Distributed Systems. IEEE Computer Society Press, Silver Spring, MD, pp. 226–233 (Chapter 5).

Xu, J., Romanovsky, A., Randell, B., 1998. Coordinated exception handling in distributed object systems: from model to system implementation. In: International Conference on Distributed Computing Systems, ICDCS-18.

Xu, J., Randell, B., Romanovsky, A., Stroud, R., Zorzo, A., Canver, E., von Henke, F., 1999. Rigorous development of a safe-critical system based on coordinated atomic action. In: Proceedings of the 29th IEEE International Symposium on Fault-Tolerant Computing. Madison, USA.

Xu, J., Romanovsky, A., Randell, B., 2000. Concurrent exception handling and resolution in distributed object systems. IEEE Transactions on Parallel and Distributed Systems 11 (10), 1019–1032.

Yemini, S., Berry, D., 1985. A modular verifiable exception handling mechanism. ACM Transactions on Programming Languages and Systems 7 (2), 214–243.

Zorzo, A., Romanovsky, A., Xu, J., Randell, B., Stroud, R., Welch, I., 1999. Using coordinated atomic actions to design complex safety-critical systems: the production cell case study. Software: Practice and Experience 29 (7), 1–21.