An exception handling mechanism for parallel object-oriented programming

Valérie Issarny

UNITÉ DE RECHERCHE
INRIA-RENNES

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
B.P.105
78153 Le Chesnay Cedex
France
Tél.:(1) 39 63 55 11

Rapports de Recherche

1 9 9 2

ème anniversaire

N° 1757

Programme 1
Architectures parallèles, Bases de données,
Réseaux et Systèmes distribués

AN EXCEPTION HANDLING MECHANISM FOR PARALLEL OBJECT-ORIENTED PROGRAMMING
Towards the Design of Reusable and Robust Distributed Software

Valérie ISSARNY

Octobre 1992

RR-1757

IRISA

INSTITUT DE RECHERCHE EN INFORMATIQUE
ET SYSTEMES ALEATOIRES

Campus Universitaire de Beaulieu
35042 - RENNES CEDEX FRANCE
Tél. : 99 84 71 00 - Télex : UNIRISA 950 473 F
Télécopie : 99 38 38 32

An Exception Handling Mechanism for Parallel Object-Oriented Programming

Towards the Design of Reusable and Robust Distributed Software

Valérie Issarny*

INRIA-Rennes / IRISA

Campus de Beaulieu, 35042 Rennes Cédex, FRANCE

Publication Interne n° 673 - Août 1992 - 36 pages - Programme 1

Abstract

Paradigms of parallel object-oriented programming are attractive for the design of large distributed software. They notably provide sound basis to develop applications that are easy to maintain and reuse. This paper investigates the issue of robustness for parallel object-oriented applications. An exception handling mechanism for strongly-typed, parallel object-oriented programming is introduced. The mechanism is based on a parallel exception handling model whose features enforce the development of correct and robust programs. Moreover, the proposed mechanism is defined according to object-oriented programming paradigms. In particular, the effect of exception declaration upon subtyping is addressed. Advantages of the mechanism are illustrated through an example. A reusable implementation of the two phase commit protocol is presented. Finally, an assessment of our proposal and a comparison with related work are described.

Key words: exceptions, exception handling, language constructs, parallel object-oriented programming.

*Author's present address: University of Washington, Dept. of Computer Science and Engineering, FR-35, Seattle, WA 98195, USA.

Un Mécanisme de Traitement des Exceptions pour La Programmation Parallèle à Objets

Vers la conception d'applications distribuées réutilisables et robustes

Résumé

Les caractéristiques de la programmation parallèle à objets sont attrayantes pour la conception d'applications distribuées de taille importante. En particulier, la facilité d'héritage permet la réutilisation de composants logiciels existant. Dans cet article, nous examinons la définition de constructions linguistiques pour un langage parallèle à objets, fortement typé, en vue de permettre le développement d'applications qui soient non seulement réutilisables mais aussi robustes. Plus précisément, nous introduisons un mécanisme de traitement d'exceptions qui exprime un modèle dont les caractéristiques encouragent le développement de programmes parallèles robustes corrects. Le mécanisme proposé est défini en accord avec les paradigmes de la programmation à objets. La répercussion de la déclaration des exceptions sur le sous-typage est notamment étudiée. Les avantages du mécanisme sont en outre illustrés au moyen d'un exemple. Une mise en œuvre réutilisable du protocole de validation à deux phases est introduite. Enfin, les apports de notre solution ainsi qu'une comparaison avec des travaux apparentés sont présentés en conclusion.

Mots clés: exceptions, traitement des exceptions, constructions linguistiques, programmation parallèle à objets.

1 Introduction

Besides mechanisms to deal with distribution, large distributed software design requires mechanisms for structuring data and algorithms. Furthermore, distributed applications should be easy to maintain and exhibit qualitative features such as reliability and readability. Object-oriented programming fulfills many of the above *desiderata* and also provides paradigms that greatly facilitate reusability. Expression of parallelism in object-oriented languages gave rise to many satisfying proposals (e.g., [Yonezawa et al.87]) even though some fail to integrate properly inheritance. However, parallelism makes language design more complex if features for fault tolerance are to be provided. To help the design of fault tolerant (or robust) applications, dedicated mechanisms, such as *exception handling mechanisms*, have been integrated within programming languages. Sequential exception handling has been widely examined (e.g., [Cristian84, Yemini et al.87, Bolot et al.89]) and is now a well understood subject. On the other hand, parallel exception handling has been scarcely studied and existing proposals tend to evict fundamental issues such as program correctness. Furthermore, in the framework of strongly-typed, object-oriented programming, benefits of subtyping in the presence of exceptions is often weakened; for instance, specialization of exceptions is not considered in operation redefinition. In this paper, we introduce an exception handling mechanism for strongly-typed, parallel object-oriented programming. The mechanism is based on a parallel exception handling model whose features enforce the development of correct and robust programs. Moreover, the proposed mechanism is defined according to object-oriented programming paradigms. In particular, the impact of exception specialization on subtyping is addressed. In order to illustrate our proposal, the mechanism is defined in the framework of a particular language. However, we claim that the mechanism may be integrated within other existing parallel object-oriented languages.

1.1 Exception Handling

Exception handling relies on the decomposition of operation domain into operation standard and exceptional domains. An operation (e.g., a procedure) that is invoked in an initial state belonging to the operation's *standard domain*, a subset of the operation's domain, returns a result satisfying the operation's standard output assertion. On the other hand, if an operation is invoked in an initial state outside the operation's standard domain, this state belongs to the operation's

exceptional domain. The actual detection of the exception during the operation execution leads to the exception *raising*, which is followed by the execution of a specific computation, called *exception handling.* If the exception cannot be handled within the operation where it was raised, the exception is *signalled* to the embedding environment and the operation is referred to as the *exception signaller.* A *model of exception handling* defines the interaction between a signaller and its handler. Finally, an *exception handling mechanism* defines a set of appropriate language constructs that are integrated within a programming language to express a given model of exception handling.

There exist two major sequential exception handling models [Knudsen87]: the *continuation model* and the *termination model.* In the continuation model, the signaller suspends its execution, invokes the handler and resumes its activity. In the termination model, signalling an exception causes the termination of the operation raising the exception and the subsequent execution of the handler. The main advantage of the termination model stands in its simplicity. It introduces very few primitives in the host language, that is, a command for explicit signals and a command to define exception handling scope rules. Examples of languages using the termination model are Clu [Liskov et al.79], Ada [Ada83], Modula-2 + [Rovner et al.85] and Eiffel [Meyer88]. On the other hand, the continuation model is intrinsically complex. In the pioneering proposal of [Goodenough75], three cases dependent upon the exception signaller are considered: the signaller has to be resumed, the signaller must not be resumed and resumption is optional. The more recent proposal of [Yemini et al.85] alleviated this complexity. Even though this proposal introduces a model of exception handling (called *replacement model*) as powerful as the continuation model, it requires few primitives to be added in the host programming language. However, this model still remains more complex than the termination one.

Exception handling has been addressed for various parallel programming models (e.g., [Campbell et al.86], [Huang et al.90], [Ichisugi et al.90]). To our knowledge, existing proposals were not directly concerned with correctness of robust parallel programs. Even though a proof system has been defined for a subset of Ada with exception handling [Lodaya et al.90], this work was done *a posteriori* and did not influence design choices. In this paper, we retain a model of parallel exception handling whose design was mainly guided by the correctness issue of robust parallel programs [Issarny91a]. Hence, this model provides sound basis for the design of correct programs.

1.2 Towards the Design of Reusable and Robust Distributed Applications

The chosen model of exception handling defines basic control structures to be integrated within a parallel language to program robust parallel applications. Additional features that are required to maintain all the benefits of object-oriented programming have still to be specified. For instance, potential reusability and specialization of classes should be supported in the presence of exceptions. Furthermore, from the perspective of language consistency, notions that are related to exception handling have to be defined according to object-oriented programming paradigms.

The remainder of this paper shows how provision for exception handling can be consistently integrated within a parallel object-oriented language. Section 2 briefly describes the imperative programming language Arche used as a support for our discussion. Section 3 introduces the exception handling mechanism of Arche. Section 4 then exemplifies the advantages of the proposed mechanism to design reusable robust applications. An implementation of the two phase commit protocol that can be reused for specific applications is presented. Conclusions and assessment of our proposal follow in Section 5.

2 Embedding Language

The programming language Arche [Benveniste et al.92] has been developed to simplify the construction of distributed applications. Arche applications are intended to execute on a dedicated object-based system called Gothic [Banatre et al.91]. This system transparently manages distribution and enforces features of object-oriented programming such as encapsulation. In this section, only Arche features relevant to the design of reusable and robust distributed software are discussed. The interested reader is referred to [Benveniste et al.92] for a complete definition of the language.

2.1 Types, Classes and Objects

In Arche, types are declared by means of constructors. The type constructor *view* defines an entity akin to an abstract data type. The declaration of a view $\mathcal{V}$ embeds the signatures of the methods that may be applied on objects of type $\mathcal{V}$. A *class* then defines a view implementation. Such an implementation should

declare a procedure for every method of the view. Finally, *objects* are instances of classes. A view and its implementation are exemplified hereafter.

Example 1

Usual expressions and commands of Arche are similar to the ones defined in the language Modula-2 [Wirth82]. The following view type *u_buffer* defines an interface of an unbounded buffer:

```
u_buffer =
   view
      put: (x: t) ();
      get: () (x: t);
   end u_buffer
```

A possible implementation of *u_buffer* assuming method *get* is never called when the buffer is empty, is:

```
class c_u_buffer implements u_buffer =
   var
      in, out: integer; buf: seq of t;
   procedure
      put: (x: t) () = begin in := in + 1; buf[in] := x end put;
      get: () (x: t) = begin out := out + 1; return(buf[out]) end get;
   begin
      in := 0; out := 0;
end c_u_buffer
```

Class *c_u_buffer* uses the type constructor **seq of** that declares a sequence of entities. The proposed implementation *c_u_buffer* of *u_buffer* is syntactically correct: a procedure is declared for every method embedded in *u_buffer*. Finally, the ending block of *c_u_buffer* defines a peculiar method called *initialization method*. This method is implicitly called when an instance of the class is created.

2.2 Parallelism and Synchronization

Much as in the language Pool-T [America87], parallelism in Arche relies on amalgaming the notions of object and process. An object creation leads to the creation of a process and then to the asynchronous call of the initialization method. Finally, objects communicate through synchronous method calls.

As discussed in [Kafura et al.89], expressing synchronization constraints in a decentralized way seems to be a key approach for integrating both inheritance and parallelism in an object-oriented language. This solution has therefore been retained in Arche. More precisely, synchronization relies on four points:

(*i*) Any non-initialization method call is synchronous, method acceptance by the invoked object being implicit;

(*ii*) An object executes only one method at a time, therefore implementing mutual exclusion within the object;

(*iii*) A conditional synchronization mechanism compatible with inheritance, is provided; and

(*iv*) Objects may be strongly synchronized in order to execute the same method in parallel.

The two last mechanisms are detailed in the following paragraphs.

2.2.1 Conditional synchronization

The Arche mechanism of conditional synchronization was inspired by the notion of *type states* [Strom et al.86]. A type t notably defines operations that may be applied on an entity $\mathcal{E}$ of type t. A type state of t then defines t operations that can be applied on $\mathcal{E}$ according to operations previously performed on the entity. For instance, the only operation that can be applied on a non-initialized integer variable is an assignment.

With respect to the above proposal, the Arche conditional synchronization mechanism relies on the definition of *synchronization states*. A synchronization state s of a view $\mathcal{V}$ specifies $\mathcal{V}$ methods that are accessible when an object of type $\mathcal{V}$ is in state s. Moreover, *post-states* are introduced to indicate synchronization states that may be reached from any method of a view. State transitions appear within method implementations through the use of the command **become** that names the state in which the transition occurs. A state transition becomes effective only if and when the method terminates. However, if no transition state takes place while a method executes, the previous object state remains valid. The notion of synchronization state is illustrated in the following example.

Example 2

Syntactically, synchronization states are declared in the clause *state*; post-states of methods are declared in the clause *post*; and post-states of the initialization method appear in the view header. The following type *buffer* declares a bounded buffer:

```
buffer =
   view (n: integer) { empty }
      put: (x: t) ();
      get: () (x: t);
   state
      full: { get };
      partial: { get, put };
      empty: { put };
   post
      put: { full, partial };
      get: { empty, partial };
   end buffer
```

where n is an instance parameter setting the buffer size. Type *buffer* defines three synchronization states: *full*, *partial* and *empty*. Let us examine state *full*. This state allows only execution of method *get*; any caller of *put* is blocked until the object reaches a state embedding *put*. A possible implementation of *get* whose execution leads the enclosing object either to state *empty* or *partial* is:

```
get: () (x: t) =
   begin
      out := (out + 1) mod n;
      if in = out then become empty else become partial end;
      return(buf[out]);
   end get;
```

2.2.2 Object group synchronization

The language Arche enables dynamic grouping of objects. Calling a method on an object group is to be related to the *multiprocedure* notion [Banatre et al.86] which is the outcome of a generalizing approach to the integration of parallelism and procedures undertaken in [Banatre80].

Object groups are declared through the use of the type constructor **seq of**. Methods of an object group of type **seq of** $\mathcal{V}$ are called *multi-operations*. A multi-operation signature is given by applying the constructor type **seq of** to each parameter type of the corresponding method declared in $\mathcal{V}$.

Let us consider a group $\mathcal{G}$ of buffers declared as: $\mathcal{G}$: **seq of** *buffer*. The signature of the multi-operation *put* of $\mathcal{G}$ is: *put:* (*x:* **seq of** *t*) (). Let *b* be a variable of type **seq of** *t*, a call to the multi-operation *put* of $\mathcal{G}$ is written as $\mathcal{G}$! *put*(*b*) and is carried out as follows:

- All the objects belonging to $\mathcal{G}$ are synchronized;
- Input parameters are distributed among $\mathcal{G}$'s components;
- Method *put* is executed in parallel by all the components of $\mathcal{G}$;
- Objects are synchronized and their respective contributions to the multi-operation result -if any- are collected;
- Results -if any- are built out of each component contribution and made available to the caller;
- Objects of $\mathcal{G}$ become available to execute further calls.

A multi-operation may issue a *coordinated call* [Banatre et al.86], which is a natural extension of the method call mechanism. All the group components then join together to call a multi-operation and are all synchronized. When the call is terminated, results -if any- are made available to all caller's components before their parallel activities are resumed. Naturally, the coordinated call for a single element group is the traditional method call. The relationship between a calling group and the called group is a *many in many* nesting. The strong synchronization amongst cooperating processes that is offered by the coordinated call mechanism may be related to the *barrier* notion [Jordan et al.89]. In the same way, notations are introduced to express that processes have to wait for each other prior to continue their execution. However, the barrier notation is a low level synchronization primitive while the multi-operation notion allows definition of abstract parallel computation. A multi-operation is presented below.

Example 3

For the sake of conciseness, we introduce an example which may seem contrived but which has the advantage to simply illustrate both the notions of multi-operation and coordinated call. Our goal here is to interleave the contents of two disjoint buffers B_1 and B_2 in a buffer R such that two elements of the same buffer are never contiguous within R. Furthermore, assuming that B_1 and B_2 are shared objects, contents of B_1 and B_2 should not be altered during the computation of R. In the following, B_1 and B_2 are assumed to have same cardinality.

We introduce two subtypes[1] of *buffer*: *partial_buffer* and *composed_buffer*. The former defines the type of B_1 and B_2 and the latter defines the type of R. Type ***partial_buffer*** embeds the method *partial_put* intended to be used for interleaving buffers. This method is called as a multi-operation, that is, the callee is a sequence composed of the two objects B_1 and B_2. The signature of *partial_put* is: *partial_put: (dest: composed_buffer) ()* where *dest* is the buffer within which elements are to be interleaved. Type *composed_buffer* embeds the method *composed_put* that is called to add a sequence of elements to the buffer. Signature of *composed_put* is: *composed_put: (s:* **seq of** *t) ()* where *s* is the ordered sequence of elements to be added to the buffer.

An implementation of *partial_put* is given hereafter, the enclosing class is assumed to inherit an implementation of *buffer*.

```
partial_put: (dest: composed_buffer) () =
   var
      x: t; i: integer;
   begin
      i := out;
      while i < in do
         i := (i + 1) mod n; x := buf[i];
         cocall dest ! composed_put(s[me] := x);
      end
end partial_put
```

An object executing *partial_put* iterates on the number of buffer elements. At each iteration step, the method synchronizes with the other multi-operation component through a coordinated call to *composed_put* where a method (or multi-operation) call is expressed by means of the exclamation mark, as usual. In the assignment $s[me] := x$, me is a predefined Arche variable that determines the order of the calling component within the enclosing multi-operation. The assignment means that x is assigned to the me^{th} element of the call actual parameter. Finally, we do not provide an implementation for *composed_put*, it consists in adding elements of s to the buffer.

Let us examine the relevance of the multi-operation notion from the perspective of fault tolerance. Study of error recovery in asynchronous systems has led to identify the decomposition of a parallel application into parallel sub-actions as essential for the design of fault tolerant distributed software [Campbell et al.86]. As multi-operations can be composed through coordinated calls, a parallel operation may be defined in terms of smaller ones. Furthermore, replication has

[1]Subtyping and inheritance in Arche are described in the next subsection. However, a common understanding of these notions is sufficient for the present example.

been recognized as worthwhile to enforce fault tolerance in a distributed system. Multi-operations allow simple management of copy consistency. Among other solutions, a straightforward implementation consists in invoking a multi-operation any time a replicated entity is modified, the invoked sequence being composed of all the objects that own a copy.

2.3 Subtyping and Inheritance

Subtyping and inheritance although being distinct notions [Cook et al.90] are closely related in the language Arche. Inheritance is only permitted when a class implements a subtype. The Arche subtyping relation [Benveniste et al.92], noted $<:$, follows from the subtyping relation of the programming language Modula-3 [Cardelli et al.89]. Intuitively, a type u is a subtype of a type v if u is an extension of v. In particular, a view $\mathcal{U}$ is a subtype of a view $\mathcal{V}$ if it is *explicitly* defined as being an extension of $\mathcal{V}$, that is, if it is declared as: $\mathcal{U} = \mathcal{V}$ **view** ... **end**. Moreover, subtyping of view types should satisfy constraints that are related to synchronization.

Let us examine definition of type $\mathcal{U}$, two cases have to be considered: (i) a new method m' is defined in $\mathcal{U}$; (ii) a method m of $\mathcal{V}$ is to be redefined in any implementation of $\mathcal{U}$. Definition of $\mathcal{U}$ may thus require redefinition of states $\sigma_{\mathcal{V}}$ of $\mathcal{V}$ or definition of new states $\sigma'_{\mathcal{U}}$. Rules are introduced in order to maintain the specialization notion inherent to subtyping. When redefining a state $\sigma_{\mathcal{V}}$, for instance to cope with case (i), the rule is that $\sigma_{\mathcal{U}}$ should be an extension of $\sigma_{\mathcal{V}}$. And, when adding a new state $\sigma'_{\mathcal{U}}$, for instance to cope with (ii), the rule is that there should exist a state $\sigma_{\mathcal{V}}$ in $\mathcal{V}$ such that $\sigma'_{\mathcal{U}}$ is an extension of $\sigma_{\mathcal{V}}$. Finally, new states may be added to existing post-states. The following example illustrates subtyping of a view, it is inspired by [Kafura et al.89].

Example 4

Type *buffer* of example 2 is specialized by adding method *get_rear* that returns the buffer's last element. States *partial* and *full* are extended to allow execution of *get_rear*. We get:

```
buf_queue =
    buffer view get_rear: () (x: t);
    state full: { get_rear }; partial: { get_rear };
    post get_rear: { empty, partial };
    end buf_queue
```

The assignment relation, noted $\hookrightarrow$, is defined to introduce polymorph references. The Arche assignment relation directly follows from the corresponding definition given in [Cardelli et al.89] for the language Modula-3. As an example of rule of the assignment relation, we have: *Given an expression* e *of type* $\mathcal{U}$ *and a variable* v *of type* $\mathcal{V}$, *if* $\mathcal{U} <: \mathcal{V}$ *then* e $\hookrightarrow$ v.

Inheritance facility is limited to single inheritance in Arche. Furthermore, as previously stated, the inheritance mechanism allows a class C_1 to inherit from a class C_2 only if C_1 implements a type that is a subtype of the type implemented by C_2. A subclass may freely access any of the entities declared in its superclass. Procedure redefinition is explicitly specified in the clause **redefines** of the redefining class. Finally, a redefined procedure may be called from its redefining entity through the use of the command **super**.

Up to this point, we have given an overview of the parallel object-oriented language Arche. The next section focuses on exception handling in the framework of parallel object-oriented programming.

3 Exception Handling

Exception handling within programs relies on a model that defines the interaction between a signaller and its handler. The model that we have chosen to integrate within the language Arche was primarily designed to facilitate the development of *correct* and *robust* parallel programs [Issarny91b]. This model is an extension of the termination model and introduces only three basic additions to the notions already needed to cope with sequential program exceptions. The remainder of this section describes the exception handling model and its integration within Arche.

3.1 Declaration of Exceptions

Representing exceptions as classes, as notably addressed in [Koenig et al.90] for the language C++, is essential from the perspective of language consistency. Furthermore, such an approach keeps the benefits of subtyping and inheritance in the presence of exceptions. This allows specialization of exceptions without having to systematically rewrite operations that are concerned with the specialization. For instance, a handler whose only purpose is to propagate an exception has not to be updated.

Since our target language does not offer the *metaclass* notion, we introduce a

linguistic construct specifically dedicated to the definition of exceptions. The only information pertained to an exception being its parameters, exception declaration needs no special implementation to be provided. It follows that an exception declaration defines both a type and a class. Therefore, in the remainder of this paper, the term *exception* designates either exception types, classes or objects that are instances of exception classes, when its meaning can be deduced from the context.

Declaration of an exception may be compared to the declaration of a record type. For instance, an exception e with n parameters a_i of type t_i, $1 \leq i \leq n$, is declared as:

$$e = \textbf{exception}\ a_1 \colon t_1 ; \ldots ; a_n \colon t_n\ \textbf{end}.$$

Instance of an exception class is created through the use of the signal command written as:

$$\textbf{signal}\ e(x_1, \ldots, x_n).$$

Invoking the above command leads to create an instance of the class e and then to the usual signal of the *exception* whose semantics is defined in Subsections 3.2 and 3.3. The signal of an exception e is syntactically correct if all the actual parameters are assignable to the corresponding formal parameters of the exception. A parameter a_i of an object $\mathcal{O}_e$, instance of e, is accessed by dereferencing. Finally, a parameterless exception f is simply declared as: $f =$ **exception**.

As for subtype views, declaration of a subtype exception is made by extension. For instance, an exception g, subtype of e, with l additional parameters b_i of type u_i, $1 \leq i \leq l$, is declared as:

$$g = e\ \textbf{exception}\ b_1 \colon u_1 ; \ldots ; b_l \colon u_l\ \textbf{end}.$$

The following example illustrates introduction of exceptions within a view.

Example 5

Type *buffer* of example 2 is modified according to exception handling consideration. The clause **signals** may be used in any operation signature to state the exceptions that the operation may signal. In the following declaration, the initialization method and the method *put* may signal the non-parameterized exception *e_full*:

```
buffer =
   view (n: integer) { empty } signals e_full
      put: (x: t) () signals e_full;
      get: () (x: t);
   state
      full: { get };
      partial: { get, put };
      empty: { put };
   post
      put: { full, partial };
      get: { empty, partial };
   end buffer
```

Specializing an operation may require modification of the set of exceptions that the operation may signal. Let us examine the subtyping rule defined for procedure type in the programming language Modula-3 [Cardelli et al.89].

Definition 1 (Subtyping of procedures in Modula-3) *Let t and u be two procedure types, then $t <: u$ if:*

(i) t and u have the same number of parameters and parameters that correspond have same type and mode[2]*;*

(ii) t and u have same result type or none has a result type;

(iii) the exception set of t is included within the one of u.

This definition states that a subtype procedure may *at most* signal the exceptions that are listed in the definition of its supertype. We claim that this definition is not suited for type specialization. For instance, let us consider the following subtyping relations between view types:

Earth_transport <: Transport
Air_transport <: Transport

Furthermore, let us assume that the view *Transport* embeds method *Check_engine*. This method may typically signal the exception *Failure_engine*. If we now examine subtype *Air_transport*, the exceptional domain associated to *Failure_engine*

[2]The two main modes are by *reference* and by *value*.

may be subdivided, or specialized, into three domains: *Failure_left_engine*, *Failure_right_engine*, and *Failure_both_engines*. These three exceptions are a specialization, or a subtype, of the exception *Failure_engine*. This relationship is easy to express when exceptions are types. In order to authorize expression of exception specialization within procedures, we introduce the following definition.

Definition 2 (Subtyping of procedures) *Let t and u be two procedure types, then $t <: u$ if:*

(i) t and u have the same number of parameters and parameters that correspond have same type (and mode);

(ii) t and u have same result type or none has a result type;

(iii) any exception belonging to the exception set of t should be a subtype of an exception belonging to the exception set of u.

The above definition enables specialization of exceptions that are signalled by a procedure while ensuring consistency of the type system.

In the language Arche, specialization of exception signals according to point (*iii*) of **definition 2** is provided for any operation that is not an initialization method. We did not feel the need for such a facility for initialization methods and retained the Modula-3 solution instead (point (*iii*) of **definition 1**). Syntactically, the new set of exceptions that are signalled by a redefined method is expressed in the clause *exception* of the subtype view. This clause is of the form:

$$\textbf{exception} < m_i.e_j : exception\_set_{i_j}; >$$

where brackets $< >$ are introduced to denote zero or several repetitions of the enclosed material; $m_i.e_j$ designates the exception e_j signalled by the method m_i of the supertype; and any exception belonging to $exception\_set_{i_j}$ is an exception that may be signalled by the method redefining m_i, this exception having to be a subtype of e_j.

3.2 Synchronous Exception Handling

We now define synchronous exception handling, that is, handling of exceptions signalled by methods called synchronously (e.g., Arche methods and multi-operations). Its semantics follow from the definition of our base exception handling model. Handlers are declared by means of an *exception handling command* written as:

try C **except** < e_i *(v_i): C_i* > **else** C **end**

where C is a command; e_is are exception types; v_is are (optional) identifiers that name handled exceptions; C_is are commands defining the respective handlers of e_is; and the clause **else** is optional, C being a command defining the current *default exception handler.*

The informal semantics of the exception handling command is: if an exception is raised while C executes then C terminates and the exception handler is sought within the handler list. This search is implemented according to *explicit* propagation of exceptions whose benefits are notably discussed in [Yemini et al.85, Cristian87]. If any of the e_is is a supertype of the signalled exception then the corresponding handler h_i is executed. A handler may propagate the handled exception by using the command **signal** if a variable is locally declared for the exception (i.e., v_i is specified). In this case, the command **signal** specifies the locally declared exception variable instead of an exception class, which avoids an object creation. One may notice that two exception types specified in the exception handling command may be in subtyping relation. Therefore, more than one handler may be eligible to deal with a given exception. In this case, the retained handler is the first declared in the handler list among all the eligible handlers. On the other hand, if no explicit handler is found and if a default exception handler is declared, this handler is executed. Finally, if search fails, the predefined exception *failure* is signalled. However, exception handling being static, absence of exception handlers may be detected at compile time hence allowing error report to the programmer.

The informal semantics of synchronous exception handling given so far defines the termination model with explicit propagation of exceptions. However, this definition has to be further enriched when an exception is raised within a component of a parallel operation. The base model of exception handling introduces the notions of *global exception*, *exception catching*, and *concerted exception* to cope with this language feature.

3.2.1 Exception handling and nested parallel operations

For illustration purpose, we rewrite example 3 that deals with buffer interleaving. We relax our previous assumption setting that the two buffers to be interleaved have same cardinality. The method *partial_put* that defines multi-operation components now signals the non-parameterized exception *e_empty* when it reaches the end of the buffer. We get:

```
partial_put: (dest: composed_buffer) () signals e_empty =
   var
      x: t; i: integer;
   begin
      i := out;
      while i < in do
         i := (out + 1) mod n; x := buf[i];
         cocall dest ! composed_put(s[me] = x);
      end;
      signal e_empty;
end partial_put
```

The notions of *global exception* and *exception catching* are introduced to avoid deadlock subsequent to an exception occurrence. For instance, let us assume that in the above example, the two buffers to be interleaved do not have same cardinality. It follows that one multi-operation component terminates exceptionally by signalling *e_empty* while the other becomes blocked at the coordinated call, waiting for a communication with the terminated component. An exception whose occurrence prevents achievement of an expected synchronous communication causes the exceptional termination of its signaller. Such an exceptional termination is expressed by signalling a *global exception* indicating that the signaller failed to ensure an expected cooperation with at least one other process. A process then *catches* a global exception only if and when it communicates synchronously with the global exception signaller. The handler of the exception is sought within the catching process as in the sequential case (i.e., the exception is defined locally to the process). In our example, the exception *e_empty* signalled by a multi-operation component is typically a global exception; its occurrence may prevent further synchronization with the other multi-operation component. The non-signalling component catches the exception *e_empty* when it jointly calls *composed_put*. As a result of exception handling, it may signal an exception that specifies unprocessed elements of the buffer.

Finally, the notion of *concerted exception* is related to exception handling in the presence of *nested* parallel operations. The exceptional termination of at least one component of a nested parallel operation causes the exceptional termination of the operation. The exception signalled by the operation has to be defined since many components of the operation may concurrently signal an exception. As notably addressed in [Campbell et al.86], the occurrence of several exceptions may be symptomatic of an exceptional state, dependent upon the composition of all the signalled exceptions. A *concerted exception* is the resulting exception. The

computation of a concerted exception may in general not be de defined implicitly because it requires semantic knowledge about exceptions. A dedicated mechanism is therefore included in the definition of the exception handling mechanism. Such a mechanism is presented in the next paragraph. However, an implicit solution may be retained when exceptional domains are not to be precisely characterized. The default value of a concerted exception is defined as follows:

Definition 3 (Default concerted exception) *The default value of a* concerted exception *is:*

(*i*) *the predefined non-parameterized exception* failure *if at least two components signal either a different exception or a parameterized exception;*

(*ii*) *the exception* e *if either all the signalling components signal* e *and* e *is a non-parameterized exception or there is only one signalling component that signals* e.

Coming back to our example, two situations have to be considered depending upon whether the two buffers have same cardinality or not. In the first case, multi-operation components both signal the non-parameterized exception *e_empty*. According to **definition 3**, the multi-operation signals *e_empty*. On the other hand, if the two buffers do not have same cardinality and assuming that the component handling the largest buffer signals an exception (different from *e_empty*), the multi-operation signals the predefined exception *failure*.

Computation of a concerted exception is also required in the presence of a synchronous multiparty communication model (e.g., multiway rendezvous, coordinated call). Let us consider a coordinated call issued by a multi-operation composed of more than two components. If at least two of the caller components signal an exception, components actually participating to the call catch more than one exception. A concerted exception is then locally computed within each of the catching components.

3.2.2 The notion of resolution function

Using **definition 3** for computing a concerted exception may sometimes be too restrictive. In [Campbell et al.86], the use of an exception tree has been suggested. Exceptions that may be signalled by components of a parallel block $\mathcal{B}$ are organized in a tree structure whose root is the *universal exception*, that is, the exception characterizing the whole exceptional domain of $\mathcal{B}$. When at least

one component of $\mathcal{B}$ signals an exception, the concerted exception signalled by $\mathcal{B}$ is equal to the smallest subtree encompassing all the concurrently signalled exceptions. Using an exception tree to evaluate concerted exceptions is well suited in many cases. Nevertheless, it does not address composition of *parameterized* exceptions. An alternative solution which would not require the definition of a specific mechanism consists in executing a distributed agreement protocol. The advantage of this solution is that it permits to determine the condition under which a parallel operation terminates, according to final states of all the operation's components. Nonetheless, specifying a distributed protocol is a source of complexity for the programmer and is not needed when all the operation components terminate normally. In the following, we propose a new mechanism that enables taking exception parameters into account for the computation of concerted exceptions.

Our proposal relies on the definition of *resolution functions* within classes. A resolution function takes a sequence of exceptions[3] as input parameter and returns an exception. To allow compile time checking, the header of any resolution function should carry information about the exceptions that the function handles and signals. We suggest the following syntactic form:

$$R \textbf{ handles } l_h \textbf{ signals } l_s = C \textbf{ end}$$

where R is the function name, l_h and l_s are sets of exception types and C is a command defining the function body. This declaration states that the function R may take as input any sequence of exceptions provided that the dynamic type of each exception is a subtype an element belonging to l_h; R then returns an exception whose type is a subtype of a type belonging to l_s. The formal parameter of any resolution function is implicitly declared, it is named *exc_seq* and is of type **seq of** *exception* where *exception* is the root of the exception type hierarchy. The body of a resolution function is defined much like a procedure body. However, it may use a particular command to identify the dynamic type of the input exception sequence elements. This command is similar in essence to the well known type case command. Finally, let us indicate that a resolution function may read (and only read) state variables of its declaring class. This facility permits to take object state into account when computing a concerted exception.

[3]Concurrently signalled exceptions are said to be grouped within a *sequence* due to the use of sequences in the embedding language Arche. However, this is not a prerequisite of our proposal, exceptions could be grouped within any ordered dynamic structure.

The use of a resolution function can be stated in any exception handling command. In such a case, the resolution function is implicitly invoked when the execution of a multi-operation invoked within the command results in the signal of an exception by at least one of the multi-operation components. The value carried by the i^{th} element of the resolution function actual parameter is then defined as follows: if the i^{th} operation component signals an exception e then the i^{th} element refers to e; on the other hand, if this component does not signal any exception, the i^{th} element refers to the peculiar *exception "terminated"* indicating that the component terminated normally. The notion of resolution function is illustrated below.

Example 6

Let us consider the management of a sequence of buffers, called *seq_buf*. Buffer sizes can be different. Method *put* signals *e_full* when there is no more place left once the element has been appended. Calling the multi-operation *put* on the sequence *seq_buf* may result in concurrent signals of the exception *e_full*. A possible type for the resulting concerted exception may be *seq_full* whose parameter identifies all the full buffers:

seq_full = **exception** *b:* **seq of** *buffer* **end**.

The following resolution function *resol_put* computes the concerted exception signalled by the multi-operation *put* of *seq_buf*. This function returns an exception of type *seq_full* if all the signalling components of the multi-operation signal *e_full*, it returns *failure* otherwise.

```
resol_put handles e_full signals seq_full, failure =
   var
      i: integer := 0; c: boolean := true;
      buf_full: seq of buffer := <>;
   begin
      while (i < exc_seq ! length()) and c do
         i := i + 1;
         exception case exc_seq[i] of
            e_full: buf_full ! append(seq_buf[i]);
            terminated: skip;
            else c := false
         end;
      end;
      if c then signal seq_full(buf_full) else signal failure end
end resol_put
```

The command **exception case** is used to test the dynamic type of an exception object. Finally, a call to the multi-operation *put* of *seq_buf* may be expressed as:

```
try using resol_put
   seq_buf ! put(s)
except
   seq_full (e): Handler of seq_full
   else Default handler
end
```

where s is a sequence of elements of type t.

Resolution functions may be redefined within subclasses. Let us introduce a function S redefining R:

$$S \textbf{ handles } l'_h \textbf{ signals } l'_s = C' \textbf{ end}$$

Conditions under which this redefinition is correct are inspired by the subtyping rule defined on function types in [Cardelli et al.85]. The header of S should satisfy the following requirements:

(i) $\forall e \in l_h, \exists f \in l'_h$ such that $e <: f$ (*contravariance*);
(ii) $\forall e \in l'_s, \exists f \in l_s$ such that $e <: f$ (*covariance*).

Up to this point, we have defined synchronous exception handling. The hierarchical relationship that has been assumed is related to the common procedural abstraction: the caller is blocked until the callee terminates. In the presence of asynchronous operation calls, this relationship may no longer be assumed. Furthermore, the result of an operation called asynchronously is not necessarily relevant for the caller. Exception handling in this framework is discussed hereafter through Arche initialization method calls.

3.3 Asynchronous Exception Handling

Two solutions are introduced to search handlers of asynchronous exceptions. One is implicit and is chosen by default; it states that an asynchronous exception is propagated to the calling object. The other solution, which is explicit, enables statement of the objects to which an asynchronous exception is to be propagated. Let us consider the creation of an instance of a class, say *c_buffer*, that implements *buffer* (see example 5):

$$\mathcal{O} := \textbf{new c\_buffer(n)}.$$

According to the definition of *buffer*, the initialization method may signal the exception *e_full*. If *e_full* is signalled, the exception is propagated by default to the object that created $\mathcal{O}$. However, the exception may be signalled to different objects by expressing "*e_full: handlers*" in the list of actual parameters, *handlers* being of type *seq of* T where T is a view. The sequence *handlers* then defines the sequence of objects to which the exception *e_full* is to be propagated. For that creation to be valid, the exception *e_full* must be handled by any object of type T. To allow compile-time checking, exceptions to be handled by any implementation of a view are stated in the view header whose general form becomes:

$T =$ **view** (*formal parameters*) {*post-states*}
signals *exceptions* **handles** *exceptions*.

Asynchronous exception signals are carried out as follows. Exceptions are sent asynchronously to all the handling objects. From the standpoint of handling objects, exception signals are processed as incoming method calls and are mutually exclusive. However, execution of handlers cannot be controlled by synchronization states. Furthermore, should a handling object attempt to call a method of the signalling object, it catches the predefined, non-parameterized exception *async_exc*.

Finally, declaration of asynchronous exception handlers within classes is separated from that of procedures. The handlers are declared in an exception handling command (see Subsection 3.2) that encapsulated the initialization method body.

Our definition of asynchronous exception handling may be compared to the one proposed in [Ichisugi et al.90] for an actor-based language. In the same way, the programmer can specify objects to which an asynchronous exception is to be propagated. However, due to the nature of the considered target language, the above proposal does not address issues related to strong typing.

4 Example: Two Phase Commit Protocol

This section highlights the benefits of the proposed exception handling mechanism for the design of reusable and robust distributed applications. An Arche implementation of the two phase commit protocol is presented and reusability of the proposed implementation is briefly sketched. In the following, the reader is assumed to be familiar with algorithms of the two phase commit protocol whose detailed description may notably be found in [Gray78].

Let us assume a distributed action, qualified as *recoverable*, that has established recovery points and that wants to commit its computation after having modified data located on different nodes. Nodes participating to the action are called *participants*. Each of these nodes is supposed to be able to commit and abort the part of the action it performed. Finally, a particular node, called *coordinator*, is assumed to be associated to the recoverable action. This last node can access all the action participants. The two phase commit protocol ensures that all the participants either commit or abort their participation to the recoverable action.

The following subsections propose an Arche description of the two phase commit protocol.

4.1 Predefined Types and Classes

The predefined Arche class *c_timer* is used to detect timeout within participants. This class takes as input parameter, an object $\mathcal{O}$ and an integer Δt. An instance of *c_timer* calls method *alarm* of $\mathcal{O}$, Δt units of time after it was created. Type *timer* implemented by *c_timer* is defined as:

timer = **view** ($\mathcal{O}$: alarm , Δt: **integer**) **end** timer

where type *alarm* is also an Arche predefined type defined as:

```
alarm =
   view { awake }
      alarm: ()();
   state
      awake: { alarm };
      asleep: { };
   post
      alarm: { asleep };
   end alarm
```

Finally, the predefined exception *timeout* is used. It characterizes a non-parameterized exception that may be signalled by the run-time system when a required communication has not been be performed for a certain amount of time.

4.2 Types

Among types defined to implement the two phase commit protocol, there are the enumeration type *action*, the exception type *nok* and the two view types *coordinator* and *participant*. These two last types respectively describe the coordinator and participant interfaces. We define:

```
action = {a_begin, a_ok, a_valid, a_rec };
nok = exception;
coordinator =
   view { exec }
      add_part: (part: participant) ();
      commit: () () signals nok;
   state
      exec: { add_part, commit };
      terminated: { };
   post
      add_part: { exec };
      commit: { terminated };
   end coordinator;
participant =
   alarm view (delay: integer) { phase1 }
      vote: () () signals nok;
      commit: () ();
      recovery: () ();
   state
      phase1: awake: { vote };
      rec: awake: { recovery };
      phase2: awake: { recovery, commit };
   post
      alarm: { phase1, rec, phase2 };
      vote: { rec, phase2 };
      commit: { awake };
      recovery: { awake };
   end participant;
```

The type *coordinator* embeds two methods: *add_part* and *commit*. The former is used to register a new participant involved in the recoverable action, the latter is invoked to request action validation. Synchronization states declared within *coordinator* are *exec* and *terminated*. State *exec* holds as long as the recoverable action is not to be validated. When validation of the recoverable action is achieved, the coordinator is in state *terminated* and hence may no longer be invoked.

Type *participant* is a subtype of *alarm*. Thus, any participant may use an object of type *timer* to be aware of timeout occurrence during the first phase of the protocol. The timeout value is passed to any participant at creation time through parameter *delay*. The first phase of the protocol is implemented by the method *vote* and the second phase by methods *commit* and *recovery*. Synchronization states *phase1* and *phase2* characterize which phase of the protocol the object is

ready to execute. As state *phase2*, synchronization state *rec* indicates that the object is ready to perform the second phase of the protocol. However, being in state *rec* furthermore implies that the participant has replied in favor of action abortion to the coordinator and thus will not execute *commit*. Let us remark here that the definition of synchronization states satisfies subtyping requirements given in Subsection 2.3; states are all defined by extension of a state (i.e., *awake*) of the supertype *alarm*, and post-states of *alarm* have been extended.

4.3 Implementation of Coordinator

Class *c_coordinator* that defines an implementation of *coordinator* is declared as:

```
class c_coordinator implements coordinator =
   type exc_vote = exception p_ok: seq of participant end;
   var p: seq of participant := <>;
   resolution
      res_vote handles nok signals exc_vote =
         begin Later detailed end res_vote;
   procedure
      write_log: (a: action) () = begin end write_log;
      add_part: (part: participant) () = begin p ! append(part) end add_part;
      commit: () () signals nok = begin Later detailed end commit;
   begin
      become exec;
end c_coordinator
```

Participants of the recoverable action are registered in the sequence variable *p*. Procedure *write_log* aims at recording performed actions within a log and thus is specific to any recoverable action. In the proposed class, procedure *write_log* may be compared to a *virtual procedure* though not enforced by the programming language; the implementation of *write_log* is to be provided by subclasses of *c_coordinator*. Procedure *add_part* is straightforward; it appends the newly involved participant to sequence *p*. Let us now examine validation of a recoverable action. This operation is carried out through the procedure *commit*.

The procedure *commit* whose declaration is given hereafter, first logs the fact that the recoverable action is in the first phase of validation. It then invokes the multi-operation *vote* of *p*, that is, *vote* is concurrently executed by each of the action participants. Due to some failure, none of the participants may be reachable. In this case, the exception *timeout* is signalled to the coordinator by the run-time system. A component of the multi-operation *vote* may either

terminate normally or signal *nok* (see *participant*). Signalling *nok* means that the participant wants to abort its participation to the action.

```
commit: () () signals nok =
   var ok: boolean := false;
   begin
      (* Phase 1 *)
      try using res_vote
         write_log(a_begin); become terminated;
         p ! vote (); write_log(a_valid);
      except
         timeout: write_log(a_rec); p ! recovery(); signal nok;
         exc_vote (e): write_log(a_rec); e.p_ok ! recovery(); signal nok;
         else write_log(a_rec); p ! recovery(); signal nok;
      end;
      (* Phase 2 *)
      try p ! commit();
      except timeout: p ! commit();
      end;
end commit;
```

Let us first consider that all the participants terminate normally, that is, the action may commit. In this case, the second phase consists in logging validation and in calling the multi-operation *commit* on *p*. This call is enclosed in an exception handling command because the exception *timeout* may still occur, for instance, due to failure of the underlying communication medium. Notice that if a subset of the multi-operation components is reachable, this causes concurrent signals of exception *timeout* by the remaining components. According to the semantics of the underlying exception handling mechanism, a concerted exception is computed. However, there is no need for a resolution function here; **definition 3** may be applied. Since exception *timeout* is not parameterized, the default concerted exception will always be an instance of *timeout*.

Consider now that at least one of the components of the multi-operation *vote* signals *nok*. The resulting concerted exception is computed by means of the resolution function *res_vote* that always signals *exc_vote*. More precisely, the resolution function *res_vote* given below discards participants that signal *nok*. It follows that the handler of *exc_vote* sends only message *recovery* to nodes which either did not reply to the coordinator or acknowledged for validation. Let us recall here that the formal parameter of any resolution function is the sequence of exceptions *exc_seq*. Furthermore, the actual parameter of any resolution function contains as many elements as the signalling multi-operation embeds components;

the *exception* associated to a node that terminates normally (i.e., that expects to validate its participation) being of type *terminated.*

```
res_vote handles nok signals exc_vote =
   var i: integer := 0; part: seq of participant := <>;
   begin
      while i < exc_seq ! length() do
         exception case exc_seq[i] of
            nok: skip
            else part ! append(p[i])
         end;
         i := i + 1;
      end;
      signal exc_vote(part);
end res_vote;
```

Finally, for the sake of brevity, re-emission of messages has been omitted in the proposed algorithm even though the exception *timeout* may be signalled by multi-operations called within handlers.

4.4 Implementation of Participants

The class given hereafter defines an implementation of *participant.*

```
class c_participant implements participant =
   var
      my_action: action := a_begin;
      alarm_clock: timer;
   procedure
      write_log: (a: action) () = begin end write_log;
      my_vote: () () signals nok = begin if my_action = a_rec then signal nok end end my_vote
      alarm: () () =
         begin
            if my_action = a_begin then write_log(a_rec); my_action := a_rec end
         end alarm;
      vote: () () signals nok =
         begin
            try
               my_vote(); write_log(a_ok); my_action := a_ok; become phase2;
            except
               nok (e): write_log(a_rec); my_action := a_rec; become rec; signal e;
               else write_log(a_rec); my_action := a_rec; become rec; signal nok;
            end
         end vote;
```

```
        commit: () () =
            begin
                write_log(a_valid); my_action := a_valid; become awake;
            end commit;
        recovery: () () =
            begin
                if my_action = a_ok then write_log(a_rec); my_action := a_rec end;
                become awake;
            end recovery;
    begin
        become phase1; alarm_clock := new c_timer(self, delay);
end c_participant
```

The object *alarm_clock*, instance of the predefined class *c_timer*, enables instances of class *c_participant* to be aware of delay expiration during the protocol first phase. The creation of *alarm_clock* within the initialization method (i.e., **new** *c_timer*(*self, delay*)) specifies the predefined state variable *self* as actual parameter. This variable references the calling object, that is, the enclosing participant. Finally, let us recall that the variable *delay* is an instance variable whose value is passed as argument when the instance of *c_participant* is created.

View types and classes defined in this section may be reused to implement a dedicated two phase commit protocol. Nonetheless, they have to be specialized to define the implementation of *write_log* and *my_vote*, which is specific to the focused distributed action. It is interesting to note that the exception *nok*, originally signalled by *my_vote* may be specialized to provide more information about the cause of action failure. In such a case, procedure *vote* has not to be modified: the variable *e* declared in *nok*'s handler may be any exception whose type is a subtype of *nok*. On the other hand, procedure *commit* of *c_coordinator* would have to be redefined; specific handling of the exception is strongly dependent upon the considered application.

5 Conclusion

In this paper, we have presented a mechanism of exception handling for a strongly-typed, parallel object-oriented language. Even though our proposal has been sketched in the framework of a particular language, we believe that the mechanism may be retained for other existing strongly-typed, parallel object-oriented languages. The choice of the embedding language was primarily motivated by the fact that the language integrates inheritance and parallelism in a satisfying way,

and offers a means to declare nested parallel operations. This last feature has indeed been recognized as a useful tool for the design of fault tolerant software in asynchronous systems [Campbell et al.86].

5.1 Summary

Definition of exceptions according to paradigms of object-oriented programming has been discussed. From the perspective of language consistency, exceptions should be declared as classes. Such a representation of exceptions furthermore enables specialization of exceptions through the use of subtyping. To keep all the benefits of this approach, the programmer should be able to reflect specialization of exceptions when he/she redefines operations. To our knowledge, existing proposals do not allow such a reflection. We have modified the base subtyping relation to provide this facility.

We have then defined synchronous and asynchronous exception handling for a strongly-typed, parallel object-oriented language. The proposed exception handling mechanism relies on the model defined in [Issarny91b]. This model is primarily based on an extension of the termination model designed for sequential languages. Concerning synchronous exception handling, the model introduces the notions of global exception and exception catching in order to avoid deadlocks subsequent to an exception occurrence. When a process P cannot execute an expected synchronous communication, it signals a global exception e. The global exception e is then caught by any process trying to communicate with P. The cooperation model also defines the notion of concerted exception. Concerted exception handling enables to cope with global exception occurrences in the presence of processes belonging to a nested (parallel) block and of multiparty synchronous communications. A concerted exception results from the composition of global exceptions that are concurrently signalled. Finally, in our base model, exceptions signalled by operations called asynchronously are propagated to remote processes.

The exception handling mechanism that we have defined introduces very few commands within the host language. In addition to usual sequential exception handling commands, it requires only syntactic means to compose concurrently signalled exceptions. This last facility has led us to introduce the notion of *resolution function* that enables composition of parameterized exceptions. To our knowledge, proposals that consider composition of exceptions cope only with non-parameterized exceptions. In order to maintain the advantages of inheritance and

subtyping, redefinition of resolution functions has been studied. We have enriched the base subtyping relation to set conditions under which the redefinition of a resolution function is statically correct.

Finally, we have exemplified the proposed exception handling mechanism through an implementation of the two phase commit protocol. The proposed implementation can easily be reused for specific applications of the protocol.

5.2 Related Work

Our semantics of exception handling for exceptions signalled by components of parallel operations may be compared to the proposal of [Levin77]. In the same way, exceptions signalled by processes (operation components in our presentation) are propagated to other processes. However, occurrence of those exceptions does not lead to termination, only resumption is provided. The resumption facility being very tied to the notion of remote procedure calls, we think that it is more advisable to include such a mechanism in a parallel language. In the mechanism of [Levin77], a process must rely on "sequential" exception handling to terminate. We believe that termination facility at "processes level" is an important feature since it enables expressing proper termination of a set of processes in a straightforward manner. As a consequence, this is a useful mechanism for avoiding deadlocks. Finally, other related mechanisms are those of [Ada83, Szalas et al.85, Huang et al.90]. In the mechanism of [Ada83], a communication with a process that terminates exceptionally leads to catch the predefined exception *tasking_error*. In our base model, the process catches a global exception and has therefore a greater knowledge about the cause of the process exceptional termination. The two other mechanisms do not define precisely the control points where a global exception can be caught. In our opinion, this feature compromises the verifiability of the mechanism. Exception handling mechanisms dealing with nested parallel blocks also adopt a generalization of the termination model [Campbell et al.86, Jalote et al.86, Taylor86] but to our knowledge have not been defined formally.

The proposed semantics of asynchronous exception handling may be compared to the one defined in [Ichisugi et al.90] for an actor-based language. The object to which an asynchronous exception is to be propagated may be explicitly stated. However, our proposal additionally addresses issues related to strong typing.

5.3 Other Issues

Besides the integration sketched in this paper, the Arche base model of exception handling has been integrated in a CSP-like language [Banatre et al.92a] and in a simple programming language based on the *multiprocedure* notion [Issarny91a], a proof system for the resulting programming languages being proposed in the mentioned references. The definition of proof systems is worth mentioning; it demonstrates that features of the model enforce the design of correct and robust parallel programs. This property mainly results from the precise definition of global exception catching. As a global exception can only be caught at a communication point, precise properties about the state of a process catching an exception may be determined. Considering resulting proof systems, the notation that has been used to deal with exception occurrences is the one proposed in [Cristian84], exception catching being taken into account within the proof rules defining communication commands. As formal proof of robust program correctness is beyond the scope of this paper, relevance of the model for this issue is not detailed further. To get a deeper insight, the interested reader may consult [Issarny91b] and the above references.

Concerning implementation issues, a compiler for the language Arche integrating the proposed mechanism of exception handling has been implemented in the framework of the Gothic INRIA/Bull project at the research institute IRISA. The compiler generates C code that is intended to execute above the object-based system Gothic [Banatre et al.92b]. The system Gothic notably provides complex built-in operations to help management of Arche parallel features (e.g., multi-operation coordinated call). Finally, experiments have been made to investigate if the language Arche is appropriate to program robust distributed applications. This has led to encouraging results. In addition to the example discussed in this paper, an application based on the technique of N-version programming has been designed. This last example also uses the facilities of exception handling and multi-operation. Summarizing the combined advantages of these facilities, the notion of multi-operation provides a useful tool to simply express management of distributed data structures while the exception handling mechanism allows keeping these data consistent in the presence of failure.

ACKNOWLEDGMENTS: The author wishes to express her gratitude to J.P. Banâtre who directly contributed to some of the research results expressed in this paper, for his useful comments on an earlier version of this paper. She also would like

to thank M.C. Gaudel, M. Jegado, P. Le Guernic, D. Le Metayer, I. Puaut and J.P. Routeau for helpful discussions on the subject of this paper.

References

[Ada83] Ada. – *The Programming Language ADA.* – Springer Verlag, 1983, *Lecture Notes in Computer Science*, volume 155.

[America87] America (P.). – Pool-T: a parallel object-oriented language. *In : Object-Oriented Concurrent Programming*, pp. 199–220. – MIT Press, 1987.

[Banatre et al.86] Banâtre (J. P.), Banâtre (M.) and Ployette (F.). – The concept of multi-functions, a general structuring tool for distributed operating system. *In : Proceedings of the Sixth Distributed Computing Systems Conference.*

[Banatre et al.91] Banâtre (J. P.) and Banâtre (M.), editors. – *Les systèmes distribués : l'expérience du système Gothic.* – InterEditions, 1991.

[Banatre et al.92a] Banâtre (J. P.) and Issarny (V.). – *An Exception Handling Mechanism for Communicating Sequential Processes: Design, Verification, and Implementation.* – Research Report 660, Rennes, France, IRISA, 1992.

[Banatre et al.92b] Banâtre (M.), Belhamissi (Y.), Bryce (C.), Puaut (I.) and Routeau (J. P.). – *Gothic: A Distributed Object-Based System.* – Research Report, Rennes, France, IRISA, 1992. In preparation.

[Banatre80] Banâtre (J. P.). – *Contribution à l'étude de méthodes et d'outils de construction de programmes parallèles et fiables.* – Rennes, France, Thèse d'état, Université de Rennes I, 1980.

[Benveniste et al.92] Benveniste (M.) and Issarny (V.). – *Arche : un langage parallèle à objets fortement typé.* – Research Report 642, Rennes, France, IRISA, 1992.

[Bolot et al.89] Bolot (J. C.) and Jalote (P.). – Formal verification of programs with exceptions. *In : Proceedings of the Nineteenth Symposium on Fault Tolerant Computing*, pp. 283–290.

[Campbell et al.86] Campbell (R. H.) and Randell (B.). – Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, vol. SE-12 (8), 1986, pp. 811–826.

[Cardelli et al.85] Cardelli (L.) and Wegner (P.). – On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, vol. 17 (4), 1985, pp. 471–516.

[Cardelli et al.89] Cardelli (L.), Donahue (J.), Jordan (M.), Kalsow (B.) and Nelson (G.). – The Modula-3 type system. *In : Proceedings of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pp. 202–212.

[Cook et al.90] Cook (W. R.), Hill (W. L.) and Canning (P. S.). – Inheritance is not subtyping. *In : Proceedings of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pp. 125–135.

[Cristian84] Cristian (F.). – Correct and robust programs. *IEEE Transactions on Software Engineering*, vol. SE-10 (2), 1984, pp. 163–174.

[Cristian87] Cristian (F.). – *Exception Handling.* – Research report RJ 5724, Palo Alto, California, USA, IBM Almaden Research Center, 1987.

[Goodenough75] Goodenough (J. B.). – Exception handling issues and a proposed notation. *Communications of the ACM*, vol. 18 (12), 1975, pp. 683–696.

[Gray78] Gray (J.). – *Notes on DataBase Operating Systems.* – Springer Verlag, 1978, *Lecture Notes in Computer Science*, volume 60.

[Huang et al.90] Huang (D. T.) and Olsson (R. A.). – An exception handling mechanism for SR. *Computer Language*, vol. 15 (3), 1990, pp. 163–176.

[Ichisugi et al.90] Ichisugi (Y.) and Yonezawa (A.). - Exception handling and real time features in an object-oriented concurrent language. *In : Proceedings of the UK/Japan workshop on Concurrency: Theory, Language, and Architecture.* pp. 604–615. - Springer-Verlag.

[Issarny91a] Issarny (V.). - An exception handling model for parallel programming and its verification. *In : Proceedings of the ACM SIGSOFT'91 Conference on Software for Critical Systems*, pp. 92–100.

[Issarny91b] Issarny (V.). - *Un modèle pour le traitement des exceptions dans les programmes parallèles.* - Rennes, France, Thèse de doctorat, Université de Rennes I, 1991.

[Jalote et al.86] Jalote (P.) and Campbell (R. H.). - Atomic actions for fault tolerance using CSP. *IEEE Transactions on Software Engineering*, vol. SE-12 (1), 1986, pp. 59–68.

[Jordan et al.89] Jordan (H.), Benten (M.), Alaghband (G.) and Jakob (R.). - The Force: A highly portable parallel programming language. *In : Proceedings of the International Conference on Parallel Processing*, pp. II–112–II–117.

[Kafura et al.89] Kafura (D. G.) and Lee (K. H.). - Inheritance in actor based concurrent object-oriented languages. *The Computer Journal*, vol. 32 (4), 1989, pp. 297–303.

[Knudsen87] Knudsen (J. L.). - Better exception handling in block structured systems. *IEEE Software*, vol. 17 (2), 1987, pp. 40–49.

[Koenig et al.90] Koenig (A.) and Stroustrup (B.). - Exception handling for C++ (revised). *In : Proceedings of the Usenix C++ Conference*, pp. 149–176.

[Levin77] Levin (R.). - *Program Structures for Exceptional Condition Handling.* - Pittsburgh, Pensylvania, USA, PhD thesis, Carnegie-Mellon University, 1977.

[Liskov et al.79] Liskov (B. H.) and Snyder (A.). - Exception handling in Clu. *IEEE Transactions on Software Engineering*, vol. SE-5 (6), 1979, pp. 546–558.

[Lodaya et al.90] Lodaya (K.) and Shyamasundar (R. K.). – Proof theory for exception handling in a tasking environment. *Acta Informatica*, vol. 28, 1990, pp. 365–397.

[Meyer88] Meyer (B.). – *Object-Oriented Software Construction.* – Prentice-Hall International, 1988.

[Rovner et al.85] Rovner (P.), Levin (R.) and Wick (J.). – *On Extending MODULA-2 for Building Large, Integrated Systems.* – Technical report, Palo Alto, California, USA, Digital Systems Research Center, 1985.

[Strom et al.86] Strom (R. E.) and Yemini (S.). – Typestate: A programming language concept for enhancing software reliability. *IEEE Transactions on Software Engineering*, vol. SE-12 (1), 1986, pp. 157–171.

[Szalas et al.85] Szalas (A.) and Szczepanska (D.). – Exception handling in parallel computations. *ACM SIGPLAN Notices*, vol. 20 (10), 1985, pp. 95–104.

[Taylor86] Taylor (D. J.). – Concurrency and forward error recovery in atomic actions. *IEEE Transactions on Software Engineering*, vol. SE-12 (1), 1986, pp. 69–78.

[Wirth82] Wirth (N.). – *Programming in Modula-2.* – Springer Verlag, 1982.

[Yemini et al.85] Yemini (S.) and Berry (D. M.). – A modular verifiable exception handling mechanism. *ACM Transactions on Programming Languages and Systems*, vol. 7 (2), 1985, pp. 214–243.

[Yemini et al.87] Yemini (S.) and Berry (D. M.). – An axiomatic treatment of exception handling in an expression oriented language. *ACM Transactions on Programming Languages and Systems*, vol. 9 (3), 1987, pp. 390–407.

[Yonezawa et al.87] Yonezawa (A.) and Tokoro (M.), editors. – *Object-Oriented Concurrent Programming.* – MIT Press, 1987.

LISTE DES DERNIERES PUBLICATIONS INTERNES PARUES A L'IRISA

PI 661 REACHABILITY ANALYSIS ON DISTRIBUTED EXECUTIONS
Claire DIEHL, Claude JARD, Jean-Xavier RAMPON
Juin 1992, 18 pages.

PI 662 RECONSTRUCTION 3D DE PRIMITIVES GEOMETRIQUES PAR VISION ACTIVE
Samia BOUKIR, François CHAUMETTE
Juin 1992, 40 pages.

PI 663 FILTRES SEMANTIQUES EN CALCUL PROPOSITIONNEL
Raymond ROLLAND
Juin 1992, 22 pages.

PI 664 REGION-BASED TRACKING IN AN IMAGE SEQUENCE
François MEYER, Patrick BOUTHEMY
Juin 1992, 50 pages.

PI 665 CORRECTNESS OF AUTOMATED DISTRIBUTION OF SEQUENTIAL PROGRAMS
Cyrille BARREAU, Benoît CAILLAUD, Claude JARD, René THORAVAL
Juin 1992, 32 pages.

PI 666 AGREGATION FAIBLE DES PROCESSUS DE MARKOV ABSORBANTS
James LEDOUX, Gerardo RUBINO, Bruno SERICOLA
Juillet 1992, 30 pages.

PI 667 MODELES D'EVALUATION DE LA FIABILITE DU LOGICIEL ET TECHNIQUES DE VALIDATION DE SYSTEMES DE PREDICTION : ETUDE BIBLIOGRAPHIQUE
James LEDOUX
Juillet 1992, 76 pages.

PI 668 TWO COMPLEMENTARY NOTES ON SKEWED-ASSOCIATIVE CACHES
André SEZNEC
Juillet 1992, 10 pages.

PI 669 PARALLELISATION D'UN ALGORITHME DE DETECTION DE MOUVEMENT SUR UNE ARCHITECTURE MIMD
Fabrice HEITZ, Sergui JUFRESA, Etienne MEMIN, Thierry PRIOL
Juillet 1992, 34 pages.

PI 670 UN RESEAU SYSTOLIQUE INTEGRE POUR LA CORRECTION DE FAUTES DE FRAPPE
Dominique LAVENIER
Juillet 1992, 120 pages.

PI 671 EARLY WARNING OF SLIGHT CHANGES IN SYSTEMS AND PLANTS WITH APPLICATION TO CONDITION BASED MAINTENANCE
Qinghua ZHANG, Michèle BASSEVILLE, Albert BENVENISTE
Juillet 1992, 32 pages.

PI 672 ORDRES REPRESENTABLES PAR DES TRANSLATIONS DE SEGMENTS DANS LE PLAN
Vincent BOUCHITTE, Roland JEGOU, Jean-Xavier RAMPON
Juillet 1992, 8 pages.

PI 673 AN EXCEPTION HANDLING MECHANISM FOR PARALLEL OBJECT-ORIENTED PROGRAMMING
Valérie ISSARNY
Août 1992, 36 pages.

ISSN 0249 - 6399