Exception Handling and Asynchronous Active Objects: Issues and Proposal

Christophe Dony[1], Christelle Urtado[2], Sylvain Vauttier[2]

[1] LIRMM - CNRS and Montpellier II University - 161 rue Ada
34 392 Montpellier - France
dony@lirmm.fr

[2] LGI2P - Ecole des Mines d'Alès - Parc scientifique G. Besse - 30 035 Nîmes - France
{Christelle.Urtado, Sylvain.Vauttier}@site-eerie.ema.fr

Abstract Asynchronous Active Objects (AAOs), primarily exemplified by actors [1], nowadays exist in many forms (various kinds of actors, agents and components) and are more and more used because they fit well the dynamic and asynchronous nature of interactions in many distributed systems. They raise various new issues regarding exception handling for which few operational solutions exist. More precisely, a need exists for a generic, simple and expressive, programmer level, exception handling system that appropriately handles the following main exception handling issues or requirements in the context of AAOs: encapsulation, object autonomy, coordination of concurrent collaborative entities [2], "caller contextualization" [3], asynchronous signaling and handler execution, resolution of concurrent exceptions [4,5], exception criticality [6] and object reactivity.

This paper presents the specification of an evolution of the SAGE exception handling system [7], which provides solutions to those issues in the context of systems developed with active objects using one way asynchronous communications and interacting via the request / response protocol. Such a context, in which synchronizations constraints are, when needed, handled at the application level, allows for a very generic view of what could be done regarding exception handling in all systems that use active objects. The SAGE solution is original and provides a good compromise between expressive-power and simplicity.

Keywords: active objects, agents, distributed components, message driven components, exception handling, reliability, asynchronous message-based communication.

1 Introduction

Active objects are "*objects having their own computing resources i.e. their own private activity*" [8], or, said differently, objects "*decoupling method execution from method invocation*" [9]. *Asynchronous Active Objects* (AAOs) come in many

forms (actors, agents or components), with various interaction schemes (request / response or publish / subscribe [10]) and various forms of asynchronous communication (one-way or two-ways). They are more and more used as, for example, in multi-agents systems [11], in some distributed components architectures such as J2EE's with *Message Driven Beans*, in programming languages dedicated to grid applications (e.g. [12]) or to wireless devices on top of mobile networks as in [13]. AAOs, particularly in these new contexts, raise numerous issues regarding exception handling that have only been partially studied.

While the masterpieces of a generally accepted solution for exception handling in sequential programs are known, this is not yet the case for concurrent systems [14], even if some agreements exist. When systems with asynchronous communications are concerned, research works are still much more scattered. Initial actor languages included basic proposals to cope with exceptions [15] in which handlers were some dedicated actors, ancestors of today's exception supervisors, that had the same lacks, regarding handler contextualization (see Sect. 3.3), as Smalltalk or Ada initial lexical-scope handlers. Asynchrony has more recently motivated many research works in various contexts [16,17,18,19,20,21,22,23]but they only partially address AAO needs. Actually, agent systems are the AAO context in which exception handling proposals are the most achieved. However the supervisor model described in [24,25] does not properly handle the contextualization issue. Guardian [26,27] is a general and powerful solution which nonetheless proves to be complex to master and use. As explained in [26], *"Often exception handling in a program is the most complex [...] part of the system [...] and has to be either simplified or taken out of the hand of the average programmer"* and a solution for this is to *"separate global level exception handling from the application agents"*.

We have imagined an alternative solution consisting in analyzing and designing a language-level exception handling system dedicated to AAOs that:

- integrates what we consider to be the major research results from studies in sequential, concurrent or asynchronous contexts, and is expressive enough to address standard exception handling situations,
- reflects and takes into account the way AAOs and their execution are structured[3],
- is simple enough to be universally used by standard programmers.

The key requirements of the system are: to enforce encapsulation, to provide a representation for collaborative concurrent activities [14] so that they can be coordinated and controlled [2], to achieve caller contextualization [28,3] for handler definition and execution, to handle concurrent exceptions with resolution functions [4,29], to support asynchronous signaling and handler search and thus maintain object reactivity and to cope with broadcast messages, widely used in the request / response protocol.

[3] We have considered active objects in their less constrained form i.e. as autonomous entities that provide inter and intra-object concurrency, interact via a request / response protocol and use one-way asynchronous communications.

The paper is organized in four sections. Section 2 recalls some basic vocabulary and introduces an example. Section 3 presents the rationale of our main conceptual choices. Section 4 describes the system specification. It focuses on the description of the asynchronous handler search policy and explains its utility. Section 5 compares our proposal with related works.

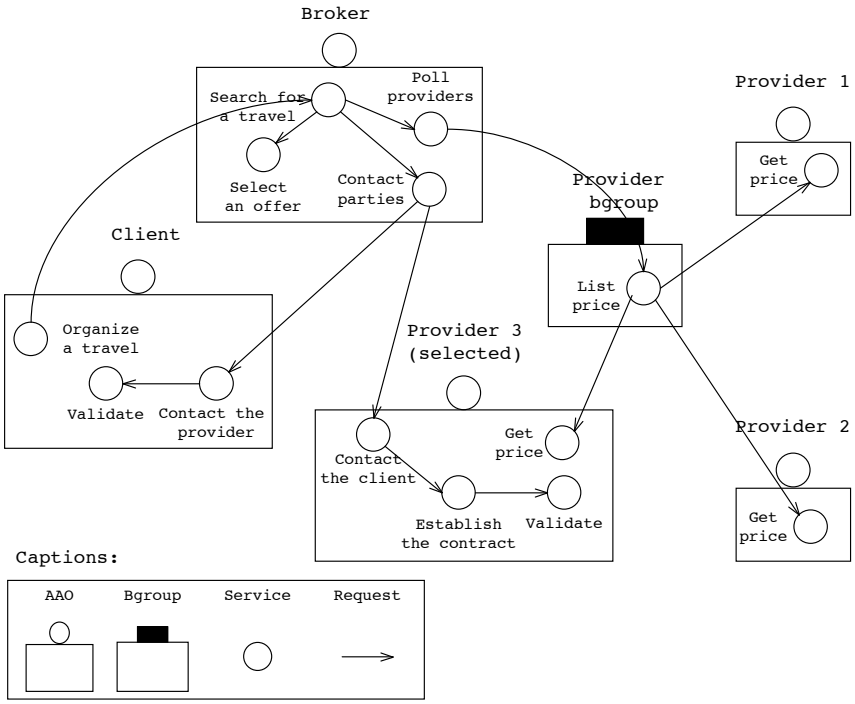

Figure 1. Execution resulting from a request to a travel agency

2 Definition, Terminology and Example

All objects in our discussion will be AAOs. The following object characteristics define the context of this study. Objects communicate by exchanging messages that carry information [10]. Messages are queued in the object's message box. Each object owns a thread dedicated to managing its message-box: its scans and interprets the received messages to trigger corresponding actions. The program unit executed when a recipient accepts a request carried in such a message is called a **service**. Objects can own several services that can be executed concurrently in dedicated threads (intra-object concurrency). The request / response interaction protocol generally comes along with a contract-based approach of software development, which states that whenever an object accepts a request,

it must provide a response, either standard or exceptional. Our objects use one-way communications[4] which means that responses to requests are not carried back away in the same communication channel that has carried the request, but by sending new separated messages back to callers [10]. Objects are autonomous: they can independently decide to start any activity or to handle any received message in whatever order. A collaborative activity is an activity that involves several objects or several services of an object in achieving a common goal.

As an illustration, we use the canonical *Travel Agency* example in which a *Client* can send a *Broker* a reservation message to request a bid for a travel. The contacted broker then sends a bid request to several travel providers and waits for their responses. Then, the *Broker* selects the best offer and requests the *Client* and the selected *Provider* to establish a contract (cf. Fig. 1). Our code examples use a Java-like syntax. Figure 2 thus shows examples of service definitions: lines 11–22 define the *Poll providers* service and lines 24–41 the *Contact Parties* one. We call **complex services**, services the code of which contain other messages and *atomic services* the others. In the example, *Get price*, that returns a *Provider*'s bid, is an atomic service (cf. Fig. 1) and *Organize a travel*, which handles a *Client*'s initial request, is a complex one. Broadcast message that contain collective requests, are frequently used in AAO applications and are generally delivered via entities that represent groups of objects as, for example, *roles* in MadKIT or *topics* in J2EE MDBs. We take this kind of requests into account and use entities that we call **bgroup**s (for *broadcast groups*) to denote such groups of objects. A *bgroup* possesses an (implicit) complex service that broadcasts the requests it receives to all objects in its collection.

This simple example brings to the fore many pertinent issues: how to control and interpret an exception asynchronously raised by one of the travel providers? Where is the best place to interpret it? Should all providers be notified when one of them fails? Should the broker be able to cancel all requests to travel providers for a given reason? Where and how to associate a handler for the collaborative activity that consists in requesting several travel providers concurrently? When should it be invoked and in which context?

3 Rationale for the SaGE Exception Handling System

Each of the following sub-sections discusses the rationale of some of our choices.

3.1 Coordination of Concurrent Activities

As shown in our previous works [7,30], efficiently handling exceptions in concurrent systems using asynchronous communications requires *cooperative concurrency* to be supported, as for other concurrent systems [2,14].This amounts to provide a representation of collective activities and a way to define handlers

[4] Two-ways asynchronous communications generally use *future* objects [16] which are more restrictive because the order in which they are read imposes synchronization constraints.

```
( 1)  public class Broker implements AsyncActiveObject
( 2)    {
( 3)    ...
( 4)
( 5)    public void handle (GlobalNetworkException exc)
( 6)    // handler associated to the Broker asynchronous active object
( 7)      { ... }
(10)
(11)    class PollProviders implements Service
(12)      {
(13)      ...
(14)      public void body ()
(15)        { ... }
(16)      public void handle (BadParameterException exc)
(17)      // handler associated to the PollProviders service
(18)        { signal (new NoAirportInDestinationException (...); }
(19)      public void handle (NoProviderException exc)
(20)      // handler associated to the PollProviders service
(21)        { ... }
(22)      }
(23)
(24)      class ContactParties implements Service
(25)         {
(26)         public void body ()
(27)           {
(28)           ...
(29)           sendMessage (new RequestMessage (aClient,
(30)                           "ContactSelectedProvider")
(31)                           {
(32)                           public void handle (OffLineException exc)
(33)                           // handler associated to a request
(34)                             {
(35)                             wait(120);
(36)                             retry();
(37)                             }
(38)                           });
(39)           ...
(40)           }
(41)        }
(42)   ...
(43)   }
```

Figure 2. Service and handler definitions in SAGE

at all places in a program where several concurrent participants are working together to the achievement of a global task. In the example, such a handler should be defined somewhere at the level of the requesting *Broker* object it should be invoked whenever one or more travel providers signal an exception, and it should be able to access the *Broker* context.

3.2 Encapsulation

A well-known consequence of the introduction of exception handling primitives in a language is that it gives programmers constructs to break encapsulation [31]. If it seems unavoidable to pass arguments from signalers to handlers, it is possible to act on another concern with encapsulation which occurs each time handlers are executed in a context where the data they need is not accessible. This can globally be the case in object languages with all kinds of supervisor-based models for exception handling [15,32,24] or, more marginally, in procedural languages with handlers associated to shared data, as initially suggested by [33].

Supervisors are objects dedicated to exception handling, which can be considered themselves as handlers or to which handlers are attached. The actor proposal for exception handling [15] is based on that idea. The issue with this approach is that supervisors are not encapsulated within the objects that experience the failure and therefore cannot access their internal state without breaking encapsulation. Our solution to prevent this, experimented in [3,7] is to define and encapsulate handlers within the object or activity they control.

3.3 Contextualization

"Contextualization" refers to two connected issues: the scope of handlers and the context in which they are executed. The scope of a handler determines the way and the order in which they are searched for. It directly impacts the signaling algorithm. The way handlers are defined and executed determines their context. Two main approaches can globally be distinguished. In the static approach, handlers have a lexical scope and are executed in an environment that lexically contains the signaling one. Its main advantages are its simplicity and the fact that it requires no additional language construct. Conversely, its main drawback is that it fails to achieve fault tolerant encapsulations [3]. In the dynamic approach, handlers have a dynamic scope: the portion of the program they control is execution dependent .

"Caller contextualization" is a variant of the latter approach in which handlers have a dynamic scope and are executed in the lexical context of the caller of the faulty routine where they are defined. A simple example of the interest of caller contextualization is the *DivideByZero* exception. It is easy to verify that, whatever the reason *DivideByZero* has been raised, only the caller of the *divide* operation can give a semantically founded interpretation of the reason why the divisor equals zero and can take an appropriate decision, in its context. This policy has been globally accepted as the best one for achieving fault-tolerant encapsulations in contract-based request / response interactions in all sequential

languages (from PL/I, Clu, Mesa, Lisp, Flavors, Clos, C++, ANSI Smalltalk, to Java).

As far as AAOs are concerned, a choice has to be made among various alternatives. Original actor languages proposed dynamic scope handlers. It was proposed in [15] to associate an exceptional continuation actor to each message sending. However, such an actor is unable to access the calling context and therefore to give context-dependent answers to exceptions. The exception handling systems based on supervisors [24] and those that do not propagate exceptions outside of the thread in which they are signaled (as J2EE MDBs) suffer from the same lack. Some languages propose both static and dynamic scope handlers to respectively achieve fault tolerance and exception handling. It is the case of Beta [34] and Smalltalk in its original blue book version[5]. They propose two kinds of exceptions and two means to signal them. The issue for a programmer with such a system is to know which kind of exceptions to signal.

In fact, caller contextualization is equally well adapted to both sequential and concurrent contexts. It has been made available to AAO systems by recent research proposals [7,27]. It has to be noted that applying it to its whole extent excludes solutions in which an exception in a participant of a collaborative task is signaled to its brother participants. In our example, this means that an exception in a single *Travel provider* would be signaled to all the other providers working on the same request. Although of effective potential interest [35], we reject this solution because of its intrinsic complexity for programmers. In our example, it could lead to very complex and intricate situations as soon as several travel providers signal exceptions concurrently.

3.4 Resolution, Criticality

Entities that represent a set of collaborating objects are a natural place where to enable programmers to specify policies to deal with the multiple exceptions they may concurrently signal. Some resolution mechanism to "concert" or "resolve" such exceptions have been proposed in [4,17,36]. A resolution function is a user defined function that can be attached to entities that represent collaborative activities (complex services or bgroups). It is invoked to concert the set of exceptions that have been signaled to the entity in which it is defined. It receives the exception object as an argument. Its role is to analyze the situation, to block and monitor under-critical exceptions [6] or to let pass through critical (concerted) ones. A concerted exception globally reflects the incorrect behavior of the collective activity. In the example, when a *Provider-Bgroup* sends n requests to n providers, a resolution function attached to the bgroup can determine that the failure of one of them is not critical for the collective activity and simply has to be monitored and that the failure of 90 percent of them should entail the signaling of a concerted one. In an asynchronous communication world, we

[5] In the original Smalltalk, lexical scope handlers are standard methods and dynamic scope handlers are lexical closures passed as arguments (e.g.: `aCollection find: anObject ifAbsent: [...]`).

propose to improve these ideas by calling the resolution function (1) as soon as an exception is signaled in a thread of a collective activity and (2) each time an exception is signaled, without waiting for the termination of all the services that constitute the collective activity. The signaling algorithm will be responsible for achieving these requirements.

4 Specification of SaGE

Our specification classically comes in four steps indicating: to which program units to attach exception handlers, how to signal exceptions, what can be written within handlers to put the system back into a coherent state and in which order handlers are searched for.

4.1 Data Structures for Coordination and Contextualization

Coordination and contextualization require that some dedicated internal data structures be defined. Caller contextualization first requires that a doubly-linked tree of service execution contexts be monitored. In such a tree, a node represents a **complex service** execution context and a leaf the one of an **atomic service**. Callee to caller links are used to look for handlers. Caller to callee are used, for example, to kill the sub-services of a terminating complex service. Figure 1 shows the execution context tree that results from the services executed in the travel agency example. Complex services execution contexts are also used to collect and monitor the results of the execution of their sub-services, either they be standard results or exceptions.

4.2 Defining Handlers

The standard Fipa request / response interaction pattern is divided in four main steps:

- **Request and acknowledgment**: a *sender object* sends a *request* to a *receiver object*[6], which can be an individual object or a *bgroup*.
- **Acceptation**: the receiver indicates whether he accepts the request or not. Acceptation is a commitment to provide a response, either normal or exceptional.
- **Execution**: the receiver executes a service.
- **Response**: the service execution is finished and the receiver either sends back a normal response or signals an exception.

[6] The complete protocol includes an acknowledgment step to check that the message has not been lost. For the sake of simplicity, we will always consider here that sent requests arrive to their destination and that it is the transport layer (middleware) responsibility to guarantee this.

These steps highlight the role of four key entities in this interaction pattern: the request, the service, the active object, the *bgroup*. They are the four program units to which exception handlers could be attached:

- Handlers attached to **requests** allow, for example, to specify two different reactions to the occurrences of two exceptions raised by two invocations of the same service. Figure 2 (lines 29–38) shows how a handler can be attached to a specific request.
- Handlers attached to **services** allow to treat exceptions that are raised, directly or indirectly, by their execution. If the service is complex, the handler has to be able to deal with concurrent exceptions, to compose with partial results or to ignore partial failures. Figure 2 (lines 16–22) shows an example in which two handlers are attached to a service.
- Handlers attached to **bgroups** amount to attach them to their implicit service (see Sect. 2).
- Finally, handlers attached to **objects** (see Fig. 2, lines 5–7) are those handlers common to all services, designed, for example, to maintain in an uniform way the coherence of the object private data.

These capabilities are powerful enough to encompass most cases and simple enough to be easy to learn and use. Other systems are either more complex or less expressive but the comparison requires that the signaling algorithm be presented. All handlers have a dynamic scope. Resolution functions will be considered later.

4.3 Signaling

Signaling is done by the means of a classical *signal* primitive (cf. Fig. 3). Signaling is possible anywhere in the code and of course within handlers.

```
signal(new SaGEException("select\_error",getownerQueue()));
```

Figure 3. The *signal* primitive

4.4 Defining Handlers.

Exception handlers are classically [37] defined by the set of exception types they should catch and by their body (as illustrated by Fig. 2, lines 32–37, for example):

- A handler can simply restore whatever should be, to put back data into a coherent state, and can **return** a value that becomes the value of the expression the handler is associated to. In case of a message sending expression (standard or broadcast), the value returned by the handler is the value of the expression. In case of a handler attached to a service, the value becomes the value of the service execution. In case of a handler attached to an object, the value becomes the value of the service execution that raised the exception.

– A handler can also classically **signal** a new exception (generally of a higher conceptual level) or **re-signal** the original one. This behavior is illustrated on Fig. 2 line 18. Of course, handlers cannot protect themselves against the exceptions they signal.
– A handler can finally **retry** the execution of the program unit it is attached to. To retry [38,39] amounts to entirely re-execute the program unit it is attached to, generally after having modified the local environment, but in the same historical context. This possibility is illustrated on Fig. 2, lines 32–37. In case of handlers attached to objects, retrying means re-executing the service that signaled the exception.

4.5 Handler Search.

Let S_n be the service in which an exception E is raised i.e. that contains the signaling point (the "*call-site*"). When E is raised, the execution of S_n is suspended (cf. Algo. 1) and the handler search is done using the thread of S_n. If S_n is complex, it continues to monitor responses and other exceptions coming from its sub-services, the execution of which is not interrupted yet. If a concurrent sub-service signals another exception E_2 during handler search for E, it will be either ignored or considered later if no handler is executed for E. It may happen that no handler be executed for E when a resolution function considers that E is not critical.

Algorithm 1 The signal primitive

```
Require: Exception exc // raised exception object
  Service sce ← service in which the signal primitive is called
  if sce's state is "suspended" // exc is signaled during a handler execution then

    if the handler that is being executed is attached to a request then
      execute LocalSearch (exc, sce)
    else
      call CallerSearch (exc, calling-service)
      terminate sce
    end if
  else
    // exc is signaled from outside a handler
    sce 's state ← "suspended"
    execute LocalSearch (exc, sce)
  end if
```

Then, a handler for E is searched for locally:

– first, in the list of handlers associated to S_n ,
– then, in the list of handlers associated to the owner object of S_n.

If a suitable handler H is found there, it is executed and its execution terminates the execution of S_n. Along with the execution of H all pending sub-services of S_n, if any, are terminated. The caller service of S_n (and all of its other sub-services) remain unaffected and normally pursue their execution concurrently with H's.

Algorithm 2 Handler Search - LocalSearch (Exception exc, Service sce)

```
Require: Exception exc of type T, Service sce // raised exception object and current
  service
  if a handler H_T exists attached to sce then
    execute H_T
  else
    if a handler H_T exists attached to the AAO to which sce belongs then
      execute H_T
    else
      if a calling service exists then
        call CallerSearch (exc, calling-service)
      else
        execute default handler // top-level has been reached
      end if
    end if
  end if
  terminate sce // terminates the service and (if it is a complex one) recursively all its
  sub-services
```

If no handler is found locally, the search process proceeds in the calling context (service S_{n-1}), in order to guarantee caller contextualization (cf. Sect. 3.3). First S_{n-1} is suspended and the search for a handler initiated. The search in S_{n-1} is done concurrently with the termination of S_n. This original capability guarantees that all activities that have become useless because of a failure are terminated as soon as possible. This preserves system resource. The process carries on as follows:

- first, it searches the list of handlers associated to the request which initiated S_n. There, the resolution function associated to S_{n-1} is executed. If it lets the exception pass through, the search process continues. If not, the search process stops and no handler will be executed (cf. Sect. 4.6).
- then, it searches the list of handlers associated to S_{n-1}
- finally, it searches the list of handlers associated to the owner of S_{n-1}.

If no handler is found, the same three steps are repeated once again into the caller's caller context (S_{n-2}). This process iterates until either an adequate handler is found and executed or the root of the service tree is reached. In the latter case, a default top-level handler is executed.

Algorithm 1 describes the signaling of an exception. Algorithms 2 and 3 describe the local and the caller's context part of the handler search process. To

Algorithm 3 Handler Search - CallerSearch (Exception exc, Service sce)

```
Require: Exception exc of type T, Service sce // raised exception object and current
  service
  if a handler H_T exists attached to the request sent by sce then
    execute H_T
  else
    log exc into sce's exception log
    execute sce's resolution function[7]
    if the resolution function returns a concerted exception then
      sce 's state ← "suspended"
      execute LocalSearch (exc, sce)
    end if
  end if
```

ease the writing of these algorithms, let us note H_T an exception handler defined to treat exceptions of type T. Let us also define two primitives to call sub-procedures: *execute*, to denote a sequential call and *call*, to denote an asynchronous concurrent call. When a procedure is called through the *execute* primitive, it executes in the same thread as its caller. When a procedure is called through the *call* primitive, it executes in a thread different from its caller's. The remaining instructions in its caller's context are then executed concurrently with the called procedure.

4.6 Concerted Exception Support

SaGE provides an exception resolution support (cf. Sect. 3.4) that is integrated to the handler search. It enables resolution functions to be defined at places where concurrent activities are launched and have to be co-ordinated i.e. at the complex service level. There is no need for a resolution function either at the request level, because requests are atomic, or at the AAO level because all semantically sound activities of objects, that need to be co-ordinated, are accessible via services. *Bgroups* own resolution functions that are attached to the implicit complex service they execute. The *bgroup* acts first as a request broadcaster and then as a response collector in order to send back a single (composite) response to the client AAO. The default behavior of the resolution function associated to a bgroup service is, once all recipients have replied, to aggregate all the exceptions that occured into a concerted one. Of course, a programmer can define his own exception resolution function as in the example of Fig. 4.

In our model, a resolution function is executed each time an exception handler is searched for in the caller's context (cf. Algorithm 3). Whatever is done in the function, three cases are finally possible:

[7] This is also valid in the case of a Bgroup because the resolution function is *in fine* attached to the (default) broadcasting service.

```
public SaGEException concert (Vector subServicesInfo)
{
   int failed = 0;

   // count the number of exceptions raised in subservices and
   // the number of subservices that are still running
   for (int i=0; j<subServicesInfo.size(); i++)
      {
      if ((ServiceInfo) (subServicesInfo.elementAt(i)).getRaisedException()
         != null)
         failed++;
      }

   // if more than 30% failed, there are too many bad providers
   if (failed > (0.3*subServicesInfo.size()))
      return new SaGEException("too_many_bad_providers",getAddress());

   // computing still running - no critical situation
   return null;
}
```

Figure 4. Java-like code of an exception resolution function associated to the Provider Bgroup

- the exception is critical for the service. The resolution function returns the exception object and the handler search process carries on.
- the resolution function evaluates that the exception is under-critical and that nothing more should be done yet. The exception is logged, the resolution function returns null and the handler search process stops. The collective activity is not affected. The only service that is terminated is the defective sub-service.
- the resolution function evaluates that the exception is under-critical but that there is a need to signal something, for example because too many under-critical exceptions have been logged. The resolution function returns a special exception that reflects the situation and the handler search carries on.

Such a use of resolution for concerted exception differs from the original one [40,29] in that it is adapted to a context in which there are no synchronization points. A mechanism to calculate the time when the resolution function should be executed has been proposed in [6]. Our solution consists in tightly integrating the execution of the resolution function to our handler search mechanism. Our resolution function is executed each time the handler search process goes back from a context to its caller. At each step, it can stop the process or let it continue (with either the original exception or a new, concerted one). This characteristic makes our system more reactive, because our resolution function evaluates the situation each time an exception is signaled.

5 Related Works

Concurrent programming systems fall into three main categories when exception handling is considered. These categories correspond to the kind of concurrency that is supported [14]. This directly determines how AAOs can interact, and, as part of their interactions, how exceptions can be signaled between them.

5.1 Isolated Concurrency

Isolated concurrency is provided by standard programming languages such as Java. Its goal is to allow several AAOs (threads) to execute concurrently in a shared context (the address space of a virtual machine) as if each of them was the only existing AAO. To achieve this, the system enforces that the activity of an AAO does not interfere with another one. For example, locks are managed on shared resources in order to transparently serialize concurrent accesses to them. In the same way, no standard means is provided to send information from a thread to another. When an exception is raised in an AAO, it is signaled along its own execution stack (in its separated execution thread). When the exception is not caught and reaches the top level of the execution stack, the AAO is destroyed by the system (the thread is discarded by the thread manager). The other AAOs are not warned of the failure in order to maintain their isolation.

5.2 Cooperative Concurrency and Exception Handling

Isolated concurrency is only suitable when strictly parallel computations are to be managed,for example when handling requests from different clients. But when a set of entities are intended to participate together to the achievement of a global activity, means to handle their cooperation are then required. More specifically, in such forms of cooperative concurrency, there is a crucial need to manage how the individual failure of an AAO impacts the global activity and, as a consequence, should impact the activity of other entities.

Monitors. A first technique is to provide specific entities which role is to monitor other entities and to implement how errors are to be handled when the global context of the monitored entities is considered. Java proposes that a thread can belong to a *ThreadGroup*. When an exception is raised and uncaught in a thread T , it is then signaled to the thread group to which T belongs. A unique handler, that catches all the signaled exceptions, can be associated with the thread group. This allows basic actions to be carried out, such as to kill threads that are still running in the thread group, in order to terminate the whole activity of the group. Some SMAs provide such a mechanism in the form of supervisor agents. Supervisors are agents that monitors other agents in the system and to which exceptions are signaled.They are used, for example, to react to the death of an agent (killed by an uncaught exception) and warn other agents that it cannot be reached any more. In Erlang [22], supervisor processes can be tied to other

ones to be informed of their termination. In Oz [20], asynchronous exception related to distribution are handled thanks to dedicated monitors. Monitors enable a good separation of concerns, because they keep behaviors dealing with errors well separated from behaviors dealing with normal activities. However, they raise encapsulation and contextualization issues. When used for AAOs, monitors can only perform external, platform-level, generic actions such as to suspend, restart or destroy an AAO. Monitors are finally somehow restricted to the handling of generic exceptions because they have no access (unless breaking encapsulation) to objects' internal state and, generally, to any contextual information about the cause of the exception. This drastically limits the applicability of monitors when specific errors, regarding the specific coordinated activity of a set of entities, are to be handled.

CA Actions. A common solution, in systems that tackle this contextualization issue, is that collective activities of coordinated concurrent entities must become explicit, in order to structure the global execution contexts and provide a support to handle exceptions. A CA Action [14] allows the representation of a collective activity to which different entities, called participants, contribute concurrently. Different variants of this concept, along with different EHSs, have been proposed. In [41], a CA Action is defined as a sort of contract that ties together all the participant entities. As a part of the contract, these participants must provide support for the handling of a common set of exceptions. When an exception is raised by one of the participants, it is signaled to the others. This way, all the participants to a CA Action suspend their individual activity (and thus suspend the execution of the whole global activity). The system then enforces a synchronization point between all the participants. When multiple, concurrent exceptions are signaled, this policy ensures that a same set of exceptions is finally signaled to each participant. Each participant resolves this exception set thanks to an exception resolution tree provided by the CA Action (and thus common to all the participants): this entails that each participant finally handles the same resolved exception. However, the handlers that are eventually triggered are specific to each participant and are cohesive parts of their behavior. This model addresses the contextualization issue stressed above. Exceptions pertaining to a global activity are handled by having its participants contribute collectively to their treatment, as a result of the coordinated execution of their own handlers. One concern with such a model is the cost of the coordination between the participants. Indeed, it implies the exchange of numerous messages in order to inform the other participants of exceptions and of execution suspensions. Moreover, it entails a strong coupling between the participants as it requires a common set of exception types and a common exception resolution tree to be used. This model is therefore not perfectly suited for highly distributed and open systems.

Guardians. Among other things, various improvements have been introduced to the above model in [26,42,27]. A CA Action is monitored by a special par-

ticipant, called a "guardian". Participants signal to the guardian exceptions that are global to the CA Action. The guardian then suspends the execution of all the participants, while collecting concurrent exceptions. The set of exceptions collected by the guardian is then resolved thanks to a first set of rules that determines what unique global exception is to be handled. Next, a second set of rules is used to transform the global exception into the specific exceptions that will finally be signaled to each participant. This way, only one participant, the guardian, needs to track the exceptions and the status of other participants. Moreover, the cooperation of the participants, when handling global exceptions, is defined by the set of rules of the guardian. Rules are tailored to adapt to the specific behavior of each participant, so that no predefined requirement is to be imposed to the participants. Providing a complex and powerful solution, Guardian is especially relevant to deal with exceptions related to shared environments where all participants can effectively cooperate to restore a consistent state. This kind of exceptions encompasses system-level exceptions that warn of the faulty state of some shared resource (disk, memory, network, ...). SAGE provides a simpler solution when handling exceptions related to collaborating pairs of objects such as clients and servers.

5.3 Collaborative Concurrency and Exception Handling

Models discussed in the previous section indeed share the idea that when an exception occurs in the context of a collective activity, handlers are sought and executed in all its participants. Besides, in situations in which couples of entities collaborate together, for example when a server informs its client that it has failed to achieve some requested service, signaled exceptions are to be handled in the context of the caller. Exceptions are therefore much more efficiently handled as responses sent by the server than as broadcasted information. To deal with such responses, many systems for asynchronous programming use "future" objects [16,43,20]. Futures are response holders that are immediately returned to client entities when they asynchronously request a service to a server AAO. When a client needs to use the value of a response, it tries to access the value of the corresponding future. If no value is yet bound, the client AAO can perform a blocking wait. When an exception is bound to a future instead of a standard value, it is signaled to a client when the future is read. The client then usually handles it with some classical built-in *try-catch* like constructs. The main advantages of such a solution are its simplicity and its ability to be seamlessly integrated to existing programming languages. Its drawback is that it does not cope with complex situations. For example, when requests are sent concurrently to different servers, it is difficult to foresee the best order in which futures should be read in order not to wait for an unbound response while others are yet available and could be treated. This is one of the reasons why we think that reactive AAO models are more interesting for exception handling. With futures, exceptions cannot be treated as soon as possible and can sometimes be simply lost when some futures are not read. In a reactive system, like the one in which we have specified our system, exceptions are signaled asynchronously by sending

messages and can therefore be treated as soon as they occur. The implementation of SaGE in a future-based context has not been done yet but the resulting system would be more limited than today's one.

Erlang [22] has a sophisticated EHS to deal with exceptions within concurrent processes and also proposes an asynchronous message sending based solution to signal process termination exceptions from one process to another. In Erlang, messages that contain exceptions cannot be distinguished from others and, as a consequence, the handling of asynchronous exceptions can only be *ad hoc*. On the contrary, SaGE carry exceptions with messages that, when received, trigger a full-fledged EHS. Finally, to cope with concurrent exceptions, [44] also suggests the introduction of future groups in order to gather the exception of a set of futures and apply a resolution function to them. But this solution requires the writing of a lot of code to explicitly deal with future groups. With SaGE, the support for exception resolution is directly integrated in the EHS. Provided that corresponding resolution functions are defined, concurrent exception management does not require any extra programming.

6 Conclusion and Future Work

In this paper, we have proposed a specification of an exception handling system adapted to asynchronous active objects. We have especially focused on service-oriented systems and on the request / response interaction scheme. Our system aims at combining simplicity, usability at the language level by standard programmers, integration and adaptation of known key-solutions for sequential and concurrent exception handling and full integration of active objects. Our solution conforms to all the key requirements identified in Sect. 1: encapsulation and reactivity enforcement, ability to write context-dependent handlers, ability to coordinate and control group of active objects collaborating to a common task, ability to configure the exception propagation policy by defining *exception resolution functions*, ability to immediately handle exceptions that are critical or to only log under-critical ones until their conjunction enables a diagnosis to be established. We propose dynamic scope handlers associated to requests, services and objects. Resolution functions can be defined at the service level, which is the place where collaborative tasks can be coordinated. They come together with a signaling primitive, a handler search algorithm and a handler invocation mechanism that take into account the execution history and, when possible, work asynchronously to improve object reactivity. So this model is especially suited for applications that need few synchronization and a high level of concurrency and reactivity.

We implemented and successfully experimented this model both with MadKIT, to handle exceptions in multi-agents systems, and with the open-source JOnAS J2EE implementation, to provide a fault-tolerant support to the execution of asynchronous message driven beans (MDBs). We think the set of design choices that make SaGE a good compromise between expressive-power and simplicity can be adapted to various kinds of active objects, to various forms of asynchronous

communications (e.g. future-based) and to different interaction protocols (e.g. publish/subscribe). We also think it is general enough to be used as a base level for the implementation of systems offering higher-level control structures for fault tolerance, such as conversations [45] or transactional systems [46,47].These all are future works objectives as is the introduction of the *resumption* model of exception handling in our system. Indeed, we do think that the *restart* construct and protocol introduced in the Flavor system [37,48] are of primary importance in a dynamic world of interacting objects.

Acknowledgments. The authors wish to thank Jacques Ferber for many fruitful discussions on SMAs and for having given Madkit to the community as an experimentation tool, Anand Tripathi for his helpful comments on the preliminary version of this paper, and Frédéric Souchon who has implemented SaGE both in MadKIT and JONAS.

References

1. Carl Hewitt, Peter Bishop, R.S.: A universal modular actor formalism for artificial intelligence. In: Proceedings of the International Joint Conference on Artificial Intelligence. (1973) 235–246
2. Randell, B., Romanovsky, A., Rubira-Calsavara, C., Stroud, R., Wu, Z., Xu, J.: From recovery blocks to concurrent atomic actions. In: Predictably Dependable Computing Systems. ESPRIT Basic Research Series (1995) 87–101
3. Dony, C.: Exception handling and object-oriented programming : towards a synthesis. ACM SIGPLAN Notices **25**(10) (1990) 322–330 *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).
4. 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, New Orleans, Louisianna, USA (1991) 92–100
5. Romanovsky, A.: Practical exception handling and resolution in concurrent programs. Computer Languages **23**(1) (1997) 43–58
6. Lacourte, S.: Exceptions in Guide, an object-oriented language for distributed applications. In Springer-Verlag, ed.: Proceedings of ECOOP 91. Number 5-90 in LNCS, Grenoble (France) (1990) 268–287
7. Souchon, F., Dony, C., Urtado, C., Vauttier, S.: Improving exception handling in multi-agent systems. In de Lucena, C.J.P., Garcia, A.F., Romanovsky, A.B., Castro, J., Alencar, P.S.C., eds.: Software engineering for multi-agent systems II, Research issues and practical applications. Volume 2940 of Lecture Notes in Computer Science. Springer (2004) 167–188
8. Briot, J.P., Guerraoui, R.: A classification of various approaches for object-based parallel and distributed programming. In Padget, J.A., ed.: Collaboration between Human and Artificial Societies. Number 1624 in Lecture Notes in Artificial Intelligence. Springer-Verlag (1999) 3–29 Invited conference.
9. Lavender, R.G., Schmidt, D.C.: Active object: An object behavioral pattern for concurrent programming. In Coplien, Vlissides, Kerth, eds.: Pattern Languages of Program Design. Addison-Wesley Reading (1996)
10. FIPA: Foundation For Intelligent Physical Agents : Request Interaction Protocol Specification. (2002)

11. Ferber, J.: Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence. Addison-Wesley Pub Co; 1st edition (February 25, 1999) (2005)
12. Clement Jonquet, S.A.C.: The strobe model: Dynamic service generation on the grid. Applied Artificial Intelligence Journal Special issue on Learning Grid Services **19**(9-10) (2005) 967–1013
13. Dedecker, J., Cutsem, T.V., Mostinckx, S., D'Hondt, T., Meuter, W.D.: Ambient-oriented programming in ambienttalk. In: Proceedings ECOOP'06 (European Conference on Object-Oriented Programming), Springer-Verlag (2006) To appear.
14. Romanovsky, A.B., Kienzle, J.: Action-oriented exception handling in cooperative and competitive concurrent object-oriented systems. In: Advances in Exception Handling Techniques. (2000) 147–164
15. Theriault, D.: A primer for the Act-1 language. Technical Report AI Memo 672, MIT Artificial Intelligence Laboratory (1982)
16. Halstead, R., Loaiza, J.: Exception handling in multilisp. In: 1985 Int'l. Conf. on Parallel Processing. (1985) 822–830
17. Campbell, R., Randell, B.: Error recovery in asynchronous systems. IEEE Transactions on Software Engineering (SE) **SE-12 number 8**(8) (1986) 811–826
18. Gärtner, F.C.: Fundamentals of fault tolerant distributed computing in asynchronous environments. ACMCS **31**(1) (1999) 1–26
19. Keen, A.W., Olsson, R.A.: Exception handling during asynchronous method invocation. In Monien, B., Feldmann, R., eds.: Proceedings of Euro-Par 2002 Parallel Processing. Lecture Notes in Computer Science. Springer-Verlag (2002) 656–660
20. Van Roy, P.: On the separation of concerns in distributed programming: Application to distribution structure and fault tolerance in Mozart. In: Fourth International Workshop on Parallel and Distributed Computing for Symbolic and Irregular Applications (PDSIA 99), Tohoku University, Sendai, Japan, World Scientific (1999)
21. Campéas, A., Dony, C., Urtado, C., Vauttier, S.: Distributed exception handling : ideas, lessons and issues with recent exception handling systems. In: Proceedings of RISE'04 : First International Workshop on Rapid Integration of Software Engineering techniques, Luxembourg (2004) 82–92
22. Carlsson, R., Gustavsson, B., Nyblom, P.: Erlang: Exception handling revisited. In: Proceedings of the Third ACM SIGPLAN Erlang Workshop. (2004)
23. Iliasov, A., Romanovsky, A.: Exception handling in coordination-based mobile environments. In: Proceedings of 29th IEEE International Computer Software and Applications Conference (COMPSAC 2005), 25-28 July, Edinburgh, Scotland, UK. (2005) 341–350
24. Klein, M., Dellarocas, C.: Exception handling in agent systems. In Etzioni, O., Müller, J.P., Bradshaw, J.M., eds.: Proceedings of the Third Annual Conference on Autonomous Agents (AGENTS-99), New York, ACM Press (1999) 62–68
25. Klein, M., Dellarocas, C.: Towards a systematic repository of knowledge about managing multi-agent system exceptions, ases working paper ases-wp-2000-01 (2000)
26. Tripathi, A., Miller, R.: Exception handling in agent oriented systems. In: Advances in Exception Handling Techniques. LNCS (Lecture Notes in Computer Science) 2022. Springer-Verlag (2001) 128–146
27. Miller, R., Tripathi, A.: The guardian model and primitives for exception handling in distributed systems. IEEE Trans. Software Eng. **30**(12) (2004) 1008–1022
28. Dony, C.: An object-oriented exception handling system for an object-oriented language. In: Proceedings of ECOOP'88. (1988) 146–161

29. Issarny, V.: An exception handling mechanism for parallel object-oriented programming: Towards the design of reusable, and robust distributed software. Journal of Object-Oriented Programming 6(6) (1993) 29–39
30. Souchon, F., Urtado, C., Vauttier, S., Dony, C.: Exception handling in component-based systems : a first study. In Romanovsky, A., Dony, C., Knudsen, J., Tripathi, A., eds.: Technical Report TR 03-028. Proceedings of the Exception Handling in Object-Oriented Systems workshop at ECOOP 2003. Department of computer science, University of Minnesota, Minneapolis, Darmstadt, Germany (2003) 84–91
31. Miller, R., Tripathi, A.R.: Issues with exception handling in object-oriented systems. In: Proceedings of ECOOP'97. (1997) 85–103
32. Dellarocas, C.: Toward exception handling infrastructures for component-based software. In: Proceedings of the International Workshop on Component-based Software Engineering, 20th International Conference on Software Engineering (ICSE), Kyoto, Japan, April 25-26, 1998. (1998)
33. Levin, R.: Program structures for exceptinal condition handling. Phd dissertation, Dept. Comput. Sci., Carnegie-Mellon University Pittsburg (1977)
34. Knudsen, J.L.: Fault tolerance and exception handling in beta. In: Advances in Exception Handling Techniques. LNCS (Lecture Notes in Computer Science) 2022, Springer-Verlag (2001)
35. Burns, A., Randell, B., Romanovsky, A., Stroud, R., Wellings, A., Xu, J.: Temporal constraints and exception handling in object-oriented distributed systems. Design for Validation (DeVa) - Third Year Report, Esprit LTR Project 20072 - DeVa (1998)
36. Tartanoglu, F., Issarny, V., Levy, N., Romanovsky, A.: Dependability in the web service architecture (2002)
37. K.Pitman: Error/condition handling. Technical report, Contribution to WG16. Revision 18.Propositions for ISO-LISP. AFNOR, ISO/IEC JTC1/SC 22/WG 16N15 (April 1988)
38. Goodenough, J.B.: Exception handling: Issues and a proposed notation. Commun. ACM **18**(12) (1975) 683–696
39. Meyer, B.: Disciplined exceptions. Technical report tr-ei-22/ex, Interactive Software Engineering, Goleta, CA (1988)
40. Issarny, V.: Concurrent exception handling. In: Advances in Exception Handling Techniques. LNCS (Lecture Notes in Computer Science) 2022. Springer-Verlag (2001) 111–127
41. Randell, B., Romanovsky, A., Stroud, R.J., Xu, J., Zorzo, A.F.: Coordinated Atomic Actions: from Concept to Implementation. Technical Report 595, Department of Computing Science, University of Newcastle upon Tyne (1997)
42. Miller, R., Tripathi, A.: Primitives and mechanisms of the guardian model for exception handling in distributed systems. In: Exception Handling in Object Oriented Systems: towards Emerging Application Areas and New Programming Paradigms Workshop (at ECOOP'03 international conference) proceedings. (2003)
43. Caromel, D., Henrio, L., Serpette, B.: Asynchronous and deterministic objects. In: Proceedings of the 31st ACM Symposium on Principles of Programming Languages, ACM Press (2004) To appear.
44. Rintala, M.: Handling multiple concurrent exceptions in C++ using futures, kokoelmassa romanovsky. In: Developing Systems that Handle Exceptions, Proceedings of ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems, ACM Press (2005)
45. Romanovsky, A.B.: Conversations of objects. Computer Languages **21**(3/4) (1995) 147–163

46. Kienzle, J.: Open Multithreaded Transactions - A Transaction Model for Concurrent Object-Oriented Programming. Kluwer Academic Publishers (2003)
47. Guelfi, N., Razavi, R., Romanovsky, A., Vandenbergh, S.: Drip catalyst: An MDE/MDA method for fault-tolerant distributed software families development. In: Proceedings of the OOPSLA and GPCE Workshop on Best Practices for Model Driven Software Development. (2004)
48. Pitman, K.M.: Condition handling in the lisp language family. In: Advances in Exception Handling Techniques. LNCS (Lecture Notes in Computer Science) 2022. Springer-Verlag (2001) 39–59