Exception Handling in a Replicated Agent Environment

Zeina Azmeh, Christophe Dony, Chouki Tibermacine, Christelle Urtado, Sylvain Vauttier

Exception Handling in a Replicated Agent Environment

Semestrial report 2 — FACOMA — LIRMM partner

Zeina Azmeh[1], Christophe Dony[1], Chouki Tibermacine[1], Christelle Urtado[2], Sylvain Vauttier[2]

[1] LIRMM - CNRS and Montpellier II University - 161 rue Ada
34 392 Montpellier - France
{azmeh, dony, Chouki.Tibermacine}@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

1 Introduction

This document is the semestrial report of the LIRMM partner for the FACOMA project. It concludes semester 2 — out of 6 — for the project. During this semester, partners from LIRMM and LGI2P have worked on:

- defining an exception handling system for agents. This work is based on previous work that has been done by these partners on the SAGE exception handling system that was adapted to the context of the FACOMA project.
- refining their knowledge and understanding of the DIMAX[3] replicated agent environment provided by the LIP6 partner.
- setting the first elements of a specification of an exception handling system dedicated to agents that evolve in a replication-based environment.

Indeed, exception handling and replication are two mechanisms that increase the reliability of an application. Exceptions are situations in which the normal flow of a program cannot continue. Replication prevents an application from suffering from system failures. Exceptions and system failures do not apply to the same situations and therefore are complementary to each other. One of the objective of the FACOMA project is to study how an exception handling system and a replication mechanism can combine to increase the reliability of agent-based applications.

The remainder of this document is structured as follows. Section 2 specifies an exception handling system dedicated to agents. Then, Section 3 provides preliminary elements of a specification of an exception handling system for replicated agents.

[3] DIMAX is the result of the integration of an agent platform nammed DIMA and a replication framework nammed DARX.

2 An exception handling system dedicated to agents

As agent programming becomes more and more common, agent platforms need to integrate sophisticated mechanisms to ensure the reliability of agent-based applications. Exception handling is one of these capabilities that are widely accepted in most (sequential) programming languages but not yet common in agent platforms.

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 [1], 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 [2] in which handlers were some specialized actors, ancestors of today's exception supervisors, that had the same lacks, regarding handler contextualization (see Sect. 2.2), as Smalltalk or Ada initial lexical-scope handlers. Asynchrony has more recently motivated many research works in various contexts [3,4,5,6,7,8,9,10] but they only partially address agent needs, even if some agent systems integrate achieved exception handling proposals [11]. For example, the supervisor model described in [12,13] does not properly handle contextualization. Guardian [14,15] is a general and powerful solution which nonetheless proves to be complex to master and use. As explained in [14], *"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 agents 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 the way agents and agent-based applications are structured[4],
- is simple enough to be used by standard programmers.

The key requirements of the system are: to enforce encapsulation, to provide a representation for collaborative concurrent activities [1] so that they can be coordinated and controlled [16], to achieve caller contextualization [17,18] for handler definition and execution, to handle concurrent exceptions with resolution functions [19,20], 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.

Section 2 is organized in four parts. Subsection 2.1 recalls some basic vocabulary and introduces an example. Subsection 2.2 presents the rationale of

[4] We have considered agents 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.

our main conceptual choices. Subsection 2.3 describes the system specification focusing on the description of the asynchronous handler search policy.

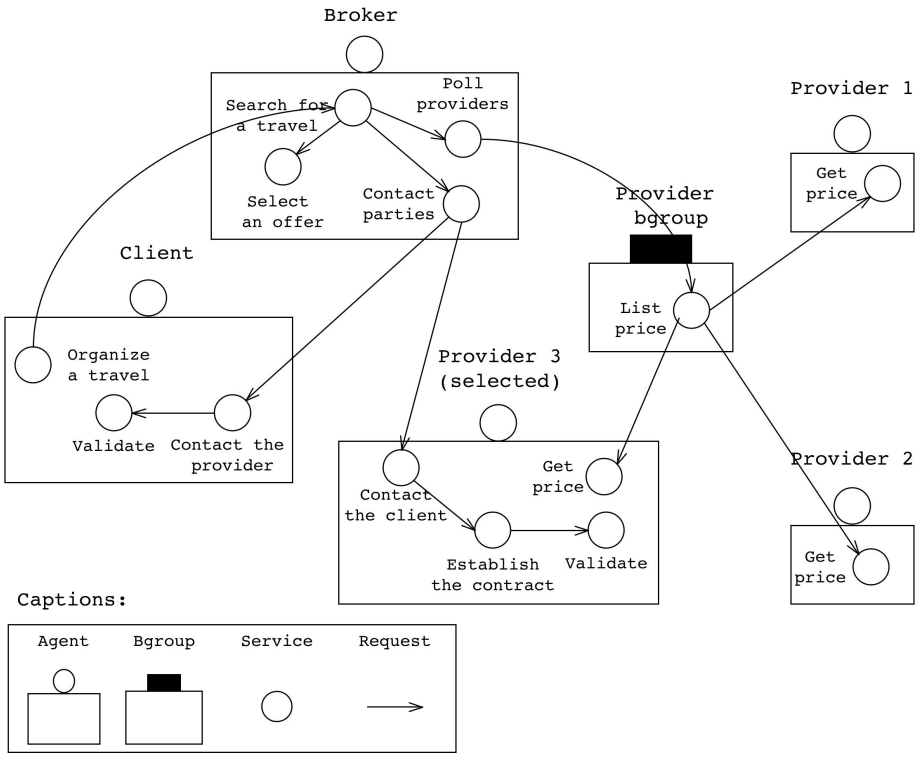

Figure 1. Execution resulting from a request to a travel agency

2.1 Definition, Terminology and Example

Agents communicate by exchanging messages that carry information [21]. Messages are queued in the object's message-box. Each agent executes a thread dedicated to managing its message-box: its scans and interprets the received messages to trigger corresponding actions. These actions are called **services**. Agents can execute several services concurrently in dedicated threads (intra-agent concurrency).

The request / response interaction protocol generally comes along with a contract-based approach of software development. It states that whenever an agent accepts a request, it must provide a response, either standard or excep-

tional. Our agents use one-way communications[5] 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 [21]. Agents 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 agents or several services of an agent in achieving a common goal.

As an illustration, we use the canonical *Travel Agency* example in which a *Client* can send a *B roker* 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 contract (*cf.* Fig. 1).

Our code examples use a Java-like syntax. Figure 2 shows examples of service definitions: lines 11–22 define the *Poll providers* service and lines 24–41 the *Contact Parties* service. We call **complex services**, services the code of which contains 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 messages that contain collective requests, are frequently used by agents. They are generally delivered via entities that represent groups of agents 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 agents. A *bgroup* possesses an (implicit) complex service that broadcasts the requests it receives to all agents 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 such an exception ?
- 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 such a handler be invoked and in which context ?

2.2 Rationale for the SaGE Exception Handling System

Each of the following paragraphs discusses the rationale of some of our choices.

[5] Two-ways asynchronous communications generally use *future* objects [3] which are more restrictive because the order in which they are read imposes synchronization constraints.

```
( 1)  public class Broker implements SaGEAgent
( 2)    {
( 3)    ...
( 4)
( 5)    public void handle (GlobalNetworkException exc)
( 6)    // handler associated to the Broker agent
( 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

Coordination of Concurrent Activities. As shown in several concurrent systems [16,1] and in our previous work [22,23], efficiently handling exceptions in concurrent systems using asynchronous communications requires *cooperative concurrency* to be supported. This amounts to provide a representation of collective activities and a way to define handlers 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* agent, should be invoked whenever one or more travel providers signal an exception, and should be able to access the *Broker* context.

Encapsulation. A well-known consequence of the introduction of exception handling primitives in a language is that it gives programmers constructs to break encapsulation [24]. 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 [2,25,12] or, more marginally, in procedural languages with handlers associated to shared data, as initially suggested by [26].

Supervisors are agents dedicated to exception handling, which can be considered themselves as handlers or to which handlers can be attached. The actor proposal for exception handling [2] is based on this idea. The issue with this approach is that supervisors are not encapsulated within the agents that experience the failure and therefore cannot access their internal state without breaking encapsulation. Our solution to prevent this, experimented in [18,22], is to define and encapsulate handlers within the agent or activity they control.

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 exception 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 environment. Its main advantages are its simplicity and the fact that it requires no additional language constructs. Its main drawback is that it fails to achieve fault tolerant encapsulations [18]. 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 dynamic approach in which handlers have a dynamic scope and are executed in the lexical context of the caller of the faulty routine. 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 agents are concerned, a choice has to be made among various alternatives. Original actor languages proposed dynamic scope handlers. [2] associates 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 [12] 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 [27] and Smalltalk in its original blue book version[6]. 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 agent systems by recent research proposals [22,15]. 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 [28], 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.

Resolution, Criticality. Entities that represent a set of collaborating agents are a natural place where to enable programmers to specify policies to deal with the multiple exceptions the agents may concurrently signal. Some resolution mechanisms to "concert" or "resolve" such exceptions have been proposed in [19,4,29]. 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 [30] 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

[6] 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: [...]`).

communication world, we 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.

2.3 Specification of the SaGe Exception Handling System for Agents

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.

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.

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

- **Request and acknowledgment**: a *sender agent* sends a *request* to a *receiver agent* [7], which can be an individual agent 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.

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

[7] 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.

- 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 2.1).
- Finally, handlers attached to **agents** (see Fig. 2, lines 5–7) are handlers common to all services, designed, for example, to maintain in an uniform way the coherence of the agent 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.

Signaling. Signaling is done by the means of a classical *signal* primitive (*cf.* Fig. 3). Signaling is possible anywhere in the code. This includes the possibility of signaling an exception from within handlers.

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

Figure 3. The *signal* primitive

Defining Handlers. Exception handlers are classically [31] defined by the set of exception types they can catch and by their body (as illustrated by Fig. 2, lines 32–37). There are three main actions a handler can classically have:

- A handler can 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 result of the service execution. In case of a handler attached to an agent, the value becomes the result of the execution of the service that raised the exception.
- A handler can **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 from the exceptions they signal.

– A handler can **retry** the execution of the program unit it is attached to. To retry [32,33] 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 agents, retrying means re-executing the service that signaled the exception.

Handler Search. Let S_n be the service in which an exception E is raised i.e.*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 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 yet interrupted. 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 agent 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.

If no handler is found locally, the search proceeds in the calling context (service S_{n-1}), in order to guarantee caller contextualization (*cf.* 2.2). First,

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 agent 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
```

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 resources. 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. 2.3).
- 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}.

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[8]
    if the resolution function returns a concerted exception then
      sce 's state ← "suspended"
      execute LocalSearch (exc, sce)
    end if
  end if
```

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 ease the reading 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.

Concerted Exception Support. SaGE provides exception resolution support (*cf.* 2.2) integrated to the handler search. It enables resolution functions to be defined at places where concurrent activities are launched and have to be coordinated (*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 agent level because all semantically sound activities of agents, that need to be coordinated, are accessible via services. *Bgroups* own resolution functions that are attached to the implicit complex service they execute. A *bgroup* first acts as a request broadcaster and then as a response collector in order to send back a single (composite) response to its client agent. 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:

- 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.

[8] 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

Such a use of resolution for concerted exception differs from the original work of [34,20] 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 [30]. 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.

3 Preliminary Specification of an Exception Handling System Dedicated to Replicated Agents

The goal of the system we have to develop is to give agents programmers (user of the DIMAX replicated agent platform) an exception handling system that works correctly in presence of replication. Replication provides automated fault-tolerance at the system level while exception handling allow programmers to develop additional programmed fault-tolerance at the application level. Putting the two systems together amounts to adapt exception handling solutions so that replication remains as transparent to programmers that it is without exception handling.

3.1 Overview of the Replicated Agent Platform

This section presents an overview of the DARX replication framework we have to adapt to.

Building a multi-agent system in the DIMAX framework requires that every DIMA agent extends the `DarxTask` class, in order to give the DARX middleware the ability to manage agents (start, stop, replicate, ..) and so that every agent be wrapped by a `TaskShell` that manages its input / output messages.

Every agent in the DARX framework has an underlying replication group, which is the set of all replicas corresponding to the same agent. Every replication group must have at least one active replica, considered as the leader. It can also have other active, semi-active or passive replicas, depending on the replication strategy.

Every replica in the DARX framework is given a unique identifier, the `ReplicantInfo`, provided by the naming server and built from the original name of the corresponding agent in the application context. The leader of a replication group distinguishes itself from a standard replica in that it has an additional thread that represents the `ReplicationManager`, and is attached to its `TaskShell`. This `ReplicationManager` is responsible for deciding what is the criticality of the associated agent, specifying the replication strategy, and maintaining the consistency of the replicas.

The consistency can be maintained following two main strategies:

1. the active one, in which all replicas process all input messages concurrently
2. and the passive one, in which only one of the replicas (the leader) processes all input messages and periodically transmits its current state (the serialized `DarxTask`) to the other replicas so they can update the state (`DarxTask`) contained in their own `TaskShell`.

Figure 5 illustrates the functionalities of the DARX middleware. The middleware gathers information on agents by analyzing their code using its application analysis service, in order to provide support without modifying the original program. Then, it evaluates the criticality of agents to decide which agents should be replicated, how many replicas must be created and which replication strategy (active or passive) is best adapted.

To explain how agent interactions are supported in DARX, we can see an example of agents interacting directly (without the DARX middleware) on Fig. 6, and the same agents interacting through replication group proxies (added by the DARX middleware) on Fig. 7.

When an agent `A` wants to send a message to an agent `C`, the original code (written by the agent programmer) will be something like:

```
A.send(msg, C);
```

In DARX, the naming server keeps track of replication groups and replicas. Every replication group is referenced by a local proxy. This proxy is implemented by the `RemoteTask` interface. It acts as an interface between the replicas in a replication group and the rest of the multi-agent system.

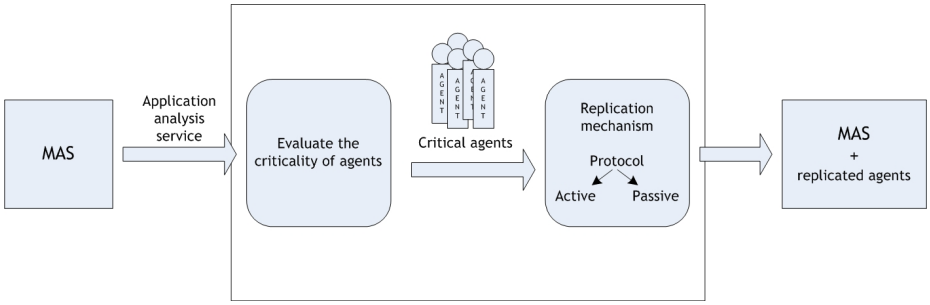

Figure 5. The DARX middleware

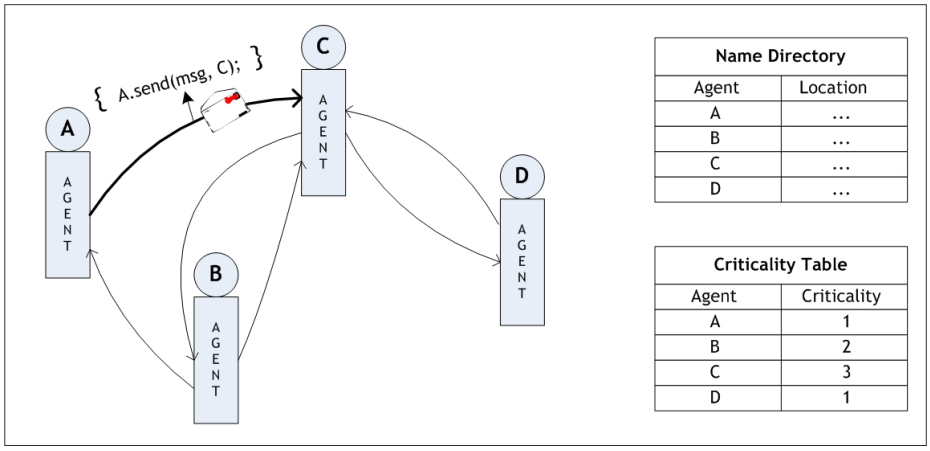

Figure 6. The original DIMA multi-agent system

All of the external and internal communications of a group are redirected to its proxy, which contains all the addresses of all replicas inside the associated replication group. This proxy will be obtained by a lookup request on the naming server using the application-relevant agent identifier.

So as we can see in Fig. 7, sending a message between two agents in Darx, using the previous line of code, will implicitly include obtaining the proxy of the receiving agent, then sending the message to this proxy:

```
ProxyC = NS.lookup(C); // Find a reference to agent C's proxy
A.send(msg, ProxyC); // Use the proxy to address agent C
```

Request / Response Message Management Inside a Replication Group. Messages exchanged by agents have different semantics that enables to achieve different interaction protocols. Classically, a request message is sent by a client agent to a server agent in order to ask for a service execution. After some calculation, the server agent sends back to the client agent a response message con-

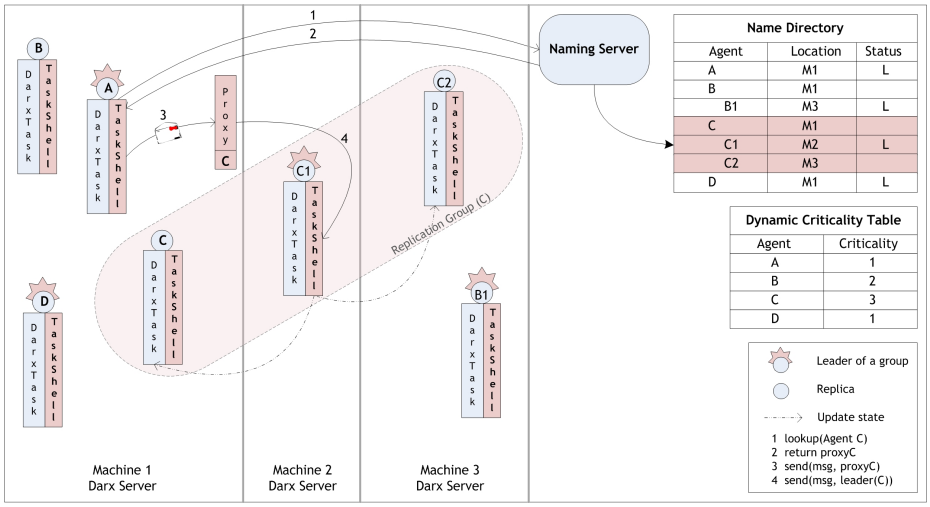

Figure 7. The DIMAX multi-agent system with replicated agents

taining a result. When agents are replicated with DARX, request and response messages are managed as follows inside the replication groups of the agents. A reasonable hypothesis is that the observable (external) behavior of all the active replicas belonging to a replication group is identical (if not it becomes difficult to guarantee that the states of the replicas can be synchronized and coherent). This means that starting from a same state and receiving the same message, all the replicas should send the same answer message.

Handling Request Messages.

Receiving request messages. For performance optimization reasons (proximity of two network nodes), any replica of a server agent is able to receive request messages. A request message is delivered to all the active replicas of the replication group, and especially to its leader (which is necessarily an active replica). Every active replica executes the corresponding service and sends a response message. The possibly new state of the leader is serialized by the DARX middleware and sent to every passive replica to update their states.

Sending request messages. As all the active replicas in a replication group are supposed to have the same observable behavior, they are likely to send the same request message to the same agent at the same time. In order to keep replication transparent for other agents (that are not necessarily replicated), only the request message from the leader is actually sent. Other request messages are discarded by the DARX middleware or possibly collected to be compared by some checking mechanisms.

Handling Response Messages.

Sending response messages. Only one response message must be sent back to a client agent in order to enforce replication transparence, although all the active replicas of a replication group will execute the same service at the same time and thus send many response messages. Here again, only the response message from the leader is actually sent. The other response messages are discarded by the DARX middleware or possibly collected to be compared by some coherence detection mechanism.

Receiving response messages. As for request messages, response messages are forwarded to the leader and then delivered to all the active replicas of the replication group. This way, all the active replicas handle the same answer and are expected to reach the same state. Here again, the new state reached by the leader is serialized and sent to update the states of the passive replicas.

Paradoxically, more replicas will provide more reliable agents but will also bear more exceptional situations to be managed, because of the multiplication of service executions and of message exchanges. Thus, the replication middleware can itself use an exception handling system (possibly the same as the agent level) for its own seek to manage this, as presented in the following, but this is a second problematic.

3.2 Introduction to the new problems

The goal of a replication system is to give the control to another replica of an agent when the active replica stops working. We will call this situation a system failure.

System failures are different from exceptions. An exception is not a failure because it is a kind of answer from an agent. It indicates that an agent is unable to continue its task the standard way but that he is still alive and potentially able to do other things.

Section 2 has presented how an agent can signal an exception and how handlers can be defined at different places to handle exceptional situations and put the system back into a coherent execution state.

The present study aims at presenting what issues, related to the detection, propagation and handling of exceptions, are raised by the fact that agents run on top of a replication system such as DARX.

The DARX system proposes different working contexts (different kinds of replicas that have different execution policies). We will deal first with the simplest ones. The questions we have to deal with to specify an exception handling system similar to the one presented in Sect. 2 and adapted to the context of the replication system are:

- What should happen when an exception is raised by an agent that has one or more replicas ?

- Where can programmers define handlers? Handlers can be defined both by the application programmer or by implementers of the replication system.
- What decisions can be taken within a handler ? Handlers defined by implementers of the replication system should certainly either put the execution back into a coherent state or propagate to the application level an understandable result, either normal or exceptional. In other words, the replication system should then return a normal result or signal an application level exception.
- How to distinguish an exception which represents a failure and should entail the election of a new leader from an exception which is a correct answer ? For example, signaling the "division-by-zero" exception is the normal reponse of the "divide" function if its second argument is zero. In this second case, it is useless to activate another replica and to elect a new leader because it will finally give the same answer. On this point, there is certainly a need to work on a typology of exceptions for each application.

3.3 Typology of exceptions

Application-level exceptions. We can in first analysis distinguish two kinds of application-level exceptions. We propose this classification to bring the problem to the fore, it seems difficult to predict before program execution to which category an exception will belong.

1. *Replica-specific exceptions.* This kind of exceptions is raised by a replica but not necessarily by the others. Examples include exceptions thrown when:
   - some resources specific to a given replica are unavailable, for example because the ressource is shared and the replica has lost the network connection.
   - a replica tries to communicate with an agent which is unreachable.
2. *Exceptions raised by all the replicas.* If a given replica raises such an exception, there is evidence that all the others will also raise it. It is for example the case of exceptions raised when:
   - some resources shared by all replicas are unavailable,
   - a parameter of the request sent to this agent (and all its replicas) is not valid (*e.g.,* leads to a division by zero),
   - the agent (and all its replicas) communicate with an unreachable agent.

System-Level Exceptions. These exceptions are raised by the replication system. We will focus on these exceptions later on in the project. As the replication system is transparent for the programmer, it is reasonnable to think that most of them should be handled internally. We will make on this point a link towards the ideas developed in [35] Our system will allow to deal correctly with all kind of exceptions that cross the systems level boundaries. Agent will on the one hand be able to define handlers to trap those (ascending) exceptions raised by the replication system that impact the agent level. For those orders that the agent

level want to transmit to the replication level, they can perhaps be delivered by direct message sending to the agents that implement the replication system. Should these orders be transmitted via the signaling of a (descending) exception, our system will, for any exception not handled within an agent, give the control to a handler associated to the agent replication manager that will be able to take the control and react appropriately.

3.4 Signaling and Handling Application-Level Exceptions

This section gives a first specification of how SaGE should be modified so that exception signaling and handling remain transparent for DARX programmers.

We will consider the case of passive replication strategy and the case of active replication strategy with the hypothesis that all active replicas of an agent share the same environment. Other cases will be considered later on.

Let us consider at this point two different cases : the replication strategy is either passive or active.

Passive Replication Strategy In this case, a replicated agent has one active replica (the leader) and several passive ones. An exception can only be signaled by the leader of the replication group. In this situation the first elligible SaGE handlers are in this order : the ones potentially defined by the programmer at the service (method) and at the agent level. Such an handler can either

- (case a) : modify the agent state and **return** a value
- (case b) : modify the agent state and ask for a **retry** of the execution of the code to which the handler is associated (either a message sending or a service) ,
- (case c) : resignal the same exception or signal a new one.

Cases a and b are transparent for DARX since the exception will have been handle internally by the agent.

Case c means that an exception will be signaled in response to a request to the agent. It is necessary to give the control to the replication manager. For this purpose, we will automatically associate a system handler to each replication manager that will be intertwined between the agent and the caller one. This handler will be systematically invoked and will have to deal with the following choice.

- If this exception is considered as a system-failure, this should have the same consequences for the replication system than a standard failure of the leader. The replication system should then probably re-evaluate the survivability of the agent to decide what should be done with the leader (reset its state, stop its execution, etc.).
- If the exception is considered as an answer and not a failure, the signaling process has to be pursued following the signaling strategy defined in section 2.3, i.e. a handler will be searched at the caller lever. As we are in a termination model, the execution context of the faulty service should be destroyed

and, from the point of view of the replication system, all replicas should be updated subsequently. For example if the execution of the faulty service has modified the agent state before signaling the exception, either the last correct state should be restored (roll-back), or the state of passive replicas should be updated.

Knowing whether the exception is failure or a result is certainly agent and application dependent and we have to imagine a way for the programmer to specify this for each exception that can be signaled by the agent.

Active replication Strategy. The active replication strategy is mode complex since there can be several replicas active at the same time. We then propose to associate, in addition to the previously described handler, a concertation function to the replication manager.

If one replica signals an exception that has not been handled by the user-defined handlers at the service or agent level, the concertation function has to be invoked. It has to determine whether the exception is meaningful or not. If it is, this should entail the invocation of the handler associated to the replication manager described in the previous (see section 3.4) section. If it is not, it means that it is advisable to wait to see :

- whether all replicas will signal the same exception (case b),
- or whether another replica will compute a standard result (case c).

Case b will be a confirmation that the agent is unable to perform the current request. The replication manager handler should be invoked an in section 3.4. Case c will have to be handled by the replication system in the same way as it handles the case of having different answers delivered by different replicas. This case clearly requires a deeper analysis that will constitute the starting point of the continuation of this work.

4 Conclusion and Future Work

In this report, we have first proposed a specification of an exception handling system dedicated to agents. We have especially focused on request / response interactions between agents. 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 agents. 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 agents 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 agents. 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 agent reactivity. So this model is especially suited for applications that need few synchronization and a high level of concurrency and reactivity.

Then, we have provided specification elements to the integration of this exception handling system into a replicated agent platform. In future work, these specification elements still have to be refined and extended. For example, the interactions between the replication mechanism and the exception handling system have to be further analyzed to manage system-level exceptions. Further on, these specifications must also lead to a prototype implementation of the proposed exception handling system.

This could lead to enhance DIMAX capabilities using the exception handling system as a "last chance" mechanism to signal failures when the DARX replication system has failed. This could be used in two distinct situations:

- to signal that the last remaining replica of an agent died (failed) in order to allow to trigger less performant modes the programmer migth have coded at the agent level,
- to signal the death (failure) of an agent that was not considered to be critical. This would allow to provide a recover strategy if the estimation of criticality was wrong.

This would imply that the DARX component dedicated to failure detection also detect those specific situations and raise an exception.

This could also lead to enhance SAGE capabilities using the meta-information on agents computed by the DIMAX platform. For example, an agent's computed criticality could be used to tune concertation strategies. We could also propose a vulnerability measure that would use information on:

- reliabilty of an agent (and its replica), using an exception history,
- criticality of an agent,
- number of existing replicas.

References

1. 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
2. Theriault, D.: A primer for the Act-1 language. Technical Report AI Memo 672, MIT Artificial Intelligence Laboratory (1982)
3. Halstead, R., Loaiza, J.: Exception handling in multilisp. In: 1985 Int'l. Conf. on Parallel Processing. (1985) 822–830
4. Campbell, R., Randell, B.: Error recovery in asynchronous systems. IEEE Transactions on Software Engineering (SE) **SE-12 number 8**(8) (1986) 811–826
5. Gärtner, F.C.: Fundamentals of fault tolerant distributed computing in asynchronous environments. ACMCS **31**(1) (1999) 1–26

6. 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
7. 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)
8. 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
9. Carlsson, R., Gustavsson, B., Nyblom, P.: Erlang: Exception handling revisited. In: Proceedings of the Third ACM SIGPLAN Erlang Workshop. (2004)
10. 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
11. Azmeh, Z., Dony, C., Tibermacine, C., Urtado, C., Vauttier, S.: Exception handling in a replicated agent environment. Technical report, ANR-FACOMA-06-SETIN-005-01 - T12 Report - LIRMM Montpellier (2008)
12. 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
13. Klein, M., Dellarocas, C.: Towards a systematic repository of knowledge about managing multi-agent system exceptions, ases working paper ases-wp-2000-01 (2000)
14. 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
15. Miller, R., Tripathi, A.: The guardian model and primitives for exception handling in distributed systems. IEEE Trans. Software Eng. **30**(12) (2004) 1008–1022
16. 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
17. Dony, C.: An object-oriented exception handling system for an object-oriented language. In: Proceedings of ECOOP'88. (1988) 146–161
18. 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).
19. 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
20. 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
21. FIPA: Foundation For Intelligent Physical Agents : Request Interaction Protocol Specification. (2002)
22. 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
23. 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
24. Miller, R., Tripathi, A.R.: Issues with exception handling in object-oriented systems. In: Proceedings of ECOOP'97. (1997) 85–103
25. 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)
26. Levin, R.: Program structures for exceptinal condition handling. Phd dissertation, Dept. Comput. Sci., Carnegie-Mellon University Pittsburg (1977)
27. 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)
28. 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)
29. Tartanoglu, F., Issarny, V., Levy, N., Romanovsky, A.: Dependability in the web service architecture (2002)
30. 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
31. 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)
32. Goodenough, J.B.: Exception handling: Issues and a proposed notation. Commun. ACM **18**(12) (1975) 683–696
33. Meyer, B.: Disciplined exceptions. Technical report tr-ei-22/ex, Interactive Software Engineering, Goleta, CA (1988)
34. Issarny, V.: Concurrent exception handling. In: Advances in Exception Handling Techniques. LNCS (Lecture Notes in Computer Science) 2022. Springer-Verlag (2001) 111–127
35. Marin, O., Guessoum, Z., Briot, J.P., Perrot, J.F.: Specification of a two-layer exception handling system. Technical report, INO CARE III "Towards Fault-Tolerant Cooperative Air Traffic Management" - Project Research Report - LIP6 (2007)