Improving Exception Handling in Multi-Agent Systems

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

[1] LGI2P - Ecole des Mines d'Alès - Parc scientifique G. Besse - 30 035 Nîmes - France
{Frederic.Souchon, Christelle.Urtado, Sylvain.Vauttier}@site-eerie.ema.fr
[2] LIRMM - 161 rue Ada - 34 392 Montpellier - France
dony@lirmm.fr

Abstract New software architectures based on multi-agents or software components allow the integration of separately developed software pieces that interact through various communication schemes. In such a context, reliability raises new important issues. This paper aims at increasing reliability in multi-agent systems (MASs) and, therefore, focuses on the study of an appropriate exception handling system (EHS). The issues specific to exception handling in MASs – preservation of the agent paradigm and support of cooperative concurrency – are presented and discussed. This paper analyses existing EHSs according to these issues and describes our proposition, the SaGE system, which integrates various solutions from existing EHSs and adapts them to the agent paradigm. SaGE is an exception handling system dedicated to MASs that addresses the stressed issues by providing means to coordinate the collective activities of agents, to embbed contextualized handlers in agents and to concert exceptions. It has been implemented and integrated in the MadKit MAS. It has been experimented with a classical travel agency case study.

1 Introduction

New software architectures (such as multi-agent systems [1] or software component based architectures [2]) are based on the integration of numerous software entities being executed concurrently and, in many cases, communicating asynchronously [3,4]. We are interested in reliability [5] concerns in the context of these new architectures but this paper chooses to narrow the focus on exception handling in the context of multi-agent programming. To our opinion, exception handling capabilities are a must-have to enable the realization of reliable large scale agent systems. To be adapted to multi-agent systems (MASs), we consider that an exception handling system (EHS) has to correctly deal with two main issues:

- preservation of the agent paradigm,
- support cooperative concurrency among agents.

This paper describes an EHS dedicated to multi-agent programming which addresses both issues and extends the MadKit platform [6].

The remainder of this paper is organized as follows. Section 2 introduces the terminology and main concepts of both exception handling and the agent paradigm. Section 3 discusses the issues of exception handling in the context of MASs and discusses related systems and solutions. Section 4 describes our proposition: the SaGE system which is illustrated through a comprehensive example. Finally, Sect. 5 and 6 explain the implementation of SaGE in MadKit and present our first experiments with this implementation.

2 Basic concepts

2.1 Exception Handling

Exceptional events, often called exceptions [7], allow to signal undesirable situations that hamper a program standard execution to continue. When such a situation occurs, a reliable software is able

to react appropriately by raising and treating an exception in order to continue its execution or, at least, to interrupt it properly while preserving data integrity as much as possible. An exception handling system [8,9,7,10,11,12,13,14] provides programmers with control structures that allow him to signal exceptions, to define exceptional continuations by the means of exception handlers and to put the system back in a coherent state by coding how some detected exceptionnal situation has to be treated. This latter objective can be:

- achieved by the handler itself by either continuing the execution where it was interrupted after having modified the context in which the undesirable situation occurred (resumption), or aborting part of the standard execution and resuming at some reliable point (termination),
- or, delegated to another handler by either propagating the exception or signaling a new one.

Signaling an exception provokes the interruption of what is currently being executed and the search for an appropriate handler. The definition of a handler (for one or more type of exceptions) results in associating code with program units which nature can vary from a system to another (e.g. the body of a procedure [8], a class [9], etc.). After an exception has been signaled, an appropriate handler is searched for among those associated to the program unit in which the exception has occurred. If one is found, it is executed. If not, the search carries on recursively in the enclosing program unit. The set of program units the exceptions of which can be treated by a given handler is called the handler scope. Depending on the models, this scope can be determined either statically (on the basis of the lexical structure of the code) or, more often, dynamically (on the basis of the history of execution contexts). This history is being built as a program unit (the called unit or the enclosed unit) is activated by another (the calling unit or the enclosing unit). An execution context is then created and associated to the called unit; it contains information related to its execution (local data, return address, parameters, etc.). This way, the calling unit precedes the unit being currently executed in the history of execution contexts. Systems that provide handlers with a dynamic scope rely on this history of contexts to determine the scope of a given handler at runtime. For example, the scope of a handler associated to a procedure covers the execution of the procedure itself and the execution of all the procedures it calls (and so on, recursively). This kind of mechanism is used, for example, in the C++ and Java languages.

2.2 Agent Paradigm

The main characteristic that distinguishes agents from other software components is their autonomy [15]. An agent has the ability to independently decide to realize an activity (among its capabilities) in order to fulfill individual objectives. Agents therefore execute concurrently in separate threads. However, agents are not isolated entities. They interact by exchanging messages thanks to asynchronous communication means. This way, agents are able to collaborate while preserving their autonomy.
Sending a message to another agent to request a service is a non-blocking action: the client agent (the sender of the request) does not have to wait for the response of the provider agent (the recipient of the request) and can carry on its current activity. The client agent will get the response later, in another message, sent in return by the provider agent. Conversely, receiving a message is a non pre-emptive action: the agent can decide to postpone the treatment of the message in order to achieve more urgent activities first.
Asynchonous communication provides agents with a means to manage advanced execution schemes such as to redundantly request the same service to different agents in order to ensure better performance, reliability or quality of service. For example, a client agent can calculate and consolidate a result with the different responses it gets from a pool of provider agents and then decide to stop the process when a given amount of time has been spent to collect information (to guarantee some time performance), or that a given ratio of responses has been received (to guarantee some result representativity). This can also be used to manage collective activities when agents are structured in social organizations. In Aalaadin (the social organization model used in MadKit) [16], every agent is member of a group, in which it plays one or more roles that define the different

responsibilities (capabilities) of the agent. The role then acts as a common interface for a set of agents. Inside a group, a request can be sent to an individual agent or to a role. In this latter case, the request is transparently broadcasted to all the agents that play the role. The client agent can therefore get a collective response from all the members of its group that hold a given capability, while ignoring both their identity and their number.
Nonetheless, the autonomy of agents should not interfere with the concept of contract-based collaborations [12], that is a fundamental principle for reliable software engineering. Indeed, the service requested by a client agent to a provider agent may be essential to complete some activity. A provider agent is free to reject a request but when it accepts to provide another agent with a service, it must fulfill its commitment. Thus, when an agent undergoes a failure that prevents it from executing any requested service, it must warn the concerned client agent that it is not able to provide the service it expects. The client agent will then be able to react to this situation and try some alternate way to obtain the needed service or to achieve the impacted activities.
In other words, agents need an exception handling system too, in order to take reliability issues into account in the management of their collaborations. But because of the principles of the agent paradigm, that imply the use of specific execution and communication models, exception management in MASs raises specific issues that cannot simply be addressed by the EHS of the underlying implementation languages. Section 3 presents these issues and discusses some related work.

3 Exception Handling in MaSs: Issues and Related Work

In this Sect., we study exception handling in the context of multi-agents platforms and discuss the two main issues we consider to be specific to exception handling in this context.

3.1 Preservation of the Agent Paradigm

As argued in Sect. 2.2, we consider agents from a software engineering point of view (we do not consider their cognitive capabilities) and, therefore, think that multi-agent systems, just as software coded in other paradigms, need exceptional situations to be dealt with adequate control structures.

A need for a specific EHS. Agents are not native concepts of the language chosen to implement the MAS. They are higher level entities that use specific communication and execution mechanisms in order to conform with the agent paradigm. The management of exceptions at the agent level therefore requires a specific EHS that is integrated and adapted to these mechanisms: exceptions should be propagated from provider agents to their client agents thanks to messages; agents should immediately react to messages signaling exceptions by searching for handlers associated with the activities that are affected by theses failures. This specific EHS and the EHS of the programming language must not be mistaken. They handle different categories of exceptions (as classified by [17,18]): the latter is used to handle low-level exceptions (related to the implementation and the execution environment); the former deals with high-level exceptions, regarding the execution of the activities of agents, their interactions or their management within the MAS. But the two EHS are related: an implementation level exception that causes the failure of the execution of an activity is to be transformed into an agent-level exception that can then be handled by the EHS integrated to the execution model of the agents.
Among the existing MASs [19], very few provide a specific EHS. Most of the MASs limit their exception handling capabilities to those provided by the programming language. For example, MadKit [6] is a generic MAS coded in Java that does not prescribe any specific execution model and, as a consequence, does not provide any specific EHS. The activities of agents are coded as methods and the exceptions raised during their execution are classically treated by the handlers associated with blocks of code of these methods. When an exception cannot be treated by these handlers, it is propagated to the top level of the call-stack of the thread in which the agent executes. The thread is then destroyed by the Java virtual machine. From the MAS point of

view, this corresponds to the accidental death of the agent. The lack of an appropriate agent level EHS implies that no exception can be propagated to other agents or more simply to the MAS itself, as exceptions are not supposed to be propagated outside the execution thread [3]. In such a context, agent level exceptions have to be managed with ad hoc solutions such as to signal a failure to a client agent by replying to its request with a message containing special values. But this solution contradicts software engineering good practices that recommend the management of errors thanks to a dedicated means like an EHS. Indeed, separately developed agents are unlikely to have a shared interpretation of those special values and the code of activities then mixes up the treatments of normal responses – that describe the standard behavior of the agent – with the treatments of exceptional responses –that describe the corresponding exceptional behaviors used to deal with critical situations.

Supervisor-based EHSs are not sufficient. The specific EHSs designed for some MASs [20,21] use a supervisor-based approach. Supervisors are specialized agents which role is to monitor the activities of other agents, in order to catch the exceptions they could raise. The supervisors are the entities to which handlers are attached. We advocate that this approach is more adapted to handle generic problems at the MAS level (such as the death of agents [20]). Indeed, from the point of view of the monitored agents, supervisors are external entities created by the MAS. The management of the particular problems related to the inner activities of an agents requires that specific, contextual handlers (aware of the precise impacts of the failures signaled by the exceptions on the current activities of the agents) be defined and triggered. As external entities, supervisors cannot easily manage such contextual handlers, unless supervisors have some right to act intrusively on the behavior of agents or unless agents inform supervisors of specific exception handlers, and thus delegate to supervisors the management and execution of parts of their behavior. In both cases, it contradicts encapsulation, abstraction and autonomy principles.
We thus propose a different and complementary solution that is more natural to deal with contextual exception handling: handlers designed to treat the exceptions regarding activities of agents should be associated with these activities, and, by the way, encapsulated and managed by the agents themselves, as part of their behaviors. Exceptions that cannot be successfully caught and treated by the handlers of an agent are then propagated outside of the agent, whether to agents with which it collaborates as a provider for some service, or to the MAS (represented by supervisors) for more generic system-level problems.

As a conclusion, we claim that the development of a specific EHS is essential to exception handling in MASs (the underlying language exception capabilities are not sufficient for agents) and that the EHS must not solely rely on a supervisor-type architecture.

3.2 Cooperative concurrency support

A multi-agent system is made of software entities (agents) that execute concurrently. Solutions to handle exceptions in MASs can thus be derived from work on concurrent programming. This section presents two concepts proposed for concurrent programming that we adopt and adapt in our proposal for agent-based programming.

A Need for Activity Coordination. [22] proposes a classification that distinguishes between three kinds of concurrency in object-oriented systems and studies their impact on exception handling. This classification is provided for classical concurrent object-oriented systems. In such systems, execution threads are orthogonal to objects: objects are passive entities that are executed by external threads. Conversely, agents are active (concurrent) entities that hold their own threads and use this processing power to act as autonomous entities. Thus, in MASs, threads are not disincarnated: a thread is created for the purpose of executing a given activity of a given agent and

[3] MadKit does not use the java *ThreadGroup* notion, that would be of little help as it does not support distribution

its life cycle follows the life cycle of this activity. The classification of [22] must thus be adapted to this specificity of MASs but still applies in our case.

First, **disjoint concurrency** points out the kind of concurrency supported by systems that actually provide no way to manage concurrency. Disjoint concurrency means that each agent is managed as if it is the only active entity in the system. Exception handling is therefore local to each agent. There is no need to provide any mechanism to coordinate exeception management between agents as no collective activities are considered. Second, **competitive concurrency** points out the kind of concurrency supported by systems that manage the isolation of each active entity. These systems provide mechanisms to avoid the inconsistencies caused by the concurrent use of shared resources (generally thanks to lock-based schemes). The goal of such systems is to let every agent act as if it was the only active entity in the system, but in a disciplined way. Exception handling is coherent with this concurrency policy: Exception management is still local to each agent. Finally, **cooperative concurrency** points out the kind of concurrency in systems that provide some support to the management of collaborations between active entities. In a MAS supporting cooperative concurrency, it should be possible to coordinate the individual activities of the set of agents that contribute to a collective activity. [22] claims that cooperative concurrency management requires an execution model that allows these collective activities to be explicitly represented. An EHS could then be designed that allows handlers to be associated with a collective activity, in order to express and manage the impact of the failure of every participant agent in such a global execution context. Exception propagation schemes can then be elaborated, from the local execution context of the individual activity of an agent, to the more global execution context of the collective activity to which it participates, and recursively, to the even more global execution context of an enclosing collective activity.

To summarize, **MASs must provide support for cooperative concurrency**, as collaboration is a fundamental principle of the agent paradigm (because of the social abilities of agents). As a consequence, the execution model of MASs must provide a means to explicitly represent and control the collective activities of agents.

MASs using supervisors in their EHS [20,21] support a form of activity coordination: a supervisor can be associated to a group of collaborating agents to monitor the global activity of the group. MASs that do not provide a specific EHS must use the exception handling capabilities of their underlying programming language. Those written in Java can use an advanced notion to coordinate the threads of agents: thread groups. A thread can be attached to a thread group. When an exception is raised inside a thread, that is not caught and reaches the top-level of the call-stack of the thread, the exception is propagated to the thread group to which the thread belongs. The exception can then be caught by handlers associated to the thread group so that the impact of this exception on other threads of the group can be managed (such as to stop the threads that are dependant on some result that the faulty thread has not produced). The thread group acts as a coordinator for the activities of its belonging threads. A thread group can in turn be part of a larger thread group. This allows large, hierachical thread structures to be build, in order to control collective activities at different nested levels. Thread groups are an interesting Java construct to coordinate concurrent thread activities but:

1. they are not used in the EHSs of the MASs we surveyed,
2. they do not provide a direct support for distribution or asynchronous messaging. Therefore, they are not sufficient, as is, to manage the coordinated activities of agents (see Point 1 in Sect. 3.1).

Concerted Exception Support. Once activity coordination is supported and exception handling mechanisms are integrated, a second step is to define a management policy that determines the impact of the co-occurrence of minor failures while completing a global activity. [3] and then [23] suggest the integration of a mechanism to concert exceptions. Exceptions concurrently signaled by the entities participating to a collective activity are composed together, by a resolution function, as a unique exception (called a concerted exception) that reflects the global state of the collective activity. This concerted exception is used instead of the individual exceptions raised by

the participants to trigger handlers at the collective activity level. This is a general principle that can be adapted, through different implementations of its basic concepts (concerted exception and resolution function) to the exception handling mechanism of any system supporting cooperative concurrency: the entity that is responsible for a collective activity is able to collect the exceptions that are propagated from the participant entities and then to compute a unique, pertinent global exception. Another advantage of this scheme is the versatile support for exception management policy definition it provides. A redefinition of the resolution function associated with a collective activity is enough to change its policy.

The next Sect. depicts how these concepts are adapted and used in our EHS proposal for MASs.

4 SaGE: an Exception Handling System Dedicated to Multi-Agent Systems

This Sect. presents our proposal: an EHS that tackles the two issues stressed in the previous section. We will use a unique comprehensive example to illustrate our model throughout the remaining of the paper. It is based on a classical travel agency case study (see Fig. 1) in which:

1. A *Client* agent contacts a *Broker* agent in order to organize a travel and get the best offer for its plane or train tickets (the transport means is chosen randomly during initialization).
2. Depending on the request made by the *Client*, the contacted *Broker* sends a request to train providers or plane providers to collect their bids.
3. Then, the *Broker* selects the best offer and requests both the *Client* and the selected *Provider* to establish a contract.

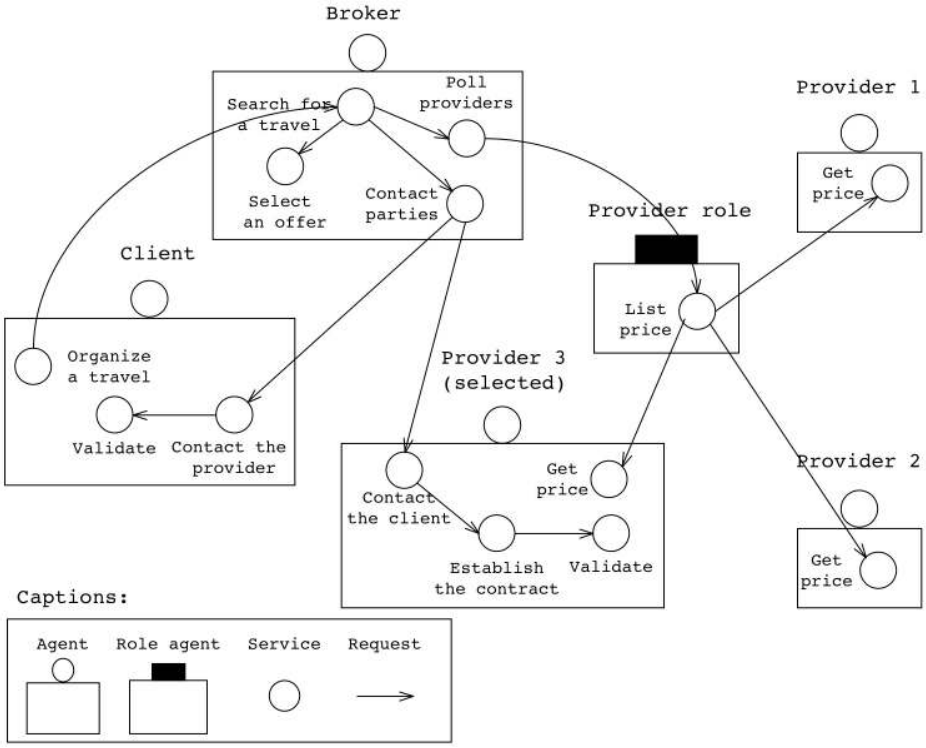


Figure 1. Execution resulting from a request to a travel agency

4.1 An Execution Model that Allows Concurrent Activity Coordination

The MAS we use for our experimentation, MadKit, is a generic MAS that does not prescribe any predefined, fixed execution model: it only provides a framework of versatile communication and management mechanisms for agents. Thus, the execution model presented in this Sect. is the one we designed for MadKit agents because we need cooperative concurrency between agents. Intra-agent concurrency is also mandatory in our system in order to preserve the responsiveness of agents to critical events such as exception signaling. Though initially designed for MadKit, the execution model presented here is generic and we propose it as a solution to manage the coordination of concurrent activities that could be transposed to any MAS.
Exception handling in MASs requires a rather sophisticated execution model. Indeed, agents must always remain responsive to messages, particularly to critical messages such as interruption calls (a client agent signals to one of its provider agents that it does not need the service it requested anymore) and exception signaling (a client agent signals to one of its provider agent that it cannot successfully treat a request because of some failure). To ensure its responsiveness, every agent owns a thread dedicated to actively scanning its message-box in order to be able to trigger actions as soon as a message is received. The model we propose is built on the service concept.

Services. When a request sent by a client agent is received and accepted by the recipient agent, the execution of the corresponding service is initiated by the recipient agent which then acts as a provider agent. In SaGE, a service is a reified concept; it is executed in its own thread and it defines an execution context. The execution of services can explicitly be controlled by their owner agents (for example, to interrupt them). Figure 2 shows how services are created in the SaGE system.

```
public class Broker extends SaGEAgent
{
   Service s = new Service ("Search for a travel", getAddress(), serviceID)
      {
      public void live ()
         {
            // body of the service
         }
      public SaGEException concert (Vector subServicesInfo)
         {
            // body of the exception resolution function
         }
      public void handle (NetworkException exc)
         {
            // handler for NetworkException
         }
      public void handle (NoProviderException exc)
         {
            // handler for NoProviderException
         }
      };
}
```

Figure 2. Service definition and association of handlers to services in SaGE

These services fall into two categories:

- **atomic services**, the execution of which does not depend on other services. For example, in the travel agency study case, the *Get price* service (see Fig. 1) that returns a *Provider*'s bid is atomic because it needs no subservice.

– **complex services**, the execution of which requires other services to be achieved. For example, in the travel agency study case, the *Organize a travel* service (see Fig. 1) which handles a *Client*'s initial request is complex as it sends requests that trigger additionnal service executions.

The nature of these two kinds of services is very different. An atomic service can be implemented by a simple thread that executes the corresponding treatment and, then, sends back to its client the corresponding answer. A complex service has to be implemented by an entity that is able to send requests and recieve responses. Consequently, implementing complex services as agents is necessary and natural, as complex services need the same communication capabilities and cooperative concurrency support as those of agents. Cascaded requests result in cascaded service executions that form a logical structure (tree) of execution contexts. Figure 1 shows the graph of execution contexts that results from the cascaded requests in the travel agency example. This structure is comparable to the call-stack in procedural or object-oriented programming: it provides an explicit representation of both the individual (atomic services) and collective (complex services) activities of agents that enables the management of cooperative concurrency.

Every time a provider agent receives and accepts a request, it logs the ID of the client agent for which it is executing the corresponding service. Accordingly, every time a client agent successfully sends a request, it logs the id of its demanding service, the id of the request, and the id of the provider agent. This log is used to return responses, propagate exceptions and manage the termination of subservices.

Managing Collective Requests. Section 2.2 introduced the concept of role which allows the broadcasting of messages to a set of agents that share a common ability. In order to manage such collective requests, the execution model has to be extended with entities that represent and manage roles. As for services, such entities must be able to send and receive asynchronous messages. In our proposition, we choose to consider these entities as dedicated agents called **role agents** that:

– maintain a list of its participating agents (those which play the corresponding role),
– define a generic treatment for the recieved requests that consist in broadcasting messages to all its member agents,
– and, collect answers and exceptions from its member agents and combine them into a pertinent collective response or concerted exception.

The execution model described in this subsection provides a means to coordinate the activities of agents as illustrated by the example of Fig. 1. In this example, *Broker* is an agent, *ProviderRole* is a role agent to which three *Provider* agents have subscribed. This execution model is used to integrate the exception handling system presented below.

4.2 An Exception Handling System Dedicated to MASs

Definition of Handlers. In SaGE, exception handlers can be associated with services, agents or role agents.

1. **Handlers associated with a service** are designed to catch and treat exceptions that are raised, either directly or indirectly, while executing the service. This enables a precise, contextual, definition of handlers: the objective of the service, its current state and the impact of exceptions on its completion can be taken into account when coding the handler.
2. **Handlers associated with an agent** are a practical means to define a single handler for all the services of this agent at a time. For example, the death of an agent or the coherence maintenance of agent-specific data can be dealt with such exception handlers.
3. **Handlers associated with a role** are designed to treat exceptions that concern all agents which play a given role. For example, when exceptions occur during the handling of a broadcasted request, partial results or QoS statistics can be returned by handlers at the role level.

These handlers are distinct from the handlers provided by the underlying implementation language. They are triggered by the signaling or the propagation of an agent-level exception. A handler is classically defined by the set of exception types it can catch and by an associated treatment (as illustrated by Fig. 2). SaGE provides a termination model that allows a handler to:

- execute some treatments, in order to manage the consequences of the abrupt interruption of a service execution, such as restoring the agent in some coherent state, sending some partial results, etc.,
- send in turn an exception that signals that it has not been able to successfully manage the exception,
- re-launch a complete execution when associated with a service, after having possibly modified the execution context in order to re-try to successfully achieve it.

Exception Signaling. Agent-level exceptions are signaled, during the execution of services (or their associated handlers), thanks to calls to a specific primitive (see Fig. 3). Both exception systems are compatible: language level exceptions, caught by language-level handlers, can be turned into agent-level exceptions by calling the SaGE exception signaling primitive within the language-level handlers. As other exception signaling primitives, ours takes the exception to be signaled as a parameter. A call to this function internally triggers the exception handling mechanism of SaGE.

```
signal (new SaGEException ("Bad client address", getOwnerAddress ());
```

Figure 3. Exception signaling in SaGE

Handler Search. The heart of an exception handling system is the way handlers are searched for. When an exception is signaled, the execution of the defective service is suspended. First, a handler for this type of exception is searched for locally, i.e. in the list of handlers associated with the service. If such a handler is found, it is executed. If not, the search carries on among the handlers associated with the agent that executed the defective service. In all the above cases, the defective service is terminated.

If no adequate handler is found in the previous step, it means that the service failed and that the consequences of its failure must be dealt with by the client. Thus, the exception is propagated to the client agent that forwards it to its concerned service. The search carries on there, first in the service itself and, then, in the agent for which it executes. This client agent can either be an agent or a role agent.

If no handler is found, the search process iterates (the whole process is illustrated in Fig. 4) until an adequate handler is found or the top-level is reached. In the latter case, the whole computation is aborted.

Concerted Exception Support. Inspired by [23] and [24], SaGE integrates concerted exception support in its exception propagation mechanism. This mechanism allows:

- not to react to under-critical situations,
- to collect exceptions to reflect a collective or a global defect.

The concerted exception mechanism is available both at the service and the role level. No concerted exception handling is required at the agent level. Indeed, the association of handlers to agents is provided as a facility to define handlers that are common to all the services of the agent. As such, they are managed as handlers associated with services. Thus, the exceptions that trigger these handlers are concerted at the service level by the exception resolution functions associated with the services.

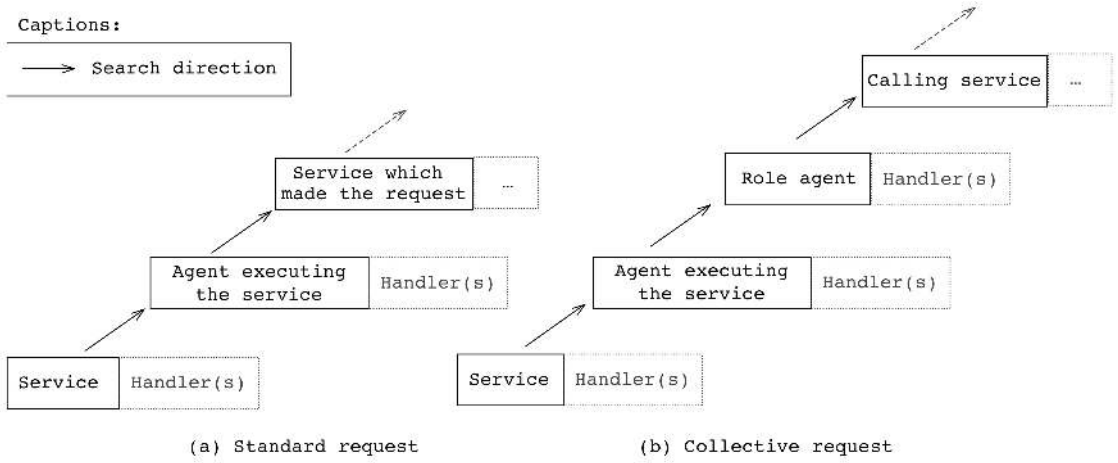


Figure 4. Handler search process in SaGE

Concerted exception support at the service level. An exception that is propagated to a service is not always critical for the service completion. Indeed, a request can be redundantly sent to several agents to increase reliability and performance. This is, for example, the case when a *Provider-RoleAgent* sends n requests to n *Provider* agents: the failure of few providers is not critical. In such a case, only the failure of a significant proportion of the requested service providers might be critical. This example is illustrated on Fig. 5.

To enable concerted exception support, propagated exceptions are not directly handled by the recipient service. Such exceptions are stored in a log which is associated to the recipient service. This log maintains the history of the so far propagated exceptions (along with information such as the sources of the exceptions). Whenever a new propagated exception is logged, the concerted exception function associated to the recipient service is executed to evaluate the situation. This function acts as both a filter and a composition function. Depending on the nature or the number of logged exceptions, this function determines if an exception is to be effectively propagated. If so, the propagated exception can be the last propagated one (in case it is critical enough) or a new exception that is calculated from a set of logged exceptions, the conjunction of which creates a critical situation (represented by a concerted exception).

As for handlers, resolution functions are associated with services and each resolution function is specific to a service. To write such functions (see Fig. 5), programmers have access to the exception log – in order to decide if an exception is to be propagated – and to the exception signaling primitive – to effectively signal the chosen concerted exception.

Concerted exception support at the role level. The set of requests emitted by a role agent to manage a collective request is transparent for the client agent that sends a request to a role agent. The role agent acts as a collector for responses and sends back a single (composite) response to its client. A comparable scheme is used to concert exceptions.

Whenever an exception is propagated from an agent belonging to the role, the exception resolution function, associated to the role agent, is invoked. It logs the exception and, when the cumulative effects of the under critical exceptions becomes critical, it computes the concerted exception to be effectively propagated to reflect the actual global situation.

In the travel agency example, there are cases where concerted exceptions are required. The *Poll Providers* service of the *Broker* agent broadcasts a request to get prices from *Provider* agents. None of these requests is individually critical. Thus, the exception resolution function (see Fig. 5) associated with the *ProviderRole_ RoleAgent* role agent will collect the exceptions signaling the failures of *Provider* agents without signaling any exception until a critical proportion of these

```
public SaGEException concert (Vector subServicesInfo)
{
   int failed = 0;
   int pending = 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++;
      else ((ServiceInfo) (subServicesInfo.elementAt (i)).isFinished () == false)
         pending ++;
      }

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

   // if not, at the end, only few providers failed
   if (failed != 0 && pending ==0)
      return new SaGEException("few_bad_providers",getAddress());

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

Figure 5. Exception resolution function associated to the *TravelProviders_ RoleAgent* role agents

agents fails. There are also cases where individual exceptions are critical. Services like *Select an offer* or *Contact parties* are critical for the successful completion of the *Search for a travel* service: their failure immediately results in the failure of the client service (the concerted exception function associated with the *Search for a travel* service does not filter exceptions propageted from the *Select an offer* or *Contact parties* services nor delay their handling).

5 Overview of the Implementation of SaGE for MadKit

For this first implementation of the SaGE model, we did not modify the kernel of the *MadKit* platform but choose to specialize classes from its core implementation (*AbstractAgent, Agent, ACLMessage*) along with the standard Java *Exception* class (see Fig. 6).

5.1 Communication

In *MadKit*, agents are referenced by their logical *AgentAddress* adresses. These addresses are used to route *ACLMessage* messages to the recipient agents through the middleware. The class *ACLMessage* has been specialized in order to encapsulate data which is specific to our execution model:

- identifiers used to manage the internal message forwarding from agents to their services,
- standard definition of message categories (such as *request*, *finish*, *terminate*, *exception*) [25].

5.2 Exceptions

We have extended the standard Java *Exception* class in order to:

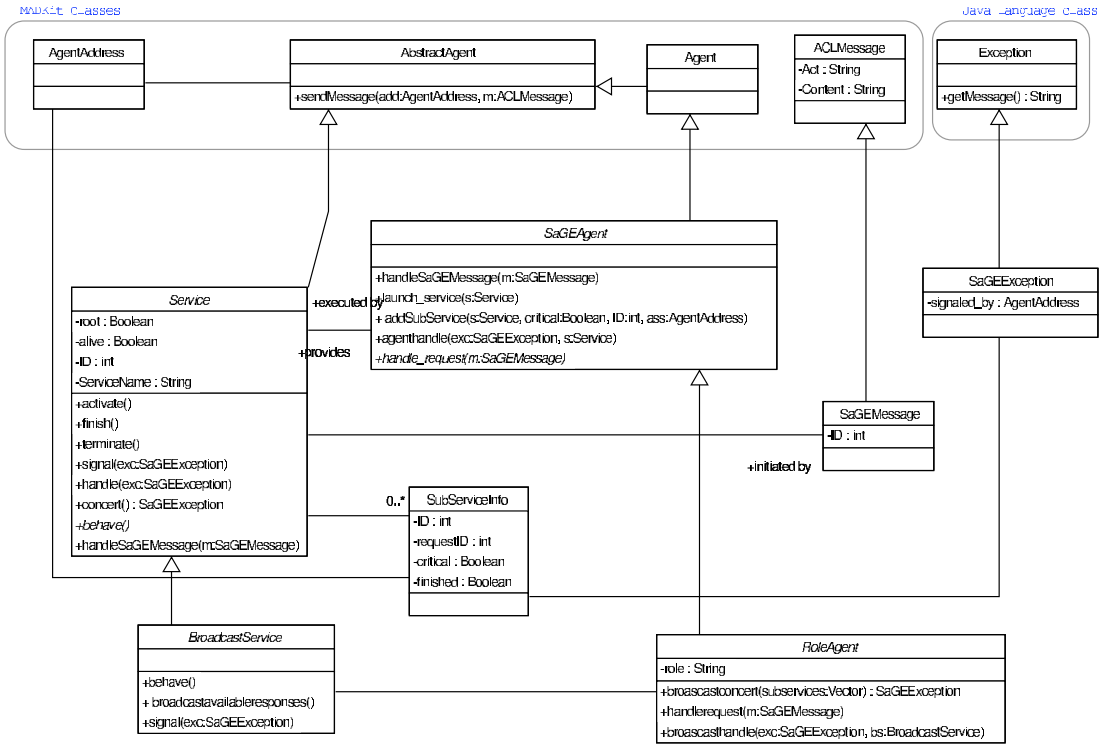


Figure 6. Class Diagram of the SaGE model

- differentiate agent-level exceptions from language-level ones,
- encapsulate information which may be useful for programmers such as the address of the agent which signals or propagates the exception,
- remain compatible with the standard Java EHS (agent level exceptions can thus be, if required, simply considered as standard Java exception and finally caught and treated by classical Java handlers).

5.3 Agents and Services

The *AbstractAgent* class is the base class of the communicating entities in the system: every *AbstractAgent* instance has an address, can send messages to another *AbstractAgent* and read the received messages in its message box. The *Agent* class is the base class for all the active entities in the system. It extends the *AbstractAgent* class by adding the *live* method, the method that is first executed after an agent is created and a thread is attached to it by the kernel of the system. This method is to be overridden in subclasses in order to define the main behavior of the agents in a given execution model. The *live* method in the *SaGEAgent* class implements a loop that actively scans the message box of the agent. When a message is received, it is handled by the *handleSaGEMessage* method that calls more specific methods depending on the category of the message (*request*, *exception*, etc.). Though they are active entities too, the base class of services, *Service*, is not implemented as a subclass of *Agent*, not to be mistaken with agents (services are internal entities, encapsulated in agents). However, *Service* is a subclass of *AbstractAgent* in order to inherit of the same communication capabilities as agents[4].

The *SubServiceInfo* class is used by services to reference the services they request along with management data such as their significance and their execution status.

[4] In the current implementation, we do not differentiate atomic services from complex services.

5.4 Broadcasting

Roles agents (introduced in Sect. 4.1) are implemented as a specialized class of *SaGEAgent* agents by the *RoleAgent* class. They handle the broadcasting of the requests they receive with the generic *broadcastService* method. It is to be noticed that the *RoleAgent* class is defined in such a way that defining a role in a SaGE application only implies defining which role the corresponding *RoleAgent* manages (see Fig. 7) and, optionally, associating handlers and a dedicated resolution function with it (see Fig. 5).

```
public class TrainProviders_RoleAgent extends TravelProviders_RoleAgent
{
   public TrainProvider_RoleAgent ()
   {
      super("train-provider");
   }
}
```

Figure 7. Definiton of the *Train-Providers RoleAgent*

6 Experimentation

After having implemented SaGE, we experimented it with the travel agency example (see Fig. 8).

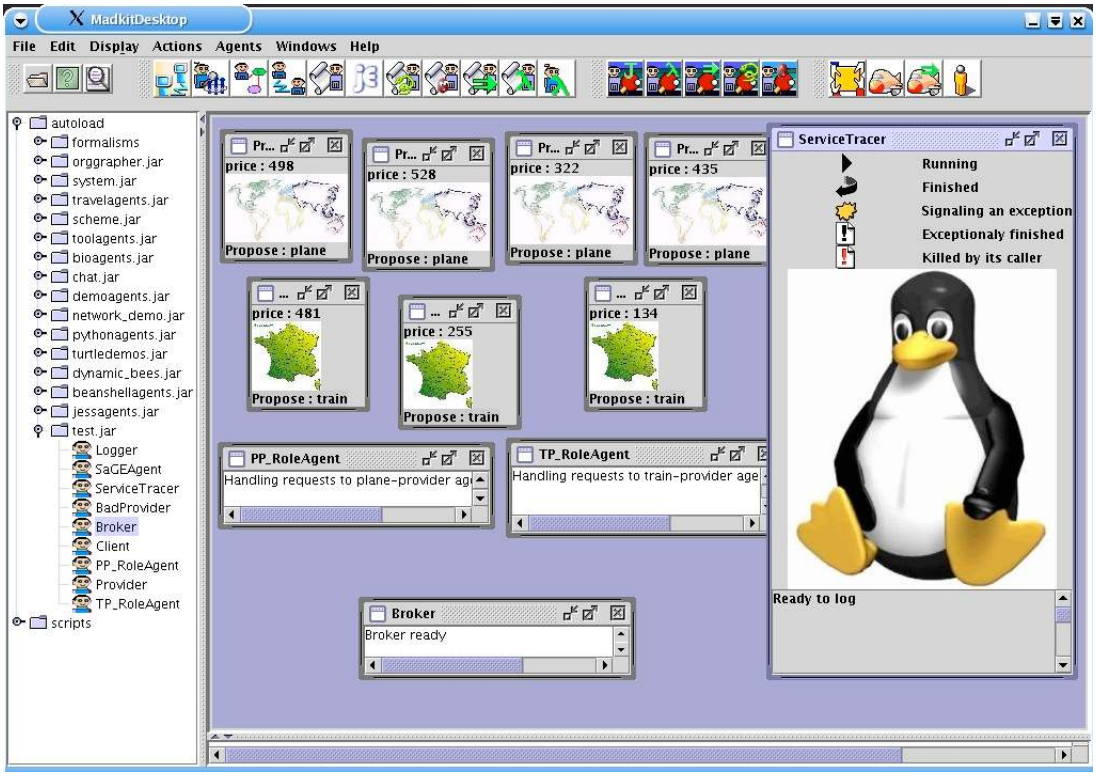


Figure 8. A typical MadKit window with launched *SaGEAgents*

6.1 Definition of the Agents of the Travel Agency Example

In order to implement this example with SaGE, we had to implement three agents (*Client*, *Broker* and *Provider*) and two role agents which handle collective requests for both transport means (see Fig. 9): the *TrainProviders_ RoleAgent* role agent and the *PlaneProviders_ RoleAgent* role agent.

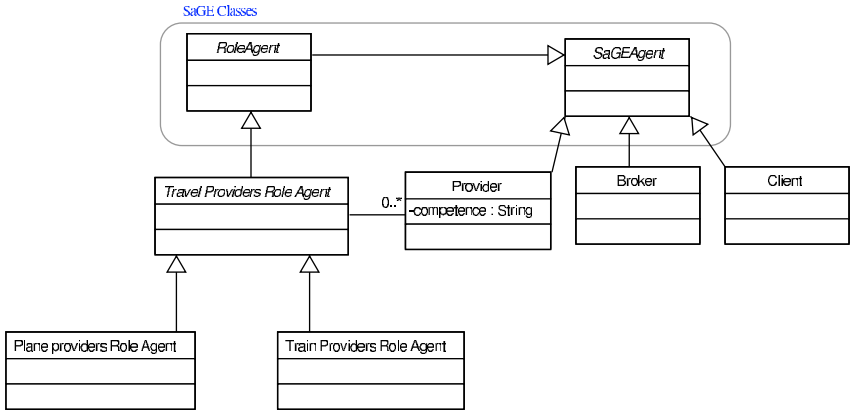


Figure 9. Class Diagram of the SaGETravel demo

In addition, in order to allow the debugging of SaGE agents, we provide two extra agents (see Fig. 8):

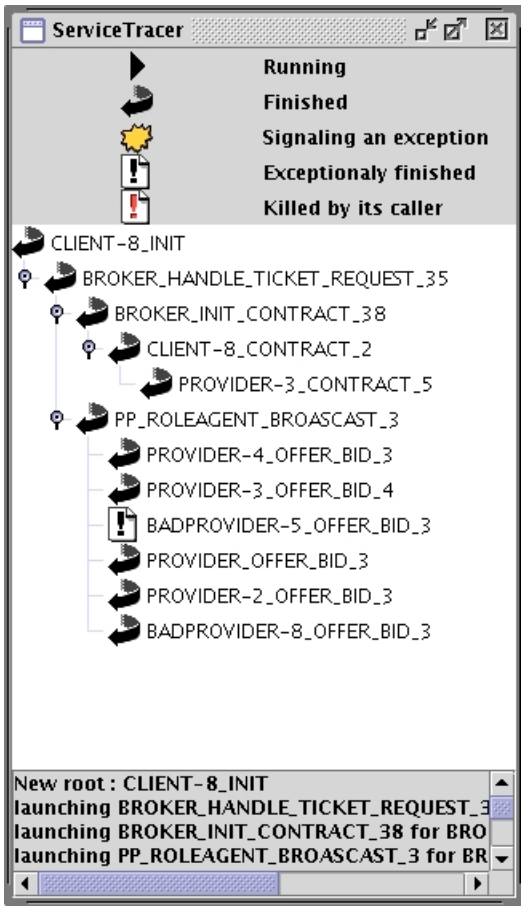
- the *Logger*, which logs into a text file actions related to services (instantiation, initiation, termination, exception signaling, etc.),
- and, the *ServiceTracer* which visually and dynamically represents the service tree, the root of which is the service initiated by the last launched *Client* agent (see Fig. 10).

6.2 Concerting Exceptions

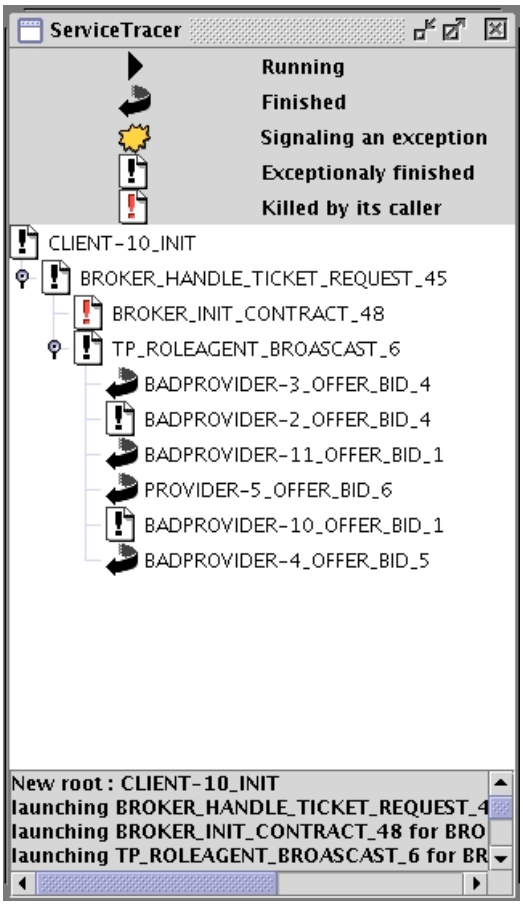
In the travel agency example, concerting the exceptions propagated from *Provider* agents at the role level allows pertinent actions to be performed. If few providers signal exceptions, partial results may still be send to the client as shown in Fig. 10(a). On the contrary, if too much *Provider* agents signal exceptions, an exception is to be propagated to the client in order to notify it of a global problem as shown in Fig. 10(b). This behavior is implemented in the exception resolution function (see Fig. 5) and the handler (see Fig. 11) associated with *TravelProviders_ RoleAgents* role agents.

6.3 Termination

A service which terminates its execution (either standardly or exceptionally) forces the termination of all its pending subservices (see Sect. 4.1). For example, in Fig. 10(b), a concerted exception is propagated up to the root of the service tree. Each service which propagates the concerted exception has exceptionally finished. In particular, the *handle_ ticket_ request* service, during its exceptional termination, forces the termination of its *init_ contract* pending subservice as it becomes useless. The logical tree structure that is formed by the cascaded service requests is thus used upward to manage the propagation of exceptions to dependant services and downward to manage the termination of pending useless requested services.



(a) using partial results



(b) signaling a concerted exception

Figure 10. Examples of concerted exceptions in the travel agency case study

```
public void broadcasthandle (SaGEException exc, BroadcastService bs)
{
   if (exc.getMessage ().equals ("few_bad_providers"))
   {
      bs.setalive (true);
      bs.broadcastavailablereponses ();
   }
   else
   {
      sendMessage (bs.getparentowneraddress (), new SaGEMesage ("exception",
                   exc.getMessage (), bs.getrequestID ()));
      bs.terminate ();
   }
}
```

Figure 11. Handler associated to the *TravelProviders_ RoleAgent* role agents

7 Conclusion and Future Work

In this paper we propose an original exception handling system for MASs. It distinguishes itself from previous work because it does not rely on the use of entities external to agents but fully integrates exception handling mechanisms to the execution model of the agents. It allows in-context, pertinent handlers to be defined that can directly be associated with the services provided by an agent, as part of its behavior. The execution model supports cooperative concurrency and manages the propagation of exceptions between cooperating agents. Moreover, individual exceptions propagated from agents that contribute to a collective activity can be concerted into more pertinent exceptions regarding the management of those global activities. Handlers and exception resolution functions can be associated with different kind of execution model entities (services, agents, roles) in order to support exception handling in different contexts (from the local behavior of agents to the collective activities in roles, through one-to-one collaborations between agents).

We implemented and successfully experimented this model: the experimentation is available as an applet[5].

Various perspectives are considered, such as to extend our EHS in order to be able to resume the execution of a service to some chosen point after the successful treatment of an exception. Another perspective is to transpose SaGE to component-based platforms (such as J2EE/JMS technologies) [26] or to other message-oriented middlewares (WebServices).

8 Acknowledgments

The authors thank Jacques Ferber, creator of the MadKit system [1,16] for his contribution to this work, and for many profitable discussions.

References

1. Ferber, J.: Les systèmes multi-agents, vers une intelligence artificielle distribuée. InterEditions (1995)
2. Szyperski, C.: Component Software: Beyond Object-Oriented Programming. ACM Press and Addison-Wesley, New York, NY (1998)
3. Campbell, R., Randell, B.: Error recovery in asynchronous systems. IEEE Transactions on Software Engineering (SE) **SE-12 number 8** (1986) 811–826
4. Gärtner, F.C.: Fundamentals of fault tolerant distributed computing in asynchronous environments. ACMCS **31** (1999) 1–26
5. Knudsen, J.L.: Fault tolerance and exception handling in beta. In Romanovsky, A., Dony, C., Knudsen, J.L., Tripathi, A., eds.: Advances in Exception Handling Techniques. LNCS (Lecture Notes in Computer Science) 2022, Springer-Verlag (2001)
6. (MADKit) http://www.madkit.org.
7. Goodenough, J.B.: Exception handling: Issues and a proposed notation. Communications of the ACM **18** (1975) 683–696
8. Anonymous: Rationale for the design of the ada programming language. ACM SIGPLAN Notices **14** (1979) 1–139
9. Dony, C.: Exception handling and object-oriented programming : towards a synthesis. ACM SIGPLAN Notices **25** (1990) 322–330 *OOPSLA ECOOP '90 Proceedings*, N. Meyrowitz (editor).
10. Koenig, A.R., Stroustrup, B.: Exception handling for C++. In: Proceedings "C++ at Work" Conference. (1989) 322–330
11. Liskov, B.: Distributed programming in argus. Communications of the ACM **31** (1988) 300–312
12. Meyer, B.: Disciplined exceptions. Technical report tr-ei-22/ex, Interactive Software Engineering, Goleta, CA (1988)
13. Weinreb, D.L.: Signalling and handling conditions. Technical report, Symbolics, Inc., Cambridge, MA (1983)
14. J.Ichbiah, Barnes, J., Héliard, J., Krieg-Brueckner, B., Roubine, O., Wichman, B.: Rationale for the design of the ada programming language. ACM SIGPLAN Notices **14** (1979)

[5] http://www.lgi2p.ema.fr/~fsouchon/sage_applet/sage_applet.html

15. Wooldridge, M., Ciancarini, P.: Agent-oriented software engineering. Handbook of Software Engineering and Knowledge Engineering. World Scientific Publishing Company (1999)
16. Ferber, J., Gutknecht, O.: A meta-model for the analysis and design of organizations in multi-agent systems. In: Third International Conference on Multi-Agent Systems (ICMAS98). (1998) 128–135
17. Klein, M., Rodriguez-Aguilar, J.A.: Using role commitment violation analysis to identify exceptions in open multi-agent systems, ases working paper ases-wp-2000-04 (2000)
18. Klein, M., Dellarocas, C.: Towards a systematic repository of knowledge about managing multi-agent system exceptions, ases working paper ases-wp-2000-01 (2000)
19. Ricordel, P.M., Demazeau, Y.: From analysis to deployment: A multi-agent platform survey. In: Engineering Societies in the Agents World. Volume 1972 of LNAI., Springer-Verlag (2000) 93–105 1st International Workshop (ESAW'00), Berlin (Germany), 21 August 2000, Revised Papers.
20. Klein, M., Dellarocas, C.: Using domain-independent exception handling services to enable robust open multi-agent systems: The case of agent death. Journal for Autonomous Agents and Multi-Agent Systems **7** (2003)
21. Tripathi, A., Miller, R.: Exception handling in agent oriented systems. In Romanovsky, A., Dony, C., Knudsen, J.L., Tripathi, A., eds.: Advances in Exception Handling Techniques. LNCS (Lecture Notes in Computer Science) 2022, Springer-Verlag (2000)
22. Romanovksy, A., Kienzle, J.: Action-oriented exception handling in cooperative and competitive object-oriented systems. In Romanovsky, A., Dony, C., Knudsen, J.L., Tripathi, A., eds.: Advances in Exception Handling Techniques. LNCS (Lecture Notes in Computer Science) 2022, Springer-Verlag (2001) Also available as Technical Report (EPFL-DI No 00/346).
23. Issarny, V.: Concurrent exception handling. In Romanovsky, A., Dony, C., Knudsen, J.L., Tripathi, A., eds.: Advances in Exception Handling Techniques. LNCS (Lecture Notes in Computer Science) 2022, Springer-Verlag (2001)
24. Lacourte, S.: Exceptions in Guide, an object-oriented language for distributed applications. In Springer-Verlag, ed.: ECOOP 91. Number 5-90 in LNCS, Grenoble (France) (1990) 268–287
25. FIPA: FIPA 97 Specification Part 2 : Agent Communication Language. (1997)
26. Souchon, F., Urtado, C., Vauttier, S., Dony, C.: Exception handling in component-based systems: a first study. In: Exception Handling in Object Oriented Systems: towards Emerging Application Areas and New Programming Paradigms Workshop (at ECOOP'03 international conference) proceedings. (2003)