CAMA: Structured Coordination Space and Exception Propagation Mechanism for Mobile Agents

Alexei Iliasov, Alexander Romanovsky

School of Computing Science, University of Newcastle
Newcastle upon Tyne, NE1 7RU, United Kingdom

Abstract. Exception handling has been proven to be the most general fault tolerance technique as it allows effective application-specific recovery. If exception handling is to make programmer's work more productive and less error-prone, however, it requires adequate support from the programming and execution environments. Scoping is a dynamic structuring technique which makes it easier for the developers to deal with the complexity of system execution by narrowing down the context visible for the individual system components. In this work we are specifically interested in scoping that supports error confinement and allows system error recovery to be limited to the area surrounding the error. The approach we propose aims at assisting in rigorous development of structured multi-level fault tolerant agent systems.

1 Introduction

Intrinsic virtues of mobile agents such as mobility, loose coupling and ability to deal with disconnections can make them look promising for structuring large-scale distributed systems. Yet agents have to face all kinds of communication media failures as well as failures of software in their fellow agents and, of course, internal failures. System openness brings even more concerns, such as interoperability, security and trustworthiness. The types of mobile agent system failure can be roughly grouped into the following categories:

1. *failure to deliver service by the hosting environment;*
2. *failure in one of the collaborating agents;*
3. *internal agent failure;*
4. *an environment failure.*

While a similar classification is discussed in [2], in this paper we are introducing different categories of faults in order to focus on the interoperability issues more and to capture in a more practical and detailed way all possible kinds of the environment failure.

Failures of the first category include all types of transient failures, such as disconnection, migration, spawning, inability to deliver messages, etc. Such failures may be caused by changes in the environment, for example those due to

migration, and are better handled by application logic. It has often been said that recovery actions for such situations must be developed at the initial stages of agent design. And, unlike traditional software, mobile agents have, thanks to mobility and code migration, a whole new kind of recovery possibilities.

The second category consists of failures of a very interesting kind. One of the appealing features of mobile agents is dynamic composition. Agents do not have to know what other agents they will cooperate with, and this allows extreme flexibility in agent system design. In open systems, where agents discover their partners dynamically and where each agent has its own interest in cooperation, there must be some mechanism to encourage communication among matching agents and prevent it among incompatible ones. In addition to the means of communication among agents, we also need means for inter-agent exception propagation and cooperative exception handling. We believe that this is essential for a disciplined and fault-tolerant composition of mobile agent systems.

The abnormal situations of the third category are detected inside an agent. All the traditional recovery techniques developed for sequential programming can be used to deal with them. If an agent fails to recover from a failure individually, then there is a need for cooperative exception handling by all the involved agents.

The last category of failures corresponds to exceptional situations in the environment that are beyond the control of a mobile agent. Examples of this are failures of hardware components, administrative restrictions, software bugs in the underlying middleware and in the core components of the environment.

All these failures are typical of the domain of mobile agent software and mobile agents usually cannot anticipate or avoid this kind of malfunctions. In this paper we are focusing on the second category of failures and propose two fault tolerance solutions. The first one is an exception handling technique for coordination space-based mobile agents. The second solution is a scoping mechanism for mobile coordination spaces. In our approach we combine these two solutions in one fault tolerance development method.

Exception handling has been proven [1] to be the most general fault tolerance technique as it allows effective application-specific recovery. If exception handling is to make programmer's work more productive and less error-prone, however, it requires adequate support from the programming and execution environments. Scoping is a dynamic structuring technique which makes it easier for the developers to deal with complexity of system execution by narrowing down the context visible for the individual system components. In this work we are specifically interested in scoping that supports error confinement and allows error recovery to be limited to the area surrounding the error.

2 Related Work

MobileSpaces [12] is a middleware with a hierarchical organisation of agents. The notion of agent nesting and the approach proposed to migration is similar to those used in the Ambient Calculus [13] algebra. An agent in MobileSpaces communicates only with its parents or descendants (nested agents). Whole branches

of the agent tree can migrate, changing their parent nodes. This approach presents quite a flexible form of isolation.

In Mole [9] inter-agent communication is based on the publish/subscribe model called Object Management Group (OMG). OMG introduces channels through which events can be propagated among agents. Channels can be created during run-time and it is the creating agent who decides to whom pass the channel reference. This encourages closed group work among several agents. However event channel is not an isolation mechanism since it can pass events to other channels and also receive external events interesting to the channel subscribers.

Paper [8] discusses an extension of the publish/subscribe scheme with the scope concept. Scopes can be nested and they regulate event propagation and publishing. Agents can create, join and leave scopes dynamically. The purpose of scopes in this model is to limit visibility of published events to a subset of agents from the same tree of scopes (all scopes in the system form a forest). Another important implication of the scope notion is the introduction of the administrator role. Administrator is a utility agent that controls event flow inside a scope and across its boundaries according to the rules statically defined for the scope.

A different approach is taken in ActorSpace [10, 11], where communication space is structured using actorSpace - an abstract agent container. Special entities called managers may control visibility of agents and actorSpaces with respect to some other actorSpace. Each agent has a set of patterns describing its interests. There are three basic ways of sending a message: using a pattern to non-deterministically describe a destination agent, using a unique agent name and a pattern-based broadcast which delivers messages to all the agents satisfying the specified pattern. In addition it possible to create arbitrary complex visibility structures by placing a reference to an actorSpace in another actorSpace.

Coordination with Scopes [14] discusses a scoping mechanism for Linda tuple space built in way similar to ActorSpace. However scope here is not a container but a viewpoint of an agent on otherwise flat tuple space. The most interesting aspect is possibility of dynamically create new scopes by using several predefined operations on already known scopes forming a kind of scope algebra. In addition to the obvious joining and nesting operations, scopes can be also intersected and subtracted. This gives extreme flexibility in structuring tuple space and adapting it to an agent needs. A dedicated scope initially known to all agent is used to exchange scope names.

3 Coordination with Scopes

3.1 CAMA Model

The CAMA (context-aware mobile agents) system consists of a set of *locations*. Active entities of the system are *agents*. CAMA agent (further agent) is a piece of software that conforms to some formal *specification*. Each agent is executed on its own *platform*. Platform provides execution environment and interface to

the location middleware. Agents communicate through the special construct of coordination space called *scope*. An agent can cooperate only with agents participating in the same set of scopes. Agents can logically and physically migrate from a location to a location. Migration from a platform to a platform is also possible using logical mobility. An agent is built on the base of one or more *roles*. Role is a formal functionality specification and composition of specifications of all the roles forms the specification of an agent. A role is the result of the decomposition of an abstract *scope model* and a *run-time scope* is an instantiation of such abstract model. After this point we will use term *scope* to refer to a run-time scope in coordination space. More details on building formal specification of roles using the B Method and general description of the CAMA system can be found in [3].

3.2 Scoping mechanism

After analysing a number of existing approaches to introducing structuring of mobile agent communication (see Section 2.1) we have found that the best way to do it for the purpose of dealing with complexity of the system behaviour during rigorous system development, and, in particular, with supporting behaviour and information hiding for fault tolerance is to *structure agent activity* (dynamic behaviour). This automatically introduces communication structuring however with a much cleaner semantics and a number of other benefits discussed below.

Structuring activity means arranging agents in groups according to their intentions and *afterwards* configuring the means of their communication to adapt to the requirements of the agent group. Reconfigurations happen automatically thus allowing agents (and developers) to focus solely on collaboration with other agents. The distinctive features of this approach are

- *higher-level abstraction of communication structuring;*
- *impossibility to create incorrect, malfunctioning or cyclic structures;*
- *strong relationship with interoperability and exception handling;*
- *simple semantics facilitating formal development.*

In a very basic view scope is a dynamic data container. It provides an *isolated* coordination space for *compatible* agents by restricting visibility of tuples contained in a scope to the participants of the scope. Concept of compatibility is based on the concepts of role and scope. A set of agent is compatible if there is a composition of their roles that forms an instance of an abstract scope model.

Agents may issue a request for a scope creation and, at some point, when all the precondition are satisfied, the scope is atomically instantiated by a hosting location. Scope has a number of attributes divided into categories of scope *requirements* and scope *state*. Scope requirements essentially define type of a scope, or, in other words, kind of activity supported by the scope. Scope requirements are derived from a formal model of a scope activity an together with agent roles form an instance of the abstract scope model. State attributes characterise a unique scope instance.

Requirements	State
- *list of roles* - *restriction on roles*	- *currently enrolled roles* - *owner* - *name*

In addition to the attributes, scope contains *data*, that in case of coordination space are *tuples*. Along with data there may be *subscopes* to match *nested activities* that may happen inside of a scope.

Restrictions on roles dictate how many agent roles there can be for any given role of a scope. Requirements are defined by two numbers - a minimum required number of agents for a given role and a maximum allowed number of agents for a given role. A scope state tracks the number of currently taken roles and determines whether the scope can be used for agent collaboration or not.

R_1	R_1^{min}	R_1^{max}
R_2	R_2^{min}	R_2^{max}
...		
R_k	R_k^{min}	R_k^{max}

$R_i^{min} \leq N_{R_i} \leq R_i^{max}$	(taken roles)
n	(scope name)
A	(owner)

Fig. 1. Scope requirements (left). Scope state (right)

In addition to obvious $R_i^{min} \leq R_i^{max}$ we also require that $R_i^{max} > 0$.

There are three important states of a scope. Their summary is given on Table 2. A scope in the *pending* state does not allow agents to communicate because

State name	Definition
pending	$\exists r \cdot (r \in R \wedge N_r < r^{min})$
expanding	$\forall r \cdot (r \in R \Rightarrow N_r \geq r^{min}) \wedge \exists r \cdot (r \in R \wedge N_r < r^{max})$
closed	$\forall r \cdot (r \in R \Rightarrow N_r = r^{max})$

Fig. 2. Three important states of a scope

there are some essential roles missing. When all the required roles are taken the scope becomes *expanding* or *closed*. In the expanding state a scope supports communication among agents while still allowing other agents to join the scope. In the closed state there are no free roles and no additional agent may join the scope.

Some scope configurations present interesting cases. A scope with zero required number of agents for all the roles is called *blackboard*. It persists even

without any participating agents and all the contained data also persist. With this scope type agents do not have to wait for any other agents to communicate, they may put some information into a blackboard scope and leave. Note, that there is an important difference between a blackboard scope and a generic tuple space. For a blackboard scope only agents implementing the roles specified by the blackboard scope requirements may enter and put or read any data whilst in a tuple space anyone can always put and read any tuples. *Container* is a scope

Scope class	Definition
blackboard	$\forall r \cdot (r \in R \wedge r^{min} = 0)$
container	$card(R) = 1 \wedge (r \in R \wedge r^{min} = r^{max} = 1)$
bag	$card(R) = 1 \wedge (r \in R \wedge r^{min} = 0 \wedge r^{max} = 1)$
unrestricted	$\exists r \cdot (r \in R \Rightarrow r^{max} = \infty)$

Fig. 3. Some interesting classes of scopes

with a single role for which only single agent is allowed and required. This is an important case since such kind of a scope can act as a private and protected data container of an agent. A variant of container scope that can survive change of owners without losing all the contents is called *bag*. Bags can be used to privately pass some bulk data between two agents.

Unrestricted scope permits an unlimited number of agents for one or more of its roles. It can be used for client-server models when there are no restrictions on number of clients.

In global view scopes form a tree. Due to the specifics of our approach the tree is mostly shallow and wide since the depth is determined by the nesting level of actions that is usually not large. All the high-level scopes are united by the dedicated scope λ. Any scopes other than λ are subscopes of λ. Scope λ has two predefined roles: role λ_A of agent requesting services from location and and role λ_L of location. Functionality of role λ_A is arbitrary and defined by agent developers. However there is a fixed set of operations called λ_L^0 that must be included into implementation of each λ_L. λ_L^0 operations are described below.

Since the CAMA system allows agent to communicate in several locations at the same time we include location name to disambiguate reference to a scope. Moreover scope names have to be unique only inside the containing scope. Thus the full name of a scope includes the names of all the containing scopes. We are omitting name of λ scope for convenience of notation. Sometimes we have to explicitly state in what location a scope is contained and what are its parents (the containing scopes). In this case location name is the initial part of a name, then follows the outermost containing scope and so on up to the name of the scope we deal with.

λ_L^0 operations:

- `engage(id)` - issue a new location-wide unique and unforgeable name for agent `id`. This name as agent identifier in all other role operations.
- `disengage(a)` - issued name `a` becomes invalid.
- `create(a, n, R)@s` (`n` $\notin$ `l.s`) - creates a new sub-scope of scope `s` with the name `n` and given scope requirements `R`. The created scope becomes a private scope of agent `n`.
- `destroy(a, n)@l.s` (`n` $\in$ `l.s` $\wedge$ `a` *is owner of* `l.s.n`) - destroys sub-scope with the name `n` contained in the scope `s`. This operation always succeeds if the requesting agent is the owner of the scope. If the scope is not in the pending state then all the scope participants receive `EDestroy` exception as the notification of the scope closure. This procedure is executed recursively for all the subscopes contained in the scope.
- `join(a, n, r)@s` (`n` $\in$ `l.s` $\wedge$ `r` $\in$ `n` $\wedge$ `n` *is pending or expanding*) - adds agent `a` into scope `n` contained in `l.s` with roles `r`. Succeeds if scope `l.s.n` exists and it is possible to take the specified roles in the scope. This operation may cause state change of the scope.
- `leave(a, n, r)@s` (`a` *is in* `l.s.n` *with role(s)* `r`) - removes agent roles `r` from scope `l.s.n`. The calling agent must be already participating in the scope. This operation may also change the state of the scope.
- `put(a, n)@s`: advertises scope `n` contained in scope `s` thus making it a public scope. A public scope is visible and accessible by other agents. contained in scope `l.s` and supporting role(s) `r`.
- `get(a, r)@s`: enquires names of the scopes contained in scope `l.s` and supporting role(s) `r`.
- `handshake(a, t)@s`: allows agents to safely exchange their names.

3.3 Naming issues

In the following discussion we discuss certain assumptions on how names of various resources are used and passed between agents. One essential requirement is that a scope name can be known to an agent only if the agent joins the parent of the scope. A scope name passed as a message between two agents may violate this rule so we have to take special care of the names used by agents. However we still allow agents to learn names of other agents, scopes, locations and traps (discussed further below) through communication with other agents. To make it impossible for an agent to use incorrect names and pass names without permission to do so to third parties we do not use any absolute names or references. Instead the naming mechanism is based on *tickets* issued by location. Whenever an agent needs to have a name for some resource (e.g. a new scope created by its request) a location generates a new structure consisting of the agent name and the resource reference. This structure is associated with a random ticket number which is passed to the agent. The ticket issuing location is the only entity that can decode ticket numbers into resource references. Location has the full control over name passing and prevents issuing and usage of incorrect names. Note that a legitimately owned name may become invalid if a resource is destroyed or the agent no longer has rights to access it. Assuming that agent

names are unforgeable (we need some additional scheme to ensure agent names validity) tickets numbers can be exposed without any risk for the owner. When an agent wants to share or pass a resource to another agent it requests the location to issue additional ticket for another agent. The major advantage of this scheme is that the agent requesting a new ticket still has, being the owner of the ticket, the full control over the resource access which this new ticket allows. At any moment it can send a request to the location to invalidate the ticket which will have an immediate effect on the ticket user. In addition an owner of a resource can add other agents to the list of owners and remove itself from the list. When an agent becomes a resource owner it can control resource usage and issue new tickets.

This procedure cannot be used to exchange agent names since to do that agents would have to already know the names of each other. For this purpose we introduce the `handshake` operation which implements secure and atomic exchange of names within a group of agents. Each agent receives names of all its peers made specifically for the agent. Each name is a ticket number usable only by the agent and referring to a name of another agent. The second argument of `handshake` operation is the list of tuples received from fellow agents within one scope. Location knows how to identify agents from tuples they produce (see [5]). `handshake` operation must be executed symmetrically by all the agents for the same list of agents (although tuples in the list may be different for each agent). The operation fails for *all* agents if there is at least one not executing the handshake at all or executing it with a different set of agents. Unsuccessful handshakes are unblocked after a timeout determined by the location.

4 Exception Propagation via Coordination Space

Previously we have developed a mechanism for propagating exceptions among independent, anonymous and asynchronously-communicating agents. In this section we give only a general and brief overview of the work to allow us to discuss problems of exception propagation between scopes, introduced in section 5. The detailed discussion of the mechanism and its experimental implementation for the Lime mobile agents system [7] can be found in [5].

The mechanism of the exception propagation is complimentary to the application-level exception handling. All the recovery actions are implemented by application-specific handlers. The ultimate task of the mechanism is to transfer exceptions between agents in reliable and secure way. However the enormous freedom of behaviour in agent-based systems makes it impossible to guarantee reliable exception propagation in a general case. Although we can clearly identify the situations when exceptions may be lost or not delivered within a predictable time period. If an application requires cooperative exception handling at certain moments then for that time, agents behaviour must be constrained in some way to disallow any unexpected migrations or disconnections.

There are three basic operations available to agents to receive and send inter-agent exceptions. They are supplementary to the application-level mechanism and their functionality do not overlap.

throw	**wait**	**check**

The first operation, `throw`, propagates an exception to an agent or a scope. Important requirements is that the sending agent prior to sending an exception must have got a message from the destination agent and they both must be in the same scope. These two variants of the operation has the following form:

- `throw(m, e)` - throws exception `e` as reaction to message `m`. The message is used to trace the producer and to deliver the exception to it. The operation fails if the destination agent has already left the scope in which the message was produced.
- `throw(s, e)` - throws exception `e` to all the participants of scope `s`.

The crucial requirement to the propagation mechanism is to preserve all the essential properties of agent systems such as agents anonymity, dynamicity and openness. The exception propagation mechanism does not violate the concept of anonymity since we prevent disclosure of agent names at any stage of the propagation process. Note that the `throw` operation does not deal with names or addresses of agents. Moreover we guarantee that our propagation method cannot be used to learn names of other agents.

Also the mechanism itself does not restrict agent activities in any way. Though agent dynamicity and reliability of exception propagation are conflicting concepts we believe that it is the developers who must take the final decision to favour either of them. Notion of openness is the key for building large-scale agent systems. Proper exception handling was proved to be crucial for consistent and reliable component composition. It is even more so for mobile agent systems where composition is dynamic and parts of the system are developed independently. To support large-scale composition of exceptions-enhanced agents we are going to elaborate a formal step-wise development procedure.

Two other operations, `check` and `wait` are used to explicitly poll and wait for inter-agent exceptions.

- `check` - if there are any exception pending for the calling agents raises exception `E(e)` which is a local envelop for the oldest pending exception.
- `wait` - waits until any inter-agent exception appears for the agent and raises it in the same way as the previous operation.

We also redefine semantics of blocking Linda operations so that unblock whenever a coordination space exception appears and throw this exception inside of the agent.

4.1 Traps Mechanism

The propagation procedure expressed only with the primitives above would be too restrictive and inflexible for mobile agent systems. To control the propagation process in a way that accounts for various agent behaviour scenarios we

introduce a notion of *trap*. A trap is a set of rules created by agents that controls exception propagation and exists independently of the creating agent. Traps are stored and manipulated by a location that provides the coordination space. A trap is essentially a list of rules that chose reaction for a coordination space exception. It can be represented as a CASE construct where rules are associated with exceptions (see Figure 4). Exception matching and comparison are non-trivial issues usually dictated by the language of choice.

```
case e is
   when E1 => op1
   when E2 => op2
   ...
   when En => opn
   when others => abort
end case
```

Fig. 4. Trap is a CASE-style construct

A trap can be enabled when there is an incoming message for the agent that created the trap. Agent may have several traps and traps are automatically organised into an hierarchical structure. When an exception appears, the most recently added trap is activated. If the trap fails to find a matching rule for the exception, the exception is propagated to the second most-recent trap and so on. Agents can dynamically create, add and remove traps. The following operations are used to express trap structure:

- `deliver` - delivers the exception to the destination agent. The exception is stored until the destination agent is ready to react to it or the containing scope is destroyed;
- `relay(t)` - propagates the exception to a trap `t` which may be a trap of another agent. Name of a trap can be only learnt through negotiations with the trap owner. Owner of the trap becomes the destination the propagated exception;
- `abort` - leaves the current trap and transfers control to the next trap in the hierarchy.
- `if` *(condition)* `then` *ac* - action *ac* is applied conditionally;
- . *(concatenation)* - forms a new action by concatenation of two other actions.

The `deliver` operation was designed to be able to tolerate agent migration and connectivity fluctuations. It introduces some level of asynchrony and makes the whole exception propagation scheme more suitable to the asynchronous communication style of coordination space. The `relay` operation is a tool for building linked trap structures supporting a disciplined cooperative exception handling. Discussions and examples related to this approach can be found in [5].

Preconditions for the `if` operation are formed from the following primitives:

- `local` - holds if the owner of the trap is joined to the current scope
- `local(a)` - holds if agent `a` is joined to the current scope
- `tuple(t)` - holds if there is a tuple matching template `t`
- $\neg$, $\vee$, $\wedge$ - logical operations that can be used on the predicates above

Rule preconditions and concatenation provide a very expressive mechanism that may form traps for many interesting and useful scenarios. For example, a rule in a trap could make multiple deliveries to involve several agents, or, depending on the locality of the trap owner, another agent or even a trap in a different location.

5 Exception propagation through scope boundaries

The exception propagation mechanism described above works well within the boundaries of a scope. However in many cases a scope corresponds not to a completely isolated activity but rather forms a part of a more general activity of the containing scope. In such a case a failure of a scope may disrupt activity of the containing scope and require agents of the containing scope to execute some recovery actions. In this work our intention is to introduce exception propagation through scope boundaries in a way that is smoothly integrated with the concepts of the scoping and exception propagation mechanisms discussed above.

It is very natural to try and take the advantage of the scope nesting mechanism to build a scope-based exception recovery. However the specifics of the mobile agents and the scoping mechanism bring unexpected complications. First of all, scope nesting does not necessarily correspond to a logical nesting of scope activities and this presents problems in interpreting exceptions propagated from inner scopes. Another complication is that different kind of scopes - nested, at the same level or even unrelated and from different locations, can be linked by a common global state of an agent. Hence, in addition to the hierarchy of scopes introduced by the scoping mechanism, we have to take into consideration relations between scopes introduced by agents.

Currently we are looking into possible realisations of the propagation mechanism for the CAMA system. Below we briefly present several promising approaches. They can be classified into the two categories - the first one looks at the problem from the viewpoint of a failed scope and the second one discusses recovery schemes for a containing scope.

Case 1. Throwing an external exception. One possible solution is to allow agents to throw an external exception which results in the immediate termination of the failed scope. When this happens the scope closes and all other scope participants get a notification of the scope failure.

Case 2. Common root trap tree. Agents create a common root trap tree to propagate exceptions in a disciplined manner outside a scope. Through a negotiation process they exchange trap names and put them together in a common trap tree structure. For example, this tree may be initially created by one of the agents and then updated by others. An exception indicating the scope failure (as classified by the traps) is propagated through the chain of traps and finally arrives to some or all of the agents. If it during this propagation it arrives to the root of the common tree, the scope is terminated with an external exception. This solution is more general and flexible than the first one because different recovery scenarios can be built for different types of exceptions and agent groups.

Case 3. Internal propagation. Taking into account the fact that each agent participating in a nested scope is also present (though does not necessarily active) in the containing scope we can propagate exceptions through the internal state of an agent and, if required, trigger recovery actions in the containing scope. In this case after an unsuccessful cooperative recovery in the failed (nested) scope each participant throws an exception in some of its active scopes. Exception propagation here is fully controlled by an agent and does not necessarily relate to any existing scopes structure. We believe that offering such flexibility may be dangerous and, besides, it is becoming very hard to analyse systems formally. However some situations may require such propagation style. For instance, when an agent acts concurrently in two scopes and these activities are interrelated (say it buys something in one scope and sells it in another). An exception thrown in the scope where the agent sells may require actions in the scope where it buys (but never vice versa as there is one-direction information flow between these scopes).

Another part of the problem is developing a recovery scheme for a containing scope. An important point to note here is that the activities of a containing and a nested scopes in a general case are asynchronous. Some effort must be taken to put agents in the containing scope into some correct state suitable for recovery actions caused by the failed sub-scope. We discuss here two possible solutions.

Case 1. Throwing to everyone. Whenever an exception from a nested scope appears it is thrown to each participant of the containing scope. This triggers normal trap mechanism which involves all the usual recovery procedures that would take place in a case of a local exception. In other words an external exception from a sub-scope appears as a new local exception for all the agents although it may be distinguished as an sub-scope exception by its type.

Case 2. Throwing to the failed scope participants. According to the previous approach, whenever a sub-scope exception is thrown each agent of the containing scope may be interrupted and involved in handling the exception. However we can exclude those agents that are not associated with the failed contained scope from the recovery initial recovery actions because their involvement may be superfluous in the situations when recovery can be done by the nested scope participants. In this case we still must ensure that the exception is propagated

to all the agents of the containing scope if that group of agents fails to recover themselves.

6 Acknowledgements

This research has been supported by the EC IST grant 511599 (RODIN).

References

1. F. Cristian. Exception handling and tolerance of software faults. In M. Lyu, editor, Software Fault Tolerance, pages 81107. Wiley, 1995.
2. Di Marzo, G. and Romanovsky, A. *Designing Fault-Tolerant Mobile Systems*. In Proceedings of the International Workshop on Scientific Engineering for Distributed Java Applications (FIDJI 2002), Luxembourg-Kirchberg, Luxembourg, 28-29 November 2002 Guelfi, N., Astesiano, E. and Reggio, G. (eds.). LNCS 2604 pp.185-201. Springer-Verlag 2003.
3. A. Iliasov, L. Laibinis, A. Romanovsky, E. Troubitsyna. *Towards Formal Development of Mobile Location-based Systems*. To be presented at Workshop on Rigorous Engineering of Fault Tolerant Systems, 19th July 2005, at FME 2005. Newcastle upon Tyne, UK.
4. J.-R. Abrial. The B-Book. Cambridge Univ. Press, 1996.
5. A. Iliasov, A. Romanovsky. *Exception Handling in Coordination-based Mobile Environments*. In Proc of the 29th Annual International Computer Software and Applications Conference Edinburgh, Scotland, July 26-28, 2005. IEEE CS Press. 2005.
6. *The Mobile Agent List*. reinsburgstrasse.dyndns.org/mal/preview/preview.html.
7. G. P. Picco, A. L. Murphy, G.-C. Roman. *Lime: Linda Meets Mobility*. Proc of the 21st Int. Conference on Software Engineering (ICSE'99), Los Angeles (USA), May 1999.
8. L. Fiege, M. Mezini, G. Muhl, A. P. Buchmann. *Engineering Event-Based Systems with Scopes*. In Proc. of ECOOP 2002, pp.309-333. 2002.
9. J. Baumann, F. Hohl, K. Rothermel, M. Straßer. *Mole - Concepts of a Mobile Agent System*. World Wide Web Journal. 1(3), pp.123-137. 1998.
10. G. Agha and C. J. Callsen. *ActorSpace: An Open Distributed Programming Paradigm*.Proceedings 4th ACM Conference on Principles and Practice of Parallel Programming, ACM SIGPLAN Notices, pp.23-323, 1993.
11. C. J. Callsen and G. Agha. *Open Heterogeneous Computing in ActorSpace*. Journal of Parallel and Distributed Computing, vol. 21,3, pp.289-300, 1994.
12. I. Satoh. *Mobile agent-based compound documents*. Proc. of the ACM Symposium on Document Engineering 2001, pp.76-84. 2001. I. Satoh. MobileSpaces: A Framework for Building Adaptive Distributed Applications using a Hierarchical Mobile Agent System. Proc. of the ICDCS 2000, pp.161-168. 2000.
13. L. Cardelli and A. D. Gordon. Mobile ambients. In Maurice Nivat, editor, Proc. FOSSACS'98, volume 1378 of Lecture Notes in Computer Science, pages 140–155. Springer-Verlag, 1998. Also appear in Theoretical Computer Science, 240(1), pp.177-213, June 6, 2000.
14. I. Merrick, A. Wood. *Coordination with Scopes*. Proceedings of the 2000 ACM symposium on Applied Computing, Como, Italy, pp.210-217, 2000.