Formal development of cooperative exception handling for mobile agent systems

Linas Laibinis
Åbo Akademi University
Finland
llaibini@abo.fi

Elena Troubitsyna
Åbo Akademi University
Finland
etroubit@abo.fi

Alexei Iliasov
Newcastle University
England
alexei.iliasov@ncl.ac.uk

Alexander Romanovsky
Newcastle University
England
alexander.romanovsky@ncl.ac.uk

ABSTRACT

Mobile agent systems often require sophisticated cooperation and coordination during error detection and recovery. In this paper we propose novel fault tolerance mechanisms that support co-operative exception handling in such systems. The paper demonstrates how mechanisms like these can be formally developed and analysed. We start with identifying the typical modes of failures in agents and analysing possible failure and recovery scenarios in mobile systems. Stepwise refinement is used as our formal framework for top-down development and verification. Using the framework we formally verify the essential model properties, such as interoperability, local and global state consistency and termination of error recovery. Our approach provides developers with formal generic patterns for incorporating fault-tolerance mechanisms into mobile agent systems. We also demonstrate how the results of our formal development can be instantiated and reused in developing real-world agent software.

1. INTRODUCTION

High complexity of software is one of the major obstacles to developing dependable systems. The agent paradigm aims to address complexity by explicitly separating inter-agent communication from internal computations conducted by agents. It has also been very successfully applied in the area of mobile computing. The concept of an *autonomous*, self-contained software unit, which can dynamically establish collaboration with other similar units, fits the idea of mobile computing perfectly. The agent paradigm has evolved further to address the issues specific to mobile computing, such as mobility, openness and anonymity [22]. Supporting these three characteristics is the key to building large multi-agent applications. A mobile agent is a software components that is able to change its physical or logical location in a search for resources or other agents. Openness allows a number of agents from different administration domains to form multi-agent applications. Anonymity is collateral to openness and equally essential for decoupling, security and scalability. Agents are autonomous because they participate in a multi-agent application to achieve their own goals, so they do not have to follow protocols enforced by the third parties.

Exception handling has proven to be the most effective and general mechanism for system recovery as it supports application-specific recovery ensuring that the system is moved to a correct state (as opposed to backward error recovery moving the system state back or masking recovery requiring substantial redundancy and diversity) [6, 3]. Agent communication and collaboration forms the core functionality of any agent system. Dealing with complex erroneous conditions in such systems typically requires recovery involving several agents to ensure both system and agent consistency, and efficient recovery healing the whole erroneous state, usually spreading over several interacting agents. Unfortunately, only few multi-agent systems provides general support for cooperative recovery based on exception handling. Typically in case of an error, an agent is left to conduct its local recovery using its own resources. Thus any error from which an agent is unable to recover leads to agent termination. There have been several solutions for introducing recovery based on exception handlers attached to applications, agents or services [21, 17, 15]. Such handlers monitor system states and, in case of an error, execute some recovery actions. It is hard to implement really complex recovery actions with such an approach, since the handlers, being external entities, cannot manipulate the agent state or behaviour, and since agents themselves are not designed for (cooperative) recovery.

Moreover, there have been few general mechanisms for exception propagation in multi-agent systems. Paper [20] has an excellent discussion of the motivations and the design of a mechanism for inter-agent exception propagation. Paper [10] discusses an exception propagation mechanism designed specifically for mobile, asynchronous and anonymous agents. However this approach does not address many problems of cooperative recovery.

The common problem of the existing exception handling mechanisms for agents is that they do not exhibit all important characteristics of mobile agent systems. Only some of the presented mechanisms successfully address the issues of agent asynchrony, anonymity and mobility. However none of them preserve agent autonomy.

In this paper we try to go beyond the conventional approach to cooperative recovery and introduce a mechanism that addresses the needs of agents and, at the same time, we attempt to preserve all the mentioned features of multi-agent systems. We start by analysing the needs of individual agents and proceed with deriving a mechanism for cooperative recovery.

The rest of the paper is organised as follows. Section 2 introduces the agent-centric approach to cooperative recovery. Section 3 presents the middleware and the architecture on top of which the discussed mechanism is developed. Section 4 demonstrates the difference between the requirements to cooperative recovery in multi-agent and distributed systems. Section 5 presents our proposition: the cooperative recovery mechanism for multi-agent systems. Finally, Section 6 discusses our experience of formal modelling and verification of the proposed mechanism. The results of formal modelling in the form of the developed B specifications are available from [13].

2. AGENT-CENTRIC RECOVERY

Our approach is based on agent-centric recovery. Instead of reasoning about a global system recovery we focus on recovery of an individual agent participating in a multi-agent application. We assume that

- agents act egoistically and participate in recovery only for their own benefits;
- time is one of the most valuable resources and an agent should be unwilling to spare it without a good reason.

We believe that these properties form an important part of agent autonomy. A truly autonomous agent *cannot be forced* to do anything, for example, take part in a cooperative recovery. When an agent does participate in recovery its first priority is to recover itself, the secondary priority is to recover the environment, and the least priority is recovery of other agents. These principles are the basis of our mechanism.

To provide a rationale for our approach, next we discuss a number of typical situations.

Sometimes an agent might be willing to avoid any recovery if participation in the recovery is against its needs. Just to give a simple example, let us consider a buyer agent that comes to a shop scope/location and sees another buyer agent and a seller agent recovering from an error. It might be more efficient for it to look for another shop than spend time in the recovery process.

Another example is a malicious or malfunctioning buyer agent. It is in the interest of a seller agent to ignore or block out the annoying buyer rather than trying to investigate problems of the broken agent, thus suspending the normal shop activity.

When an agent detects an error, it should be aware that some of its peers might want to avoid cooperative recovery. Thus an agent must be conservative about initiating cooperative recovery, although in many situations cooperative recovery is beneficial for all involved agents. Going back to the shop example, a seller might be interested in helping a buyer to recover if they are already in the middle of a deal.

The conventional way of doing coordinated recovery is not only inefficient but can be also counterproductive. Forcing agents to always participate in coordinated recovery may distract them from their primary activity and reduce freedom of action. Sometimes the best recovery is not to do any recovery and retry elsewhere with other agents. In the agent systems, unlike parallel and distributed systems, there is no notion of the global system state, and no single agent should be critical to the survival of the whole agent infrastructure. If an agent fails, it can be a problem for its immediate peers and a concern for its owner. But for all other agents the failure might (and often should) go unnoticed. Needless to say, the size and complexity of agent systems make it impossible to involve all agents in recovery.

3. CAMA

The Cama *(Context-Aware Mobile Agents)* system ([11, 9]) provides the middleware to support mobile agent interactions. Any Cama application consists of a set of *locations*. A location is a container for *scopes*. A scope provides a coordination space for interaction of compatible agents. An agent is a piece of software that conforms to some formal *specification*.

To deal with various functionalities provided by an individual agent, Cama introduces the concept of agent role as a finer unit of code structuring. A role is an important part of the scoping mechanism because it supports dynamic composition of multi-agent applications by allowing agents playing several scope roles to come together to cooperate. Moreover, the roles and the scopes are used to ensure agent interoperability and isolation. More details on Cama can be found in [14].

Agent communication in Cama is based on the Linda [7] paradigm. Linda coordination primitives allow processes to put *tuples* in and get them out of a tuple space shared by these processes. A tuple is a vector of typed data values. Linda is sufficiently expressive, e.g., simulation of semaphores or mutexes is straightforward using the Linda primitives.

Context-awareness is one of the key features of mobile systems. Agent *context* represents all information from an agent environment which is relevant to its activity [19]. The context of an agent in Cama includes:

- locations in which an agent is engaged,
- the names, types and states of all the visible scopes in all the engaged locations,
- the state of scopes in which the agent is currently participating.

Typically an agent plays different roles in different scopes. A set of agents with different roles constitutes a multi-agent application. A simple example is a client-server system, where a distributed application is constructed when agents playing two roles meet and collaborate. A server agent can provide the same service in many similar scopes. In addition it can also implement a client functionality and, hence, act as a client in some other scope.

We use the reactive model [18, 5] for building agent roles. A reaction is an action associated with an event. The reaction mechanism allows an agent to coordinate in a pro-active manner. In CAMA, a reaction is triggered by a matching tuple (message) in a tuple space. An agent role is constructed from a set of reactions (see Figure 1).

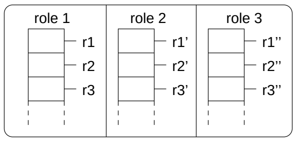


Figure 1: Agent structure

An agent role consists of a number of reactions. Reactions can be executed concurrently, and can disable and enable other reactions. They can also produce new messages (tuples) which will trigger reactions in other agents.

Agents may create and join a scope in the hosting location. A scope is a dynamic data container, which provides an *isolated* coordination space for *compatible* agents by restricting visibility of tuples contained in the scope only to the agents involved in it. In the following we assume that each agent plays only one role in a scope. A set of agents is compatible in a scope if the scope supports their roles.

A scope has a number of attributes divided into scope *requirements* and scope *state*. The scope requirements essentially define the type of a scope, or, in other words, the kind of activities supported by it. The scope requirements are derived from a formal model of a the scope activities. Restrictions on roles dictate how many agent roles there can be for any given role of a scope. The scope requirements also define a *scope liveness condition* - a condition describing an operational scope.

Openness plays an essential role in the CAMA abstractions and the middleware. In our understanding, openness is the ability to create distributed applications composed of agents developed independently at different locations. For this, we provide powerful abstractions that help to dynamically compose applications from individual agents, agent isolation mechanism (which also contributes to security and error confinement), and service discovery based on the scoping mechanism. A scope structures the activity of several agents in a specific location by dynamically encapsulating roles of these agents. A scope also provides an isolation of several communicating agents, thus structuring the communication space.

During our work on CAMA and on various agent applications we have formulated a set of requirements for the exception handling support which suits best the development of complex mobile coordination-based agent systems. In this paper we describe this exception handling mechanism and show its formal development.

4. TRADITIONAL EXCEPTION HANDLING IN DISTRIBUTED SYSTEMS

Exception handling has been widely used as the major mechanism for structuring complex software systems and for implementing their fault tolerance [6]. The fundamental work in [4] introduces the main principles of recursive structuring and exception handling for the distributed applications consisting of a number of cooperating processes. This structuring is based on the concept of an atomic action, i.e. an activity in which several processes participate to cooperate. Because there is no information crossing the action boundaries, these actions serve as the units of error confinement and error recovery. The processes cooperate within an action during its normal and abnormal execution (i.e. recovery). The recovery is implemented as cooperative exception handling which involves all action participants. The main reasons for this are that the action is the damage area containing error and that to ensure consistency of recovery we, generally speaking, need to involve the whole set of cooperating processes. The action can complete either successfully (with or without internal recovery) or unsuccessfully, in which case an exception is propagated to a containing action which takes all responsibility for the following recovery. When an exception is raised by an action participant, it gets propagated to all participants which independently initiate handling. In the situations when several processes concurrently raise exceptions, a resolution tree is used to resolve them and to find an exception handler which will ensure recovery from all those exceptions. A resolution tree is a structure developed during action design, which imposes a partial order on the action exceptions in such a way that a handler for a higher-level exception can handle any of the exceptions below in the tree.

In the following work [23] a deeper analysis of the resolution mechanism was presented and a distributed protocol implementing this scheme in a message-passing distributed systems was introduced. In particular, the paper gave a number of evidence supporting the need for exception resolution in distributed systems.

Multi-agent systems clearly need cooperative exception handling as they are built out of a number of cooperating agents. This is one of the main intentions behind introducing agent system structuring using scopes in CAMA. (Note that in the following we refer to agent structuring using scopes to contrast it with process structuring using actions). Moreover, it is clear that in the agent systems several exceptions can happen at the same time, so the exception resolution should be supported. Unfortunately, the existing solutions ([4], [23]) are not directly applicable for the mobile agent systems as they do not address openness, dynamicity, anonymity and asynchronous coordination. The main reasons for this are that these structuring, exception handling and exception resolution mechanisms are developed for closed, self-contained systems, operating under rather strong assumptions. In particular,

- in the traditional approaches, processes always stop after an exception is raised, and they continue after that with a cooperative handling of one exception, so that the processes cannot receive any new exceptions during exception resolution. However, mobile agents may be unwilling or unable to stop and wait to be synchronised for cooperative recovery inside a scope. Thus we have to consider a situation when a new exception is raised while an agent is involved in exception resolution.

- these mechanisms typically use global ordering for the messages sent within a scope (e.g., an action). It is forbiddingly expensive to provide a global message ordering mechanism in the mobile agent systems. Thus agents should be able to see exceptions coming in a different order.

- these mechanisms enforce process synchronisation on action entry and exit. In particular, they force all cooperating processes to agree on the action outcome. We believe that agents cannot be constrained in such ways.

- as opposed to process recovery, agent recovery needs to be context specific. In particular, this means that resolution should be context specific, and the idea of using statically defined resolution trees is rather restrictive for the agent systems.

- agents can become disconnected or can disappear during cooperative exception handling and, in particular, exception resolution - in the traditional schemes developed for distributed systems, the processes do not usually fail by crashing, they always raise exceptions.

- the termination model of exception handling [8] is not applicable for the scopes used in multi-agent applications as activities in scopes can last for a very long time. The computational model used in the multi-agent systems is different from the ones used in the traditional sequential or distributed programming. For these systems we need reaction-based or event-based exception handling models in which the recovery structuring units (e.g., scopes) can run for very long time and are able to recover several times from the same or different exceptions.

- the traditional process systems are over restrictive in enforcing the action abort if there are exceptions raised during handling, these exceptions are typically propagated to a higher level action. Agent scopes should be longlived and allowed to continue with handling such exceptions.

5. EXCEPTION HANDLING MECHANISM

Our mechanism is based on cooperative recovery. When an agent is unable to recover from an error itself, it raises an *external exception*. Such an exception attempts to terminate activity of all agents in a scope and forces them to recover from the exception. All the agents initiate cooperative recovery independently. We use the resumption exception model [8]: if all agents succeed in recovery, then the scope proceeds with a normal activity. If one of the agents fails to recover, a new exception is raised in the scope. Applying this scheme to multi-agent systems gives a rise to a number of issues which need to be resolved:

- an agent cannot be forced to stay in a scope or participate in error recovery. It is the very nature of agents to migrate whenever they want. In addition, multi-agent systems are inherently decentralised and hence there is no notion of a global manager or controller.

- once an agent has left a scope, it cannot be involved in any activity of the scope, including error recovery.

- agents come from different authority domains and can be malicious or malfunctioning. Although it is not our intention to address security features, it is important to keep in mind that an exception can be purposely generated by malicious agents to break collaboration or get some private information.

- it is widely agreed that the asynchronous communication style is essential for mobile agents systems. All cooperative recovery schemes require some kind of global synchronisation, which is unacceptable in agent systems.

Despite the problems highlighted above, we believe that complex, large-scale mobile agent applications are unfeasible without a proper support for coordinated error recovery.

As we explained before, it is impossible to guarantee several essential properties of coordinated recovery, such as involvement of all agents and eventual handling of all exceptions, without violating the fundamental properties of agent systems. Our mechanism does not attempt to ensure such properties, at least in a general sense. Instead the mechanism is based on an entirely different ideology where participation in coordinated recovery is voluntary and the whole process of recovery is asynchronous. However, with proper application support, the mechanism can become similar to its analogue in (much simpler) distributed and parallel systems. In our approach, we offer application developers an opportunity to apply complex recovery schemes where they are needed without setting any limitations on the agent architecture in general.

5.1 Adding recovery actions

The structuring of agent activity in roles has an important impact on design of exception handling mechanisms. Each role encapsulates functionality of an agent in a particular type of a scope. This is the level at which the coordinated recovery is introduced. Separation of recovery actions from normal agent behaviour is achieved thanks to the structuring provided by the reactive model. Error recovery actions are introduced by extending a role with new reactions implementing coordinated recovery actions for possible external exceptions (see Figure 2). Each external exception is associated with a block of reactions.

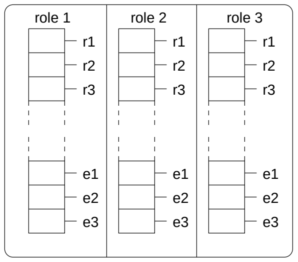


Figure 2: Structure of agent with recovery reactions

5.2 Raising exceptions

From an agent viewpoint, there are two exception types: local exceptions, thrown and handled inside an agent, and external exceptions, coming from outside. We assume that agents have some local exception handling capability and focus on the external exceptions. We will use the term *exception* to refer to an external exception.

An exception can be raised by an agent or by the middleware. In both cases an exception is sent to all the scope participants. Violation of the *scope liveness* conditions is the only case when the middleware raises an exception. An agent sends an exception to the scope participants when it fails to recover locally from an error. An error can be caused by a local agent failure or by a failed recovery initiated by an external exception. In general, coordinated recovery is an iterative process in which agents exchange exceptions until they find the one from which each of them can recover.

Agents do not directly deal with external exceptions. An external exception must be converted to a form specific for the language in which the agent is implemented. When an agent sends an external exception it first creates a native exception which is then transformed into an external one. In other words, external exceptions exist at a different abstraction layer and at each abstraction layer there is only one exception handling mechanism.

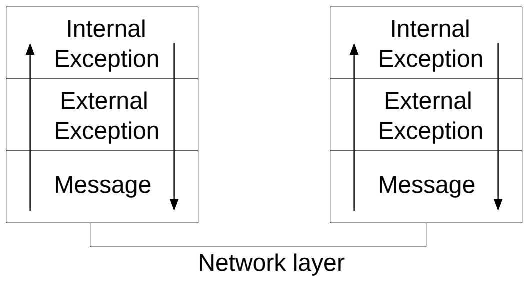


Figure 3: Exceptions transformation

A number of restrictions are imposed on how exceptions are raised by an agent. Firstly, it cannot be an arbitrary exception. All the collaborating agents must share the same set of exceptions. An agent must be designed in such a way that it can deal with these exceptions. This is an essential prerequisite for ensuring that all the agents interpret all the possible exceptions in the same manner. Secondly, an agent is limited in selecting exceptions by the *resolution function*. The function has two purposes - it guarantees termination of coordinated recovery and resolves a set of concurrent exceptions into a single one.

5.3 Handling exceptions

Before an agent can start recovering, an external exception must be transformed into a local one. An exception raised by an agent does not immediately trigger recovery actions in all other agents in the scope. For each agent the exception must be delivered to it and then the agent must stop all other activity and switch to the recovery mode. The exception delivery service in CAMA is provided by the middleware. In most aspects exceptions are similar to messages. The same transportation mechanism is used for messages and exceptions and all the problems and shortcomings of the message delivery also affect exception delivery. Exceptions can be lost and delayed. However, once an exceptions is delivered to an agent, it has a priority over normal messages. Execution of all the coordination primitives and most other operations on scopes can be interrupted by exception(s) pending for an agent. Non-blocking operations, like **out**(), check for an exception prior the their execution. Behaviour of blocking operations, such as input operation **in**(), is changed so they return either with a matched tuple or an exception, whichever appears first. If an operation detects an exception, it immediately returns and throws the exception in the point of its call. In our Java implementation of the mechanism, a new Java exception is created as an envelope for an external exception and is thrown using the standard statement *throw*. An additional benefit of the mechanism is a clear separation of normal and abnormal behaviour at the level of agent code ([11]).

Once an external exception is transformed into a native one, the agent switches to the recovery mode. If an agent succeeds in recovery, it switches back to the normal activity (see Figure 4, (a)). Otherwise it throws a *greater* exception, as defined by the resolution function, to all the scope participants and starts recovering from this new exception itself. Recovery can lead to a degraded operation mode where not all the previous agent activities are possible (see Figure 4, (b)). One example of such a situation is a failure of an agent in the scope, which leads to the loss of the service associated with the failed agent.

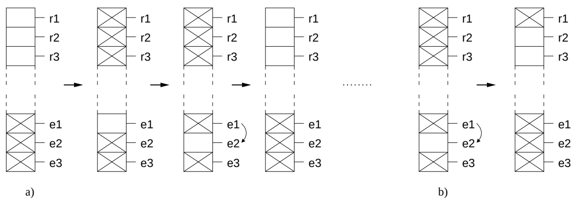


Figure 4: Evolution of agent in cooperative recovery

5.4 Operations

We introduce a number of new operations for working with the external exceptions.

$\boxed{\textbf{raise} \quad \textbf{syncwait} \quad \textbf{syncleave}}$

Operation **raise**(e_l) converts a local exception e_l into an external exception and raises it in a scope. The middleware propagates the exception to all agents of a scope. The operation fails if the scope liveness conditions are not satisfied.

Operations **syncwait** and **syncleave** are optional. They restrict an agent behaviour to make cooperative recovery more robust, at the same time making it more synchronous. Operation **syncwait** waits until a scope changes its state from blocked to operational mode, as determined by the scope liveness condition. The usual cause for a blocked scope is a failure, disconnection or migration of an agent. If agents in a scope have some additional information they can decide to wait until a disconnected agent reconnects, a new agent appears and so on. Operation **syncleave** acknowledges completion of agent activity in the scope and waits until all other scope participants acknowledge a completion. Instead of an acknowledgement, the operation can receive an exception. This would activate a normal procedure for cooperative recovery, after which the operation should be restarted.

5.5 Termination of coordinated recovery

Termination is a central problem for many distributed algorithms and we consider it to be important for our recovery mechanism. The termination is guaranteed by introducing a partial order on the set of external exceptions. We also require that the closure of the relation yields some common exception which terminates the whole scope activity. For example, a set of external exception can be defined as follows: $e_2 < e_3$; $e_4 < e_3$; $e_5 < e_4$. If an agent fails to recover from e_1 it can only raise exception e_2, e_4 or e_3. If recovery fails for the exception e_4, an agent must raise exception e_3. Exception e_3 is unrecoverable and forces all agents to abort the scope. The partial ordering of exceptions is encapsulated into the resolution function (similar to [12]).

5.6 Resolution of concurrent exceptions

It takes some time for an exception to reach an agent. And it also takes time for an agent to note presence of an exception. Thus, when an agent finally starts recovery, it could happen that there are several exceptions waiting. We use a resolution function to map a set of these exceptions into a single one. That single exception is associated with a handler which provides recovery for all the original exceptions.

In our mechanism the resolution function is based on the lattice structure (this idea was first mentioned in [4] but it was not used in the scheme proposed in the paper). It is more general than a common tree-based approach and allows the coordinated recovery mechanism to benefit from the context-awareness of agents. In the systems which do not support context-awareness, a new exception is chosen only on the basis of the set of received exceptions. In context-aware system, the context must affect the procedure of finding the resolved exception. The agent context and its state can provide a number of hints for choosing a recovery path. In the mathematical terms, a resolved exception is chosen from a set composed of the common parent of the pending exceptions in the exception lattice and all its parents. In the example of the resolution function given in Figure 5, the pair of exceptions e_1 and e_2 can be resolved into exceptions e_5, e_7, e_8 and e_9.

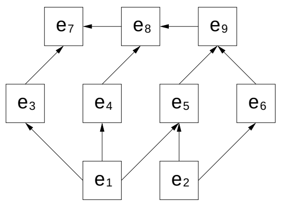


Figure 5: Lattice-based exception resolution

5.7 Further Discussion of the mechanism

In this section we show the applicability and the shortcomings of the mechanism proposed by analysing several typical scenarios. Diagrams are used to graphically represent a system configuration. Each line shows evolution of an agent over time. Actions executed inside of a scope are put into a bounding box. Each diagram contains only one scope; in the diagrams in Figure 7, a scope contains no agents at all for some time - this is demonstrated by several disjoint boxes. Non-blocking operations, such as **out**(), occupy a single time line point. Blocking operations, such as **in**(), can execute for some time and are shown as a time line interval. Their start and end points are designated separately. A blocking operation can finish successfully (shaded circle for the end point) or be terminated by an exception (cross-hatched circle). The case when an operation cannot start because of a pending exception e is shown as $= e$ in the operation name. Message travelling paths are shown by dotted arrows.

5.7.1 *Successful cooperative recovery*

This scenario demonstrates a successful coordinated recovery involving three agents. `agent 1` produces a message for `agent 2` (Figure 6). This messages causes an error in `agent 2`, which raises exception e_1. The exception terminates blocking **in**() operations in `agent 1` and `agent 3`. `agent 3` successfully recovers from the exception while `agent 1` raises a new exception e_2, which is greater (more general) than e_1. Exception e_2 interrupts blocking operations in `agent 2` and `agent 3`. This time all the agents successfully recover and switch to the normal activity and leave the scope asynchronously.

Asynchronous exit from a scope can lead to the situation when an exception is raised after some agents have already left the scope. This problem is discussed in the next scenario.

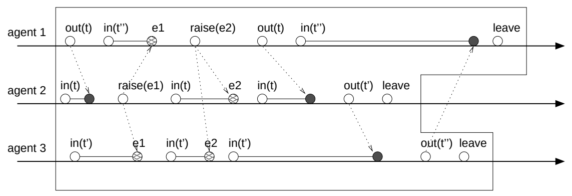


Figure 6: Agents recover together and continue their activity

5.7.2 *Asynchronous and synchronous exit*

In this scenario, `agent 1` from the previous configuration decides to leave the scope after it has produced the first message (see Figure 8, (a)). This message is consumed by the second agent which then writes another message for `agent 3`. However, `agent 3` finds a problem with the message and tries to involve all the scope participants into a coordinated recovery. It fails as `agent 1` and `agent 2` have already left the scope. The raise operation fails with an exception indicating this situation and `agent 3` can try to recover itself or abort the scope.

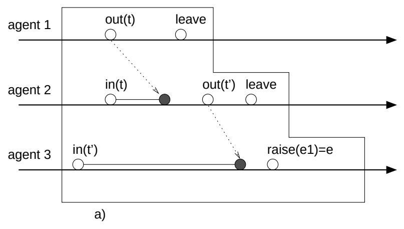
However, for some critical applications, it might be desirable to restrict agent behaviour in order to have a guarantee of a correct coordinated recovery. This is achieved with the operations **syncleave** which replaces asynchronous scope exit with a barrier-style synchronisation (see Figure 8, (b)). When all the agents finish their activity in the scope they synchronously leave the scope. This allows a failed agent to initiate a coordinated recovery at any stage of execution.

5.7.3 *Decoupled communication*

Time decoupled communication is an essential feature of mobile agents. In this scenario, agents communicate in a message board style. The scope is designed so that it permits only one agent at a time. It remains in a "frozen" state when there are no agents in it (hence its liveness condition is always true). An agent enters the scope, reads a message and posts a new one (Figure 7). If a read message causes a failure in an agent, the agent raises an exception. This exception is stored in the scope until another agent attempts to read (or post) a message. In this scenario, handling of an exception is done without involving the agent which has produced the offending message. We believe this is a normal practice for such mobile agent systems. Though it is impossible to propagate the exception to the source of failure, the primary objective - recovery of the scope - still can be achieved.

5.7.4 *Agent disconnection*

During a coordinated recovery, an agent may crash or decide to leave a scope. This may, or may not, depending on the scope configuration and the state of other agents, prevent recovery of the scope. If disappearance of an agent



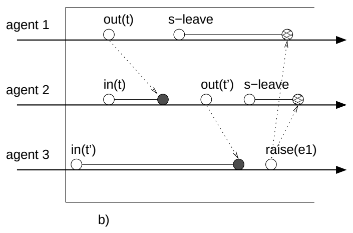


Figure 7: (a) Asynchronous exit can leave one of agents in trouble. (b) Operation `s-leave` provides synchronous exit.

breaks the scope liveness conditions, the remaining agents cannot communicate.

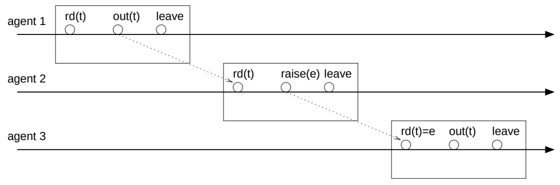


Figure 8: Coordinated recovery in a message board-style coordination

They can abort the scope or stop and wait for a new agent to enter the scope. Then, together with the new agent, they can proceed with the recovery. There are several other possibilities, such as when an agent disappears but the scope is still in operating mode. If the failed agent was not involved in any activity, its disappearance might go unnoticed. Moreover, with a proper design of the multi-agent application, it should be possible to continue scope activity since the failed agent cannot be a unique or critical one (because scope liveness conditions are not broken).

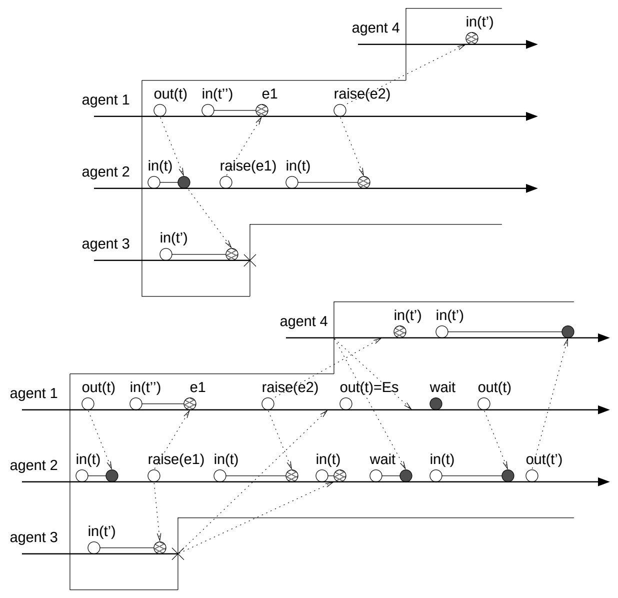


Figure 9: top: agent fails or disconnects during recovery; bottom: new agent appears in a recovering scope

The Figure 9,(top) shows a case when an agent crashes during a coordinated recovery (e.g. because of an external exception). However the failed agent is not a critical one and the scope activity may continue. One of the agents succeeds in recovery and attempts to restart the normal activity. However, the other one raises a new exception. A new agent appears in the scope and gets involved into the recovery as its attempt to read a message is interrupted by an exception. All the three agents continue with coordinated recovery.

In the second scenario, shown in Figure 9,(bottom), crash of the agent breaks the scope liveness conditions and the two remaining agents cannot recover themselves as the scope gets blocked. However, blocking of the scope does not happen immediately. Before the message about the invalid scope state is propagated, the remaining agents start recovering from another exception raised by one of the agents. After some time their activity is terminated with the exception indicating an invalid scope state. They both decide to wait until a new agent appears and the scope becomes operational. This is done by waiting with the operation *syncwait*. When a new agent enters the scope and attempts to issue an operation, it immediately gets an exception raised previously by one of the agents. At the same time, two other agents unblock and continue with normal activity while the new agent first has to recovers from the exception.

6. DESIGN WITH THE B METHOD

The mechanism we have presented in the previous section might look simple but the complexity of agent behaviour and coordination makes it difficult to analyse it for the potential pitfalls. Thus we decided to formally build the mechanism from scratch using the refinement technique supported by the B method [2]. The whole mechanism is too large and complex for a formal analysis. This is mainly due to the ability of agents to dramatically change the style of cooperative recovery by deciding whether to use the **syncleave** and **syncwait** operations. We decided to restrict our formalisation to the case when agents synchronously leave a scope and do not attempt to recover when scope liveness conditions are not satisfied. However we succeeded to model and verify all the main features of the mechanism: asynchrony in recovery, termination of coordinated recovery, lattice-based resolution function, concurrent exceptions and non-deterministic failures in agents.

In the rest of this section we discuss the main principles and highlight the interesting points of our experience the verification of the proposed cooperative recovery mechanism.

6.1 General principles

We start formal design of the mechanism with an abstract model of the Cama infrastructure and then formally develop it by gradual incorporation of implementation details. In the development process we focus on integrating cooperative exception handling into the formal specification of Cama by specifying and refining error detection and error recovery. Application of formal methods allows us to develop systems that are correct by construction and prove the essential properties of such systems (e.g., termination of error recovery).

Our chosen formal framework is the B Method - a formal approach to the industrial development of highly dependable software. The development methodology adopted by B is based on stepwise refinement of an abstract system model into an implementable program. Since the method uses theorem proving for verifying correctness of refinement transformations, it is free of the state explosion problem and fits well for designing large complex systems, such as Cama: While refining a system we preserve globally observable behaviour but change the local data structure and control flow to implement the desired behaviour.

In this paper we use the Event-B [1, 16] version of the B Method to reflect the reactive nature of the Cama systems. In Event-B the system behaviour is specified in terms of events (system reactions). While refining event-based systems, we elaborate on existing events as well as introduce new events specifying behaviour on the newly introduced local variables.

In our formal development we focus on specifying cooperative exception handling in a single scope. We assume that a number of agents has joined a scope and perform some activities in it. However, any agent can fail spontaneously or decide to leave the scope at any moment. The remaining agents try to recover from these errors but might fail themselves while doing it. Therefore, error recovery is iterative. The recovery should eventually result in either restoring a normal system state or closing the scope due to unrecoverable error.

6.2 Initial specification and first refinement

In the abstract model we specify a global, high-level view on the system behaviour. Namely, the system might be in one of two states: normal and stopping. The state machine representing the system behaviour is given below.

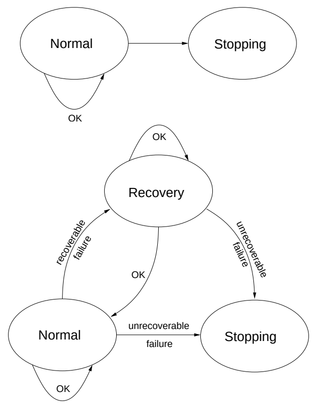


Figure 10: State chart of the initial model (top) and states of the refined model (bottom)

The system can remain in the normal state, performing some activity. However, upon occurrence of a critical (i.e., unrecoverable) error, the system is transferred to the stopping (i.e., closing) state. Such a simple system model can be trivially specified in Event-B with the corresponding event. Note that in our initial model we abstract away from agent representation, yet we introduce them in the later refinements.

In our initial model we abstract away from specifying various failure modes, so that any error leads to scope closing. In our first refinement we introduce a representation of various failure modes, i.e., distinguish between recoverable and unrecoverable errors. Upon detecting a recoverable error, the system interrupts normal activities and enters a recovery state. At this state it attempts to execute various recovery actions to restore the normal system state. As a result of our first refinement step we extend the abstract model with the explicit representation of the system recovery state and the corresponding state transitions. The extended state machine is depicted by the bottom diagram in Figure 10.

In the corresponding B model, the additional recovery state and state transitions are defined in the new event operation *Recovery*. Error recovery can be iterative, which means that the execution of this operation can loop. One of the essential properties we should guarantee by our formal development is that error recovery will eventually terminate. This is achieved by proving that the model is a valid B refinement.

To prove refinement in B, we have to show that new events do not take control forever. This is done by supplying a *variant* - a natural number expression which is decreased after each execution of a new operation. Since our system model is still very abstract, we solve the termination problem by introducing an abstract variable *n_rec*, which is forcibly decreased by the operation. Once we introduce sufficient implementation details into our model, we will replace this abstract variable with an a more detailed expression of the refined system. In the refined specification the current value of the system state (Normal, Recovery, or Stopping) is stored in the variable *sys_state* of the B model.

6.3 Second refinement

In this refinement step we distribute the system behaviour among a group of active agents participating in a scope. We assume that each active agent is in either normal or recovering state. The current values of agent states are stored in the array variable *ag_state*. In addition, we introduce the abstract predicate *min_cond*, which determines whether the system (scope) can continue its activity with the current group of active agents, i.e., liveness condition. This abstract predicate can be instantiated with the actual liveness conditions for concrete scopes. To establish validity of this refinement step, we should establish a connection between the abstract system state used in the previous model and the concrete (distributed among the active agents) state defined in the refinement. In B, we do this by defining the relationship - data invariant - between the abstract variable *sys_state* and the concrete variables *ag_state* and *min_cond*. For example, the following part of the data invariant

$$\begin{aligned}&RecoveryState \in ran(ag\_state) \wedge \\ &\quad min\_cond(active\_agents) = true \\ &\qquad \Rightarrow sys\_state = Recovery\end{aligned}$$

defines the relationships between the state of active agents in the refined specification and the system state in the more abstract specification. Namely, it stipulates that while at least one active agent is in the recovery state (i.e., $RecoveryState$ belongs to the range of the function ag_state) and the liveness condition is still satisfied, then in the more abstract specification the system is in the recovery state. The remaining parts of the data invariant formulate similar relationships associating the normal and stopping states of the abstract model with the corresponding states of the refined system.

Since system recovery is now distributed among the active agents, we also have to redefine our variant expression n_rec needed to prove termination of recovery. We assume that each agent decreases its own variant expression $ag\_nrec(agent)$. Then we can define the abstract variable n_rec via concrete variables $ag\_nrec(agent)$ in the following way:

$$n\_rec = \sum_{agent \in inactive\_agents} ag\_nrec(agent)$$

6.4 The consequent refinement steps

We aim at specifying cooperative error recovery in the distributed CAMA system. Hence in general all agents in the scope should participate in recovery from another agent failure. To achieve this the exception generated as a result of an agent failure is broadcasted to all active agents in the scope, thus involving them in the cooperative recovery. In this refinement step we extend our model by introducing an abstract data structure for modelling exceptions, as well as the special event operation broadcast to model broadcasting of the current exception to the active agents.

The operation broadcast is an operation of middleware. Hence it is centralised, i.e., not distributed among the active agents. It is responsible for "spotting" the exception to be propagated (either because of a spontaneous failure of an agent or the inability of an agent to recover) and then delivering it to the remaining active agents.

In the previous refinement step we introduced a set of exceptions that are broadcast to the agents during error recovery. However, we abstracted away from specifying which particular exception has to be propagated in each situation. Also, to show termination of error recovery, we used the variant expressions $ag\_nrec(agent)$ defined for each active agent. However, these expressions still remain underspecified. In order to finalise proving termination of recovery in our model, we have to instantiate these expressions with concrete data of the system.

During this refinement step we tackle these two problems by introducing the (partial) order between exceptions. The order is based on criticality of exceptions. All agents use the same mathematical structure (a lattice) defining the order on the exceptions. The introduced order allows us to guarantee that, when an agent fails to handle a particular exception, it generates a more critical (i.e., more general) exception. The error recovery continues by handling this exception. We assume that there is the most critical exception $TotalFailure$ which leads to closing the system. By associating the agent variants $ag\_nrec(agent)$ with the (inverse) criticality of the last generated exception for each agent, we can prove that this expression is decreased every time an agent remains in the recovery state. Therefore, it also proves that the recovery process of all agents eventually has to terminate. Due to the space limit we have omitted representation of the formal specifications. The complete B specification can be found in [13]. In the consequent refinement steps we can further elaborate on the created model of fault tolerance mechanism in the CAMA system by modelling, e.g., internal recovery, subscopes etc.

7. SUMMARY

In this paper we propose a mechanism for cooperative recovery in multi-agent systems. Our mechanism is agent-centric - we put first the needs of an individual agent. It is based on the extended interpretation of agent autonomy: each agent has freedom to decide what role it should play in cooperative recovery. Arguably, such freedom is crucial for open multi-agent systems. We address the core points of cooperative recovery such as termination and resolution of concurrent exceptions. We believe that this scheme is realistic and lightweight. The experience from the prototype version of the mechanism for the CAMA system shows that the mechanism smoothly integrates with coordination paradigm and performs well in real applications.

The proposed mechanism has been formally analysed to identify possible problems. In particular, we have demonstrated how to formally guarantee termination of the iterative error recovery. Our approach is based on gradual top-down development by stepwise refinement and relies on theorem proving for verification of system properties. The formal modelling has helped us in making several decisions improving the final version of the mechanism.

8. ACKNOWLEDGEMENTS

This work is partially supported by the EPSRC/UK TrAmS platform grant and the FP7 ICT DEPLOY integrated project. We are grateful to the anonymous referees for their comments.

9. REFERENCES

[1] J. R. Abrial. Extending B without changing it (for developing distributed systems). In H. Habrias, editor, *1st Conference on the B method*, pages 169–190. IRIN Institut de recherche en informatique de Nantes, 1996.

[2] J. R. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

[3] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[4] R. H. Campbell and B. Randell. Error recovery in asynchronous systems. *IEEE Transactions on Software Engineering*, 12(8):811–826, 1986.

[5] B. Carbunar, M. T. de Oliveira Valente, and J. Vitek. Coordination and Mobility in CoreLime. *Mathematical Structures in Computer Science*, 14(3):397–419, 2004.

[6] F. Cristian. Exception Handling and Fault Tolerance of Software Faults. In M. Lyu, editor, *Software Fault Tolerance*, pages 81–107. Wiley, NY, 1995.

[7] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[8] J. B. Goodenough. Exception handling: issues and a proposed notation. *Commun. ACM*, 18(12):683–696, 1975.

[9] A. Iliasov. Implementation of Cama Middleware. Available online at http://sourceforge.net/projects/cama.

[10] A. Iliasov and A. Romanovsky. Exception Handling in Coordination-based Mobile Environments. In *Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, pages 341–350. IEEE Computer Society Press, 2005.

[11] A. Iliasov and A. Romanovsky. Structured coordination spaces for fault tolerant mobile agents. In C. Dony, J. L. Knudsen, A. B. Romanovsky, and A. Tripathi, editors, *Advanced Topics in Exception Handling Techniques*, volume 4119 of *Lecture Notes in Computer Science*, pages 181–199. Springer, 2006.

[12] V. Issarny. 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)*, pages 29–39, October 1993.

[13] L. Laibinis. B specification of the Cama exception handling mechanism. Available online at http://cama.sourceforge.net/downloads.html.

[14] L. Laibinis, E. Troubitsyna, A. Iliasov, and A. Romanovsky. Rigorous development of fault-tolerant agent systems. In M. J. Butler, C. B. Jones, A. Romanovsky, and E. Troubitsyna, editors, *RODIN Book*, volume 4157 of *Lecture Notes in Computer Science*, pages 241–260. Springer, 2006.

[15] G. D. Marzo and A. Romanovsky. Using exception handling for fault tolerance in mobile coordination-based environments. In *ECOOP 2003, workshop on Exception Handling in Object Oriented Systems: Towards Emerging Application Areas and New Programming Paradigms, Darmstadt, Germany.*, 2003.

[16] C. Metayer, J. Abrial, and L. Voisin, editors. *Rodin Deliverable D7: Event B language*. Project IST-511599, School of Computing Science, Newcastle University, 2005.

[17] S. Pears, J. Xu, and C. Boldyreff. A dynamic shadow approach for mobile agents to survive crash failures. In *ISORC*, pages 113–120. IEEE Computer Society, 2003.

[18] G. P. Picco, A. L. Murphy, and G.-C. Roman. Lime: Linda Meets Mobility. In *Proceedings of 21st Int. Conference on Software Engineering (ICSE'99)*, pages 368–377, 1999.

[19] G.-C. Roman, C. Julien, and J. Payton. A Formal Treatment of Context-Awareness. In M. Wermelinger and T. Margaria, editors, *Fundamental Approaches to Software Engineering, 7th International Conference, FASE 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, LNCS 2984*, pages 12–36. Springer, 2004.

[20] F. Souchon, C. Dony, C. Urtado, and S. Vauttier. Improving exception handling in multi-agent systems. In C. J. P. de Lucena, A. F. Garcia, A. B. Romanovsky, J. Castro, and P. S. C. Alencar, editors, *SELMAS*, volume 2940 of *Lecture Notes in Computer Science*, pages 167–188. Springer, 2003.

[21] A. R. Tripathi and R. Miller. Exception handling in agent-oriented systems. In A. B. Romanovsky, C. Dony, J. L. Knudsen, and A. Tripathi, editors, *Advances in Exception Handling Techniques*, volume 2022 of *Lecture Notes in Computer Science*, pages 128–146. Springer, 2000.

[22] M. Wooldridge and P. Ciancarini. Agent-Oriented Software Engineering: The State of the Art. In P. Ciancarini and M. Wooldridge, editors, *First Int. Workshop on Agent-Oriented Software Engineering*, volume 1957, pages 1–28. Springer-Verlag, Berlin, 2000.

[23] J. Xu, A. Romanovsky, and B. Randell. Concurrent exception handling and resolution in distributed object systems. *IEEE Transactions on Parallel Distributed Systems*, 11(10):1019–1032, 2000.