Using Exception Handling for Fault-Tolerance in Mobile Coordination-Based Environments

Giovanna Di Marzo Serugendo[1] and Alexander Romanovsky[2]

[1] Centre Universitaire d'Informatique, University of Geneva,
CH-1211 Geneva 4, Switzerland
Giovanna.Dimarzo@cui.unige.ch

[2] School of Computing Science, University of Newcastle,
NE1 7RU Newcastle upon Tyne, UK
Alexander.Romanovsky@newcastle.ac.uk

1 Introduction

Mobility of users and code, coupled with todays powerful handheld devices, allows us to anticipate that a class of applications, which will take more and more importance in the next years, is that of mobile agent-based applications. Indeed, handheld devices need light code that can be freely moved from one device to another, according to the user's mobility or needs. Mobile agent-based applications typically run on a mobile coordination-based environment, where programs communicate asynchronously through a shared memory space. There is a number of outstanding issues in providing fault tolerance of such applications. The aim of this paper is to propose an exception handling model suitable for mobile coordination-based environments.

1.1 Exception Handling

One of the chief trends in providing dependability of modern systems is the decreasing role played by tolerance to hardware-related faults. This is due to several factors: improvements in hardware quality, a dramatic growth of the complexity of software, the increasing involvement of the unexperienced users in managing such systems, a growing variety of abnormal situations in the system environment. As a result of this, modern applications have to be designed in such a way that they are capable of dealing with a growing number of various abnormal situations in a disciplined fashion. The only general solution to these problems is to systematically incorporate software fault tolerance into the applications. One approach is to use backward error recovery techniques (such as rollback) which can be made almost transparent for the application. Unfortunately it is not general, usually quite expensive, and often not-applicable at all. This is why employing application-specific exception handling is nowadays playing a major role in building complex dependable applications.

Exception handling relies on features for declaring exceptions in different scopes (exception contexts) and associating handlers with them, as well as on ability to raise exceptions and propagate them outside the (nested) scopes. Most practical languages (including Java and Ada) as well as a number of component technologies (such as EJB) provide features for handling exceptions. Exception handling features should match the characteristics of the application to be developed and the environment, the design paradigm, the computational model and the language/technology used [8]. Developing specialized exception handling mechanisms is an area of a very active research. For example, a number of object-oriented mechanisms were proposed around 1990, atomic action based mechanisms for providing fault tolerance (exception handling) in concurrent cooperative systems were developed in mid 80s.

1.2 Coordination-Based Mobile Environments

Mobile coordination-based environments usually follow a data-driven coordination model [3], using a shared data space à la Linda. Agents coordinate in a Linda space by inserting, reading or removing tuples of data from a blackboard. Tuples are retrieved according to an associative search. Mobile coordination-based environments provides several tuple spaces and ways for agents to denote and access the different spaces.

Among mobile coordination-based environment, we can cite Lime [7], MARS [2], and Lana [1]. Lime (Linda in a mobile environment) is well suited for both logical mobility of agents and physical mobility of devices. Lime mobile agents coordinate through Linda tuple spaces using a fixed set of interaction primitives. Lime tuple spaces at different sites are merged together and agents access them transparently as if they were local. MARS (Mobile Agent Reactive Spaces) is an object-oriented Java-based environment, where Java agents coordinate through an object-oriented tuple space à la Linda, using a programmable set of primitives. A MARS agent can only access the tuple space associated to the host where it is currently executing. Lana is an object-oriented Java-based environment, which combines provision of coordination and security and is designed to run on both standard user PCs connected to the Internet and handheld devices. Lana agents coordinate through an object-oriented tuple space à la Linda, using a default fixed set of primitives, which can nevertheless be extended by the programmer. An agent can access both remote and local tuple spaces. Lana defines protection domains for information access, and prevents application crashes by considering network failures as normal events.

1.3 Fault-Tolerance in Mobile Coordination-Based Environments

Early work on fault-tolerance in coordination-based environments focused on using transactions in Linda-like environment. Two additional primitives allow all produced tuples to be retained until the transaction commits, the tuples are then actually added to the tuple space. Retaining tuples ensures the transaction semantics, but raises problems among dependent agents. A relaxed version [9] provides the all or nothing transaction semantics, avoiding problems among dependent agents, by relaxing the atomicity property. Tuples are available immediately in the tuple space, even if later the transaction does not commit.

Mobile coordination-based environments, like WCL and JavaSpace, define notification mechanisms. Agents ask to be notified whenever a matching entry is written to the space. The notification mechanism, coupled with the notion of transactions, provides a fault-tolerant mechanism, where an event catcher is notified of matching entries for the duration of the transaction. In the case of Lana, fault-tolerance is achieved in an asynchronous fashion, through the notion of events. Events are deposited into the tuple space and retrieved by the corresponding agent: immediately, if the agent was waiting for the event; later on, if it searches for it later in the tuple space; or never, if the agent does not care about the event.

However, all the above mentioned mechanisms for dealing with fault-tolerance at the application level, suffer from the fact that exception handling, in its traditional sense, cannot be realized. Indeed, an exception, signaling an abnormal event, is not necessarily caught or handled, neither synchronously or asynchronously, by an agent.

1.4 Contribution

The focus of this paper is on discussing a new model of exception handling suitable for mobile applications developed in coordination-based environments. Our general view here is that exceptions are special (abnormal) events that cannot be treated as usual (normal) events. Exceptions have to be *always* caught and handled, they are abnormal events always needing reaction. Generally speaking, the agent putting tuple in the space and continuing its work acts under wrong assumption that its processing would go smoothly. The problems here are: due to decoupling event producers from consumers this agent continues its execution as if no exception has been raised, so it is not ready to be involved in handling. Moreover, it can leave the location. There is clearly a need to have (to dynamically create or link) a local handler that would synchronously deal with exceptions, while the processes raising exceptions keep communicating asynchronously. In addition to this, it is important to make exception handling as flexible as possible. For example, to offer features to change dynamically the direction of exception propagation.

2 Running Example

We introduce a small banking system that operates with mobile agents and requires fault tolerance, and which is implemented in the Lana environment. This example will be used to illustrate the discussion on the fault-tolerant techniques intended for mobile coordination-based environments.

2.1 The Lana Environment

Since Lana will serve as a basis for our discussion on fault tolerance, we will first explain some Lana features. Communication inside a program occur through synchronous method calls. However, Lana programs communicate using *asynchronous* method calls. These calls are secure in that the calling program is given a fresh *Key* object when the call is issued, and only this key can be used to interpret the method reply. This mechanism prevents malicious programs from intercepting or tampering with messages destined for others.

Events are used for signaling returned values of asynchronous method calls. They are also used for signaling errors or exceptions, such as: security violation (no permission for a method call), or the fact that the required program has moved. Pre-defined event types include those: that indicate that a method has returned (`MethodReturn`); that the method call has failed due to the invoked program having moved (`MigrationEvent`); no access right having been granted (`SecurityViolation`); the execution of the method code has generated an exception; the remote platform generated an (system) exception. *Keys* are used to lock objects and events. *Unique* keys are automatically generated for each asynchronous method call. *Fixed* keys can be generated by several programs, they allow transfer of well-known object copies through the use of a common key.

The asynchronous notification of events is at the heart of the current fault-tolerance mechanism provided by Lana. The typical scenario is the following: an agent performs first an asynchronous method call, it then continues its execution (without waiting for the method return), once it is interested in having the method return, it performs an `observe()`, which blocks the agent until the method has returned. The return can either be the expected result, or an "exception" explaining the problem. Lana allows such exceptions to be generated from a remote platform, however the `observe()` command must be local, i.e., an agent that moves must delegate the `observe()` to an agent that stays in the platform from where the call has been issued.

2.2 Market Place Example: Lana Design

Figure 1 shows a user, wishing to acquire some product. He launches a buyer agent that will roam the Internet searching for a seller agent offering the requested object. The buyer and seller agents will meet in an electronic market place, where they will exchange information regarding products that their respective users desire to buy or sell. Each agent is equipped with an e-purse holding some amount of electronic money. If the two agents have matching requests, they reach an agreement (the product is booked, and a contract is concluded), and the payment is then realized by transferring some money from the e-purse of the buyer agent directly to the e-purse of the seller agent. In this case the e-purse acts as cash money, there is no need to ask for a bank certification. If the payment fails, then the agreement has to be cancelled, and the seller agent releases the product. An e-purse is implemented by an additional agent that handles the amount of money of the e-purse, access privileges, deposit and withdrawal operations.

In this scenario, the buyer agent can be either alone or composed of several agents roaming the Internet simultaneously. It may be difficult or even impossible for the user to contact the buyer agent or for the buyer agent to contact the agents distributed world-wide. Indeed, a mobile IP scheme enabling to contact mobile agents irrespectively of their position is difficult to consider when agents neither know each other nor participate in the same application.

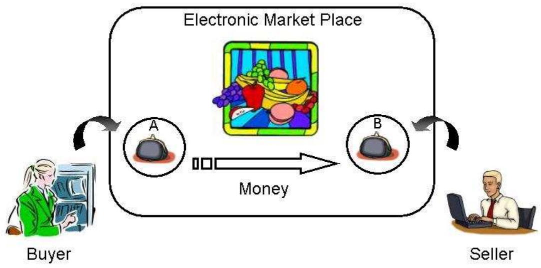

Fig. 1. Market Place Example

This examples consists of two main Lana agents: the buyer and the seller agent that work on behalf of the buyer, and seller respectively. Once both the buyer and seller agents have reached the market place, a sequence of data exchange occurs through the message board (in the case where there is no error): the buyer (seller) agent inserts a buying (selling) request respectively in the tuple space. The buyer agent retrieves the selling request and proposes a contract, which is accepted by the seller. The interaction ends with the payment, i.e., by transferring money from the buyer e-purse to seller e-purse.

We will now consider three types of errors: system error, application errors, network failures; and see how they can be handled within the current Lana model:

- *System error.* The local or remote platform generates an error, e.g., the code cannot be executed, the called agent has moved, etc.
- *Local Application Errors.* For instance, we can mention:
  - there is no offer matching the request;
  - there is a bug in the seller agent code: it is impossible to reach an agreement or to access to e-purse;
  - the buyer e-purse does not contain sufficient money;
  - the buyer e-purse does not present sufficient privileges, e.g., a configuration error does not authorize money to be withdrawn from the e-purse.

- *Distributed Application Errors.* Several agents work cooperatively, e.g., to buy several items of the same series. In the case of an error, there is a need for a collective recovery of the error.
- *Network Failures.* There are communication failures between the buyer and seller agent platforms, or between the agent platform and its user.

In the case of system errors, the local or remote Lana platform generates an event, i.e., it inserts the corresponding tuple in the shared memory space. Thanks to the key, the tuple is then retrieved by the agent that was waiting for the corresponding operation.

In the case of application errors, e.g., there is an error during the payment due to insufficient money, the buyer agent inserts a payment fail event, retrieved by the seller agent that was waiting for the payment. The latter cancels the transaction, and reinserts the selling request. Finally, the buyer agent informs the buyer in the original platform that an error occurred during the payment, it inserts the corresponding event in the original platform.

The case of distributed applications errors is the most interesting one. Indeed, the problem here is to reach, even asynchronously, but necessarily, agents whose location is not known. A possible approach to realizing this in Lana is for the agents to agree upon a common platform where events, related to abnormal situations, are stored. After that both the Buyer and the Seller agents leave an additional assistant agent in this platform, whose role is to observe those events, and to inform their respective agents.

Finally, the Lana model allows network disconnections to be handled in a very similar way. Agents are informed of the failure through events, and either wait for the availability of the connection, or continue their execution until the connection is up, or definitely give up.

2.3 Analysis

Lana treats exceptions as the conventional tuples and offers basic primitives for implementing exception handling. But we have found that there is clearly a space for expanding these features to help application programmers in developing fault tolerant applications in a safer and less error-prone way. First of all, Lana allows exceptions to be left unhandled - which is clearly error-prone and can have serious effects on system reliability. Secondly, this model mixes normal and abnormal flows of control and code, and does not separate sufficiently the normal system behaviour from the abnormal one - which is the main idea behind exception handling [6] (as a matter of fact in their early work the authors of Lana stated clearly [1] that exceptions should not be introduced as normal events). Thirdly, the Lana model does not include any specific support for exception handling and leaves all complicated issues with the application designers. These has several serious consequences which may complicate system design and make it more error-prone. One of the example is that to guarantee that the information is delivered the producer should wait for the notification to be implemented at the application level, in the code of both the producer and the consumer. Another example is that Lana does not provides any systematic ways for transferring responsibility for handling exceptions from the producer of an event to some other process/code. Moreover, this model does not introduce the concept of exception context, which is crucial for any exception handling methodology - which makes it impossible to understand who handles exceptions, when and if they are handled at all. Too many responsibilities (and room for mistakes) are left with application programmers. In addition to this Lana exception handling does not support nesting, nor it supports cooperative handling that involves several processes.

3 Exception Handling Model

In this section we discuss our approach to introducing coordination-based exception handling which specifically focuses on code mobility.

3.1 Requirements

Novel fault tolerance techniques to be applying in developing complex mobile systems have to correspond to their specific characteristics. As our analysis in [4] shows the main way of ensuring fault tolerance in such systems is by employing application-specific exception handling mechanisms. Among other important properties these mechanisms should be light and flexible, they should allow for dynamic adjustments and for autonomous (localized) handling of abnormal situations. These mechanisms should be suitable for open systems in which mobile agents are to operate in unknown environments and be capable of dealing with abnormal situations that are not known in advance.

3.2 Model

The first problem here is to stay within decoupled (asynchronous) communication model and at the same time to guarantee delivery of exceptions to handlers. Only in this case we can guarantee that all exceptions are caught and after that handled. Our idea is to keep using usual standard asynchronous ways of communication (i.e. keys or events) but to make sure that, should an exception be raised, the handler always exists in the location.

As depicted in Figure 2, the underlying idea here is that we assume that, if an `event` is consumed by a program `EventConsumer`, and an exception `E` is raised during processing of this event, then this event is the cause of the exception, and such exception should be treated outside `EventConsumer`. One possible solution is to allow only synchronous communication, in which case the producer of the event, `EventProducer` would be the best handler. But if we want to allow asynchronous communication and agent mobility we cannot bind the producer. So the only possible solution is to create a new process `H(E)` to handle the exception.

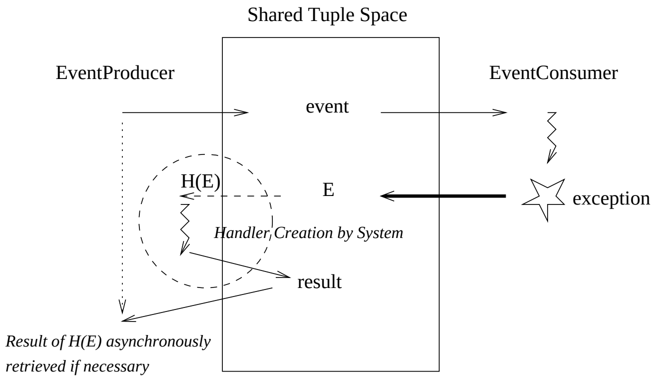

Fig. 2. Exception Model

The specialized process `H(E)` is created when an exception `E` is signaled by `EventConsumer`. In our approach any exception is a tuple of a special type: there is always a process waiting to handle (i.e. to consume) it. Such handler `H(E)`, representing the exception context, is usually designed by the developer of `EventProducer` but it can be designed by the designer of `EventConsumer` as well. In the latter case either the handler provided by `EventProducer` is ignored in runtime and only the handler supplied by `EventConsumer` is used, or both handlers receive exception `E` (in which case they can decide to handle it cooperatively). The handler process usually completes handling and dies. But they do not have to. Generally speaking, they can be involved in further system execution for as long as necessary.

In our approach each tuple `T` produced by `EventProducer` has a number of exceptions declared in its signature: `E1`, .., `En` in addition to a set of parameters. When `T` is put into a tuple space the system should have references to declarations of n handlers: one for each exception. The handler process is created locally, when an exception is signaled but it does not have to be always local: it can move to better handle the exception. But it is important to always create it when an exception is signaled. We believe that it would be wrong to make any existing process to be a handler because there is no guarantee that it will handle the exception without delays: it can move to another location before an exception is signaled or it can be busy doing other job and because of the asynchronous nature of communication may decide to handle it when it is too late.

Our scheme allows defining a process handling several (or, even, all exceptions) from the signature of `T`, or, even, from the signatures of different tuples.

One problem we are planning to address is: what happens if `EventConsumer` moves to another location and signals exception `E` while continuing processing the consumed tuple `T` in this location. We should be able to create handler `H(E)` in the new location because this is where the exception tuple is put in the local tuple space.

To make our scheme even more flexible we plan to allow: an exception to be propagated to several handler processes and the handler and the exception to be dynamically associated (e.g. while inserting `T` into a local tuple space).

Another feature which would add flexibility to our scheme is to allow any existing process, which is local to `EventConsumer` and to exception `E`, to handle it together with standard handler processes. In particular, it seems to be useful to allow `EventProducer` to join in the handling.

3.3 Market Place Example Revisited

We demonstrate our approach using the market place example introduced earlier.

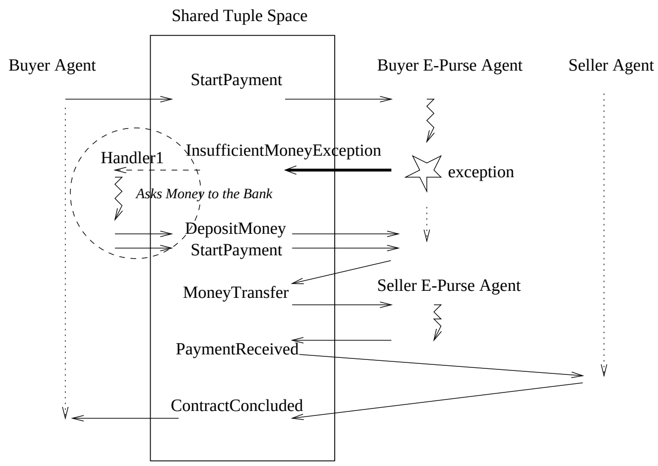

Fig. 3. Example Revisited: Successful Recovery

In this particular scenario the Buyer Agent asks its E-Purse Agent to release the money and provides the E-Purse Agent with a name of its proxy agent (Handler1) that can be locally created when necessary to deal with exceptions. The Buyer E-purse Agent raises an exception due to the fact that there is not enough money in the e-purse for realizing the payment. The system then created Handler1, which catches the exception. Handler1 accesses the bank, in order to transfer money from the bank account to the e-purse (see Figure 3). It deposits then the money on the e-purse, and starts again the payment procedure. The money transfer from the Buyer e-purse to the Seller e-purse occurs now correctly. The Seller e-purse agent then informs the Seller agent that the payment has been realized, and the Seller agent finally informs the Buyer agent that the transaction was successful.

Figure 4 shows a different case. Handler1 does not succeed in retrieving additional money from the bank (the bank account is not sufficiently furnished, or there are no privileges). Handler1 informs the Buyer Agent through a regular

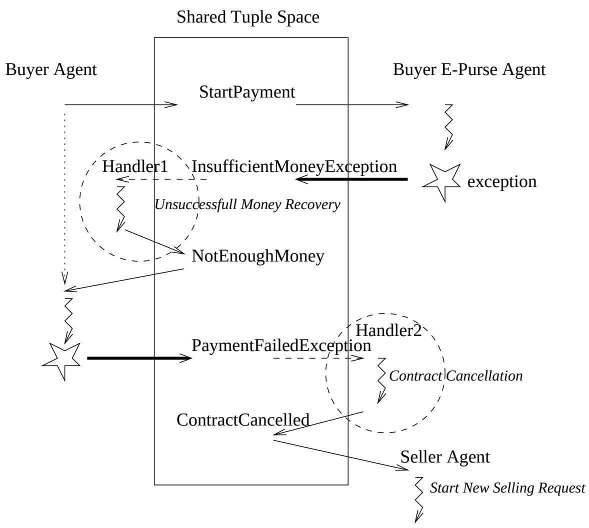

Fig. 4. Example Revisited: Failed Recovery

event about the insufficient money present in the e-purse. The Buyer Agent is not able to cope with this situation, and raises an exception, caught by Handler2, provided by the Seller Agent. Handler2 is in charge of cancelling the on-going contract, it then informs the Seller Agent, that subsequently starts a new selling request. As an alternative, we could also envisage that Handler2 creates an additional Seller Agent that will start the same selling request, leaving the original Seller Agent free to continue its work (e.g., buying a series of items). This is particularly useful, when the original Seller Agent needs to move.

This simple example allows us to draw several conclusions:

– in Lana the interacting agents have to always wait until the end and until all notifications have been received, but in our approach they do not because the handler agents can handle problems (our solution allows us to build really asynchronous systems);
– moreover, our approach allows agents to freely move to other locations and start other work without waiting for all notifications - if an exception is signaled in the original locality there always is a handler to deal with it locally;
– in Lana all handling is a part of the Seller, the Buyer and the E-Purse Agents but in our approach handling is implemented in separate handler agents - our design is cleaner and we separate the normal code from the abnormal one. Because of this, our approach is more flexible. We can, for example, use different handler processes for different locations which the Buyer Agent visits, while in Lana handling is hardwired into the Buyer Agent.

With respect to dealing with distributed application errors we will be extending our approach with the concept of the context (see the Discussion Section) but in Lana everything again should be developed explicitly by the application programmers with no additional support from the system.

It is clear that any particular feature which our approach provides can be implemented in Lana or any other coordinated models. The real question is how expensive and error-prone these solutions are and what sort of support the programmers have to systematically employ exception handling. This is where our solution provides a number of benefits ensuring, for example, that any exception is always treated. In addition to that our approach respects asynchronous communication mechanism and does not restrict unnecessarily process mobility.

4 Related Works

To the best of our knowledge there are no general exception handling features developed for mobile coordination-based environments meeting the requirements above. There are only few relevant papers on exception handling in coordination-based systems. Diaz et al [5] put forward a basic exception handling framework for the logic channel-based coordination model. In this model when a process is created a special logical channel is associated with it, so when the process raises an exception it is propagated through this dedicated channel. The idea here is that when a process is created a special process used for handling all its exceptions is created and sent as well. This scheme is not oriented towards mobile environments, it relies on asynchronous exception handling making no difference between normal and abnormal events (we will discuss this issue in detail later on) and, besides, associating handlers with the process processing information appears to be very static. Two more issues with this approach are: there is one handler for all possible exceptions raised by a process and there is no support for raising exceptions in several processes.

Another relevant work, by Rowstron [9], considers fault-tolerance for distributed tuple space, in the framework of stationary agents. It introduces the notion of "mobile coordination", where mobile is related to the movement of coordination primitives, forming together a coordination operation, to the server storing the tuple space. The operation is executed according to an all or nothing semantics, which nevertheless allows intermediate tuples to be available in the tuple space. Fault-tolerance for small segments of programs is then realized by transferring the corresponding primitives, and associated state, to the server. This approach does not enable the propagation of exceptions, in their traditional sense, outside the coordination operation. However, the additional notion of "agent will" allows to perform some exception handling outside coordinations. Wills are mainly used to cleanup the tuple space if an agent fails, and are associated with a tuple space, and the same will is applied to all operations.

5 Discussion

In our future work we plan to apply the ideas discussed in the paper for developing a novel exception handling mechanism for Lana. One topic of our following work will be introducing rules for scoping to involve all processes from the same scope in cooperative handling of any exception signaled by any of them. All processes belonging to the same locality is the most likely candidate for introducing scoping rules but we are considering other ways of scoping as well. This would resolve the following problem with the solution proposed in Section 3: we implicitly assume here that the consumer always detects an error and signals an exception before it produces any tuple or before it starts consuming the next tuple - which is in effect a very strong assumption for many practical situations.

6 Acknowledgment

We are grateful to M. Pawlak and C. Bryce for their invaluable comments regarding Lana and its exception model. A. Romanovsky is partially supported by EPSRC/UK DOTS Project. G. Di Marzo Serugendo is supported by Swiss NSF grant 21-68026.02.

References

1. C. Bryce, C. Razafimahefa, and M. Pawlak. Lana: An approach to programming autonomous systems. In B. Magnusson, editor, *ECOOP 2002 - Object-Oriented Programming*, volume 2374 of *LNCS*, pages 281–308. Springer-Verlag, 2002.
2. G. Cabri, L. Leonardi, and F. Zambonelli. Reactive tuple spaces for mobile agent coordination. In K. Rothermel and F. Hohl, editors, *Proceedings of 2nd International Workshop on Mobile Agents*, volume 1477 of *LNCS*, pages 237–248. Springer-Verlag, 1998.
3. P. Ciancarini, A. Omicini, and F. Zambonelli. Coordination Technologies for Internet Agents. *Nordic Journal of Computing*, 6(3), 1999.
4. G. Di Marzo Serugendo and A. Romanovsky. Designing fault-tolerant mobile systems. In G. Reggio N. Guelfi, E. Astesiano, editor, *Scientific Engineering for Distributed Java Applications. International Workshop, FIDJI 2002*, volume 2604 of *LNCS*, pages 185–201. Springer-Verlag, 2003.
5. M. Diaz, B. Rubio, and J. M. Troya. Distributed Programming with a Logic Channel-based Coordination Model. *Computer Journal*, 39(10), 1996.
6. J. B. Goodenough. Exception Handling: Issues and a Proposed Notation. *Communications of the ACM*, 18(12):683–696, 1975.
7. G. P. Picco and G. Catalin-Roman. Lime: Linda meets mobility. In *Proceedings of International Conference on Software Engineering*. IEEE Computer Society Press, 1999.
8. A. Romanovsky and J. Kienzle. Action-oriented exception handling in cooperative and competitive con-current object-oriented systems. In A. Romanovsky, C. Dony, J. Lindskov Knudsen, and A. Tripathi, editors, *Advances in Exception Handling Techniques*, volume 2022 of *LNCS*, pages 147–163. Springer-Verlag, 2001.
9. A. Rowstron. Mobile co-ordination: Providing fault-tolerance in tuple space based co-ordination languages. In P. Ciancarini and A. Wolf, editors, *Coordination Languages and Models: Coordination99*, volume 1594 of *LNCS*, pages 196–210. Springer-Verlag, 1999.