# On Rigorous Design and Implementation of Fault Tolerant Ambient Systems

Alexei Iliasov[1], Alexander Romanovsky[1], Budi Arief[1], Linas Laibinis[2], and Elena Troubitsyna[2]

[1] School of Computing Science, Newcastle University, Newcastle upon Tyne NE1 7RU, England.
{ Alexei.Iliasov, Alexander.Romanovsky, L.B.Arief }@newcastle.ac.uk
[2] Department of Information Technologies, Aabo Akademi University, 20520 Turku, Finland.
{ llaibini, etroubit }@abo.fi

**Abstract.** Building fault tolerant ambient systems is an even more challenging task than building fault tolerant distributed systems, as their components are loosely synchronised, can move, become disconnected, and their behaviour may depend on the changing context. In this paper, we demonstrate how to build such a system using an ambient lecture scenario. The approach we are putting forward relies on using a combination of formal methods applied for rigorous development of the critical parts of the system and a set of design abstractions proposed specifically for the ambient applications and supported by a special middleware. Our middleware provides system structuring through the concepts of roles, agents, locations and scopes, making it easier for the developers to achieve fault tolerance. In addition to these, the middleware facilitates recovery by detecting a set of predefined abnormal situations (such as disconnections). The paper outlines our experience in developing the lecture scenario using this approach, focusing specifically on designing and implementing the fault tolerance mechanisms.

## 1 Introduction

Building advanced methods and mechanisms for developing ambient systems and applications is a very active area of research as these systems are now used in various critical domains, such as health, transport, emergency and production. Many of these systems will rely on the *mobile agent paradigm*, which supports structuring systems using decentralised and distributed entities cooperating to achieve their individual aims. These systems have a number of characteristics complicating their development and making it difficult for the developers to meet stringent requirements. Firstly, a vast majority of emerging ambient systems and applications have mobile elements, such as code, devices, data, services and users. Secondly, such systems need to be context-aware, so that the system activities can be directly influenced by the information representing their changing environment (due to the component mobility or to changing characteristics of the physical world in which the systems are executed). Thirdly, these systems are open, in a sense that components can appear and disappear (e.g., become disconnected). Therefore, developers of such systems need certain abstractions and middleware for supporting component mobility, context-awareness, and system openness. In addition to these, due to the large number of components and the decentralised nature of these systems, the developers need to ensure system flexibility and scalability.

Fault tolerance is becoming a very important aspect of these systems due to (i) their complexity and openness, which make it impossible to avoid or remove faults altogether; (ii) a high heterogeneity of components, modes of operation and requirements, all of which can cause architectural mismatches; (iii) non-professional users; (iv) a high level of reliance society puts on such systems, trusting them to perform a wide range of everyday critical functions: these systems are rapidly becoming business- and safety-critical; and (v) the expected wide proliferation of these systems and the impact this will have on our society.

Unfortunately, the existing frameworks for developing mobile, context-aware and open ambient systems do not provide adequate support for achieving fault tolerance. The main difficulties here are due to component (agent) mobility and their autonomy. Mobility creates new challenges for ensuring system fault tolerance [1] as – in addition to all kinds of faults found in sequential, concurrent and distributed systems – mobile agents are susceptible to a number of unique faults and potentially-harmful situations due to recurrent disconnections and re-connections. Agent autonomy and anonymity, asynchronous communication, and system openness also call for the development of novel structuring mechanisms as the existing agent systems do not support recursive system structuring using units of agent cooperation, which can serve as units of error confinement and system recovery.

Fault tolerance mechanisms for ambient systems can be created at different levels: hardware, operating system, middleware or application. In our work, we are focusing on both the middleware and the application levels fault tolerance, supporting mainly forward error recovery through the systematic use of exception handling [2]. Although there have been several mechanisms proposed for general message-passing systems (such as [3] and [4]), this area has not received enough attention in the context of coordination-based systems even though it can provide the most efficient and effective recovery from a wide variety of errors and faults such as environmental faults/deficiencies, component, architectural and organisational mismatches and system-level inconsistencies, potentially damaging changes in components and environments, and degradation of component services.

In our previous work, we introduced a *Context-Aware Mobile Agents* (CAMA) system [5], which provides fault tolerance in mobile agent applications through agent structuring. This is achieved by using the concepts of roles and scopes, explicit and consistent exception handling at the level of scopes, and a specialised distributed middleware for detecting disconnections and raising exceptions at the scope level. The design of this system and its middleware implementation has been strongly influenced by Linda [6], which defines a set of language-independent coordination primitives that can be used for autonomous and asynchronous communication and coordination between several independent pieces of software.

The overall aim of our research is to propose and evaluate a rigorous method for developing fault tolerant ambient applications [7]. Introducing the CAMA abstractions and the accompanying middleware is part of this work. In addition to this, we are working on applying formal methods and tools in order to ensure a clean system design and an early verification of the system models [8, 9]. The main reason for this is that it is becoming clear for us that applying advanced fault tolerance mechanisms is itself a complex task that often results in misuse and mistakes (see some supporting evidence in [10] and [11]). To help in avoiding them – where possible – we investigate the use of formal fault tolerance methods focusing specifically on employing adequate tool support and modelling fault tolerance patterns.

This paper shows how we applied formal modelling methods and tools, formal decomposition patterns, along with the CAMA abstractions and middleware during a systematic and rigorous development of fault tolerant ambient applications in the university campus domain. More specifically, we are using an ambient campus lecture scenario [12], the development of which is part of the IST RODIN project [7]. In this scenario, we assume that each classroom is a location with wireless support, in which a lecture can be given. Our aim is to develop a system supporting a number of functionalities to be conducted by the teacher and the students during a lecture. The teacher software is run on a desktop computer available in the classroom, while the student software is run on PDAs (each student is given a PDA). During the lecture, the students can be involved in a group work where the teacher – through the application's graphical user interface – puts the students into several groups. The groups are mutually exclusive, i.e., a student cannot belong to more than one group. Within each group, the students work collaboratively on a given task. As an example shown in this paper, the students are asked to develop C programs. An important part of this application is a subsystem that supports *shared editing*, allowing multiple students to modify the same piece of C code in a consistent manner within a distributed setting.

The rest of the paper is organised as follows. Section 2 introduces our formal framework – the B method [13]. Section 3 outlines the CAMA system – including its main abstractions and support for fault tolerance, explains the implementation of the middleware and shows how CAMA agents are constructed. Section 4.1 covers the formal development of the ambient lecture using the B method, along with the application of the *decomposition patterns* developed for assisting such a formal approach. The implementation of the ambient lecture application is detailed in Sections 4.2 and 4.4. Section 4.3 reports on the fault tolerance mechanisms integrated into this application. Section 5 concludes the paper and outlines our future work.

## 2 Formal Modelling and Refinement in B

A *formal specification* is a mathematical model of the required behaviour of a (part of a) system. In B, a specification is represented by a collection of modules called *Abstract Machines.* In this paper, we use an extension of B, called EventB [14], which enables modelling of distributed, parallel, and reactive systems.

An abstract machine encapsulates a local state (local variables) of the machine and provides operations (events) on the state. A simple abstract machine can be seen in Fig. 1.

A machine is uniquely identified by its name *AM*. The state variables of the machine, $v$, are declared in the **VARIABLES** clause and initialised in *INIT* as defined in the **INITIALISATION** clause. The variables types are given in the **INVARIANT** clause. The invariant also defines the properties of the system that should be preserved during system execution.

**MACHINE** *AM*
**SETS** *TYPES*
**VARIABLES** $v$
**INVARIANT** $I$
**INITIALISATION** *INIT*
**EVENTS**
$E_1 \;=\; \ldots$
$\ldots$
$E_N \;=\; \ldots$
**END**

**Fig. 1.** Abstract B machine

Operations $E_1, \ldots, E_N$ of the machine are defined in the **EVENTS** clause. An event $E$ is defined as follows:

$$E = \textbf{WHEN} \;\; g \;\; \textbf{THEN} \;\; S \;\; \textbf{END}$$

where the guard $g$ is a predicate over the state variables $v$, and the body $S$ is a B statement describing how $v$ are affected by the event. Several events can be grouped together in an *array of events AE*:

$$AE = \textbf{ANY} \;\; i \;\; \textbf{WHERE} \;\; C(i) \;\; \textbf{THEN} \;\; S \;\; \textbf{END}$$

where $i$ is a list of local distinct indices, *C(i)* is a list of array conditions, and $S$ is the body of the event.

The occurrence of events represents the observable behaviour of the system. The event guard defines the conditions under which the body can be executed. The B statements that we use to describe the body of an event is a mixture of executable statements (e.g., assignments or conditional statements) and abstract statements using mathematical operations over sets and functions.

The B method supports the top-down development paradigm. In the development process, the abstract specification is transformed into a system implementation via a number of correctness-preserving steps called *refinements*. Refinements allow us to gradually incorporate concrete implementation details, while at the same time preserving the previously stated properties of the system. The correctness of each refinement step is validated by proofs. As a result, we get an executable system that is correct by construction.

The tools support available for B – for example, Atelier B [15] – provides some assistance to the entire development process. Atelier B has facilities for automatic verification and code generation, as well as documentation, project management and prototyping. All formal developments presented in this paper have been completely verified using Atelier B.

## 3 Context-Aware Mobile Agents (CAMA)

CAMA is both a framework and a middleware supporting the development and deployment of agent-based applications. As a framework, CAMA encourages disciplined development of fault-tolerant mobile agent applications by supporting a set of abstractions ensuring system structuring, exception handling and openness. We have implemented this framework as a middleware that can be used for supporting effective and highly scalable mobile applications, while guaranteeing agent compatibility and dependability. This section provides a brief introduction to CAMA – a more detailed description can be found in [5].

### 3.1 CAMA Abstractions

The three basic concepts which CAMA offers for system structuring are *agent*, *platform* and *location*. Agents represent the basic structuring unit in CAMA applications and they are the active entities of a CAMA system. Each agent is executed on a platform; several agents may reside on a single platform. A platform provides an execution environment for agents, as well as an interface to the middleware. A platform is typically run on a PDA, a smartphone, or a laptop. A location is the core part of any CAMA system as it acts as the middleware that provides a means for communication and coordination among agents which are situated within the range of the location middleware (connections are typically conducted over wireless networks, with wireless hotspots providing access to the location middleware).

A location is also a container for *scopes*. A scope structures the activity of agents in a specific location and provides an isolation of several communicating agents, thus structuring the communication space. Scopes are dynamically created when the entry conditions defined in the scope specification are met. Agents can cooperate only when they are participating in the same scope. Nested scopes are used to structure large multi-agent applications into smaller parts which do not require the participation of all agents. Such structuring has a number of benefits. It isolates agents into groups, thus enhancing security. Scope structuring is also crucial for developing fault tolerant applications as it links the coordination-space structuring with activity structuring, which supports error confinement, localised error recovery and scalability.

To deal with various functionalities that any individual agent provides, CAMA introduces agent *role* as a finer unit of code structuring. Each agent has one or more roles associated to it. A role is a specification of one particular functionality of an agent. A composition of all agent roles forms its specification. An agent participates in a scope by assuming one of the roles available for that scope. The scope definitions include specifications of the roles from which the scope is composed. The role specifications determine the roles available in the scope, and the number of agents allowed to take part under any given role in that scope. In other words, a role is a structuring unit of an agent, and it is an important part of the scoping mechanism. It allows a dynamic composition of multi-agent applications and ensures agent interoperability by enforcing the developers of roles and agents to conform to the role specifications.

### 3.2 CAMA Middleware

CAMA middleware is composed of two major components: the *location middleware* (which provides the coordination service) and the adaptation layer for Java language[3] called `jcama`. The location middleware is implemented in C language, which allows us to achieve the best possible performance of the coordination space and to effectively implement numerous features, such as the scoping mechanism. The location middleware implementation is quite compact – it consists of approximately 7500 lines of C code and should run on most Unix platforms. We have so far tested it on Linux FC2 and Solaris 10. The `jcama` adaptation layer defines several classes for representing – among others – the abstract notions of location, scope and coordination primitives. It also provides an interface through which mobile agents or applications can be developed easily. The current implementation of `jcama` is fairly small in size (around 40Kb) and it can be used with both standard Java and J2ME. The full implementation of the location middleware and the `jcama` adaptation layer are available at SourceForge [16].

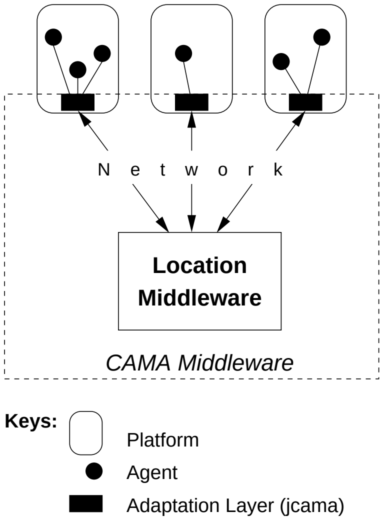


**Fig. 2.** CAMA-based system structure

[3] We use Java for developing the applications for PDAs.

A diagrammatical representation of a CAMA-based system structure can be seen in Fig. 2. Each platform carries a copy of the `jcama` adaptation layer. Agents residing on a platform use the features provided by `jcama` to connect over the network (could be wired or wireless) to the location middleware.

It is possible to extend the functionality of the CAMA middleware by adding *plugins* to the location middleware. These plugins run as special agents with greater access to the environment, enabling, for example, file management, additional communication features and connections to external systems. By using plugins, developers can customise the middleware by adding extra features that are necessary to the system, and to structure the system better. We have so far implemented plugins for *exception handling*, *mobility* (to allow agents to move from one platform to another) and *collocation* (which provides a location-attached execution environment for Java agents and is used in conjunction with the mobility plugin).

Although the middleware and the adaptation layer are complex and functionally rich, they are easy to use. The Java reflection mechanism helps the developers to write agents in a readable and compact way by hiding most of the communication-related details. Fig. 3 shows an example of a complete agent written in Java using the `jcama` library. This program allows agents within the same scope to communicate (chat) to each other. The agent is made of a single *reaction* – a piece of code that responds to a certain event – which listens for messages in a chat application scope. For any message it receives, it produces a copy of the message starting with ”someone said ”, unless the message already has the string ”someone said ” as its prefix.

```
import uk.ac.ncl.csr.jcama.*;
import uk.ac.ncl.csr.cama.extensions.discovery.*;
public class ChatBot extends RoleSkeleton {
  public void reactionMessage(String msg) throws CamaException {
    if (!msg.startsWith(”someone said ”))
      post(”Message”, new Record().addString(”someone said ” + msg));
  }
  public Chat(Scope s) {
    super(s, ”chat”);
  }
  public static void main(String argv[]) throws CamaException {
    Scope lambda = DiscoverLocations.AutoConnect(name).lamda();
    new ChatBot(lambda.JoinExisting(”Chat”, ”chat”));
  }
}
```

**Fig. 3.** A complete and fully-functional chat bot agent in CAMA

### 3.3 Design for Fault Tolerance

The CAMA framework supports application-level fault tolerance by providing a set of abstractions and a supporting middleware that allow developers to design effective error detection and recovery mechanisms. The main means for implementing fault tolerance in CAMA is a novel exception handling mechanism which associates scopes with the exception contexts. Scope nesting provides recursive system structuring and error confinement, as information cannot be passed outside such scopes. In effect, the execution of a scope is atomic from an outside observer's point of view.

Error recovery in CAMA systems is application-specific by nature and is to be implemented by the role developers. Error recovery is typically conducted at the level of individual scopes with an aim to recover the activity of this scope, although it is possible to attach handlers to individual roles (we usually do not use this feature as it breaks the abstraction levels). CAMA allows the developers to define cooperative recovery involving some or all roles of a scope when an error is detected in this scope. After detecting an error, any role can initiate application-specific recovery at the scope level.

A rich set of predefined exceptions provided by CAMA is useful for writing applications which react to abnormal situations detected by the CAMA middleware (Fig. 4). There are two types of abnormal situations: the ones which are propagated to all scope roles which are subscribed to them (including connection-disconnection exceptions, such as `CamaExceptionDisconnection`) and the local exceptions propagated to an individual role when it tries to execute an illegal action (e.g., violation of the scope constraints exceptions, such as `CamaExceptionClosed`).

| *Events* | *Exceptions* |
|---|---|
| NewScope | CamaExceptionClosed |
| Destroy | CamaExceptionInvalidReqs |
| Join | CamaExceptionNoRights |
| Leave | CamaExceptionInvalidRole |
| Disconnection | CamaExceptionDisconnection |

**Fig. 4.** Some system events and predefined exceptions

A number of predefined middleware events allow an agent to track contextual changes, most importantly, changes in the set of visible agents and scopes. This is essential for initiating both cooperative and localised recovery. For example, after discovering a disconnection of another agent, an agent may initiate local recovery actions that put it into a state from which it can continue without the disconnected agent.

It is our ongoing work to make the approach initially presented in [8] more suitable to developing fault tolerant open multi-agent applications. Compared with the work outlined in [8], our current framework imposes less restrictions on the agents during exception handling, in particular, an agent does not have to be involved in exception handling at all, if this does not suit its aims. This makes exception handling not only anonymous and asynchronous, but also voluntary, making it very different from the classical atomic action schemes (such as that proposed in [17]).

Many researchers realise now that fault tolerance is becoming a software engineering concern which needs to be addressed at various development steps. Finding the right balance between using early and late development step techniques is a difficult issue. In this paper, we show how formal models and implementation level techniques can be used in combination. Formal modelling and verification of applications typically help in eliminating a number of errors that otherwise would have to be addressed at the implementation stage. As part of our work, we investigate (i) how error detection and recovery can be integrated into formal development, (ii) how formal models can be used by extracting from them information about undesirable behaviour to incorporate error detection and recovery actions in the implementation, and (iii) how recovery can be introduced at the level of agent construction. When fault tolerance is integrated into formal models, it becomes an integral part of the system, so that fault tolerance properties are verified and satisfied during system development. In order to use formal models for incorporating fault tolerance into system implementation, we need to define the undesirable behaviour as an action or a set of actions which break the model invariant or one of the post-conditions. This helps a developer to include, at the implementation step, an additional code for recovering from the undesirable behaviour. The formal approach we are using defines a set of roles which are interoperable by construction. During system implementation, agents are constructed as configurations of several roles. This approach clearly requires agent-level error detection and recovery to be introduced during system implementation.

### 3.4 Agent Construction

A typical CAMA agent is composed of a number of simple building blocks. The overall structure of a CAMA agent is shown in Fig. 5. The discovery part is responsible for finding a location and connecting to it. Once an agent is connected to a location, it decides which application scopes to join or to create. An agent can have *physical mobility* (due to the physical movement of the hosting device) and *logical mobility* (when it changes its hosting platform). Any non-trivial agent has a monitor which oversees its context, which changes during both physical and logical mobility. The agent actions responsible for migration are put into a separate part. There are also the implementations of agent roles and the units for coordinating the roles.

The bulk of the services provided by the middleware is implemented as calls to the location service. Hence, the first action of an agent is to associate itself with one or more locations. There are two mechanisms for this: the automated location discovery based on UDP multicasting and the use of a fixed location address. The middleware also provides a simple connection method which takes a location service address as its argument. This way of connecting is not very useful in dynamic environments, for example when an agent physically moves with a PDA owner. During the lifetime of an agent, a set of available locations changes and there may be no way of determining their addresses. To overcome this problem, the middleware provides a multicast-based discovery mechanism. The mechanism supports local-area networks such as LAN, WLAN and BluetoothPAN.

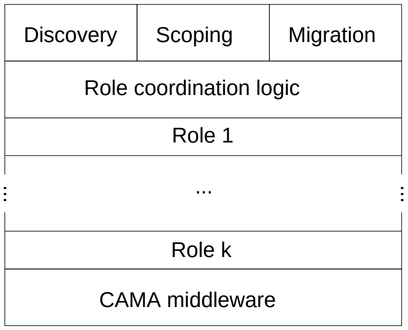


**Fig. 5.** Agent subcomponents

A location permits communication only among agents connected to the same scope. An agent must join a scope before it can send its messages. Scopes are created and deleted dynamically. The scope name can be passed as (a part of) a message from one agent to another. All location services have a predefined scope called *lambda* that is used only for creating other scopes.

An agent can discover all the existing scopes with a given scope type. Since a scope type is explicitly specified by an agent, an agent cannot see a scope of an unknown type. If there is more than one scope with a matching type, an agent can choose which one to join. The middleware provides an utility method to join a random scope of a given type with a specified role.

If an agent is unable to find a suitable scope, it may create a new scope. This situation happens when there are no scopes of a required type or when all the existing scopes are full and cannot accommodate a new agent. An agent initiating a scope creation becomes the owner of that scope. It can delete its scope or make it visible to other agents. A scope is deleted automatically when its owner disconnects or leaves.

Typically, an agent does not have full control over its physical mobility. It should, however, provide actions for compensating undesirable effects of physical mobility and possibly take advantage from the appearance of new agents. The middleware has a special service for tracking changes in the agent context. An agent can subscribe to events announcing appearance and disappearance of other agents, scope creation and so on (see Fig. 4).

An autonomous agent uses logical mobility to transfer itself from one platform to another. Although it is not a native CAMA functionality, logical mobility is easy to use thanks to the mobility extension in `jcama` and the mobility plugin provided by the location service.

The essence of agent systems is the ability to form multi-agent applications where agents can interact with each other. We use the *role* concept to structure agent so that it can cooperate with other agents as cooperation can occur only among roles. A role also provides a link between the formal development using the B method and agent implementation. The B models of roles are used to implement roles. The same B model can be used to produce several role implementations as the developer has to resolve non-determinism in a model by making specific implementation decisions. A role model is a set of operations updating variables. Some of these operations can be invoked externally while the others are activated only within a role model. The difference in the style of operation invocation leads to the important classification of operations into role reactions and role actions. Reactions and actions are implemented in different ways, although within the B formalism we do not have to distinguish them.

## 4 Group Work in Developing C Programs

This section shows how we use the B method in combination with the CAMA system for developing robust and dependable ambient applications. To demonstrate this, we pick the ambient campus lecture scenario described in [12]. We do realise the limitations of formal methods, in particular, in the complexity of conducting this work and in the limitations of the existing tools. This is why we have chosen to develop formally only one specific part of the system. This part supports the most complex functionality of the application, that is the group work using a shared editor. This application allows several students – by using a mobile device such as a PDA or a smartphone – to collaboratively work on a C program, editing it in turn while keeping the consistency of the content. After all of them have made their corrections and agreed on the text, this file can be shown to the teacher, pretty printed, or sent to the compiler (which is located on a separate computer). To support shared editing, we have chosen an implementation that

```
MACHINE coaccess
SETS AGENT
VARIABLES
  owner, agents, done
INVARIANT
  $owner \subseteq AGENT \wedge agents \subseteq AGENT \wedge done \in BOOL \wedge owner \subseteq agents \wedge card(owner) \leq 1 \wedge$
  $(done = FALSE \Rightarrow card(owner) = 0) \wedge (done = TRUE \Rightarrow card(agents) > 1)$
INITIALISATION...
EVENTS
  AddAgents = ...
  ChangeOwnership =
    ANY aa WHERE $aa : agents \wedge aa \neq \{owner\} \wedge card(owner) > 0 \wedge done = TRUE$
    THEN $owner := \{aa\}$ END;
  AssignLeader = ...
END
```

**Fig. 6.** The abstract machine specifications for *coaccess*

uses the token ring mechanism. This solution seems to be the most appropriate and effective considering the small number of students in each group (typically between four and six students).

### 4.1 Formal Development

One of the main goals of this paper is to demonstrate how formal modelling and tools can be applied for rigorous development of fault tolerant ambient applications. This development approach allows us to ensure essential properties of multi-agent applications, such as agent interoperability and fault tolerance by developing systems that satisfy these properties by the way we construct them. In this section we present the formal development of the shared editor application described above.

In our approach, the application development starts with a abstract specification of a scope – a mathematical model of the required behaviour of a multi-agent application. In the refinement process, we incorporate implementation details concerning concrete functionality, communication, and fault tolerance aspects of the involved agents. If a scope describes the activities of more than one role, at some point of the development, a scope specification should be decomposed into the corresponding specifications of the involved roles. The resulting specifications can then be developed and implemented separately.

For the shared editor application, we have identified two different scopes: a scope for modelling file operations (the *Filesystem* scope), and a scope for describing the shared editing of a single file (the *Editing* scope). The initial specifications of these scopes provide the starting points for two separate formal developments.

The *Editing* scope basically models mutually exclusive access to a shared resource. We are going to implement it as a token ring mechanism. The initial specification of the scope, called *coaccess*, is presented in Fig. 6. The specification allows new requests from the agents to a shared resource to be accepted (in *AddAgents*); a resource to be given to one of the agents (in *AssignLeader*); and a change of resource ownership to be made (in *ChangeOwnership*). The latter event is possible only when all of the requests have been received (i.e., $done = TRUE$) and no agent currently owns the resource ($aa \neq \{owner\}$).

**Scope Specification and Decomposition into Roles.** The initial scope specification defines the scope state (program variables) and the scope events (operations) that update the state. It abstractly describes the general scope activities without specifying which roles are involved in them. However, at some stage of the specification refinement, we eliminate the abstract scope variables and introduce the roles. The scope state is then partitioned by distributing the program variables among the scope roles such that for each variable, there is exactly one role responsible for updating it. Similarly, for the operations, we specify the scope events in such a way that each event updates the variables of only one role. As a result, we attribute each scope operation with a single role. At the same time, a scope operation can read the variables of other roles. This gives us an abstract way for modelling the coordination among the scope roles. As a result of our final refinement step, the scope specification is decomposed into separate role specifications that can then be used to implement compatible cooperative agents.

Therefore, the goals of our development process by program refinement are two-fold. We introduce the missing implementation details on specific functionality, communication, and fault tolerance mechanisms.

```
REFINEMENT faremote
REFINES fileacess
VARIABLES
  ⟨ variables of file server ⟩
  rfiles, outbuffer, req_cmd, req_id, ...
 ...
EVENTS
  ⟨ events of file server ⟩
  NewFile = ...
  AddBlock = ...
  ReplaceBlock = ...
  ReadFile = ...
  Submit = ...
END
```

```
REFINEMENT fserver
REFINES faremote
VARIABLES
  rfiles, outbuffer, req_cmd, req_id, ...,
  ⟨ additional variables of file user ⟩
  creq_cmd, creq_id, creq_block, ...
 ...
EVENTS
  NewFile = ...
  AddBlock = ...
  ...
  ⟨ additional events of file user ⟩
  ReqCreate = ...
  ReqRead = ...
  ReqWrite = ...
  ReqReWrite = ...
END
```

**Fig. 7.** The specifications for *faremote* and *fserver*

At the same time, we decouple the scope state and operations in such a way that we can attribute each scope variable and event to a specific role.

Next we explain how we can develop the specifications of our identified scopes (*Filesystem* and *Editing*) into the corresponding specifications of the involved roles. The development process is driven by the application of the so called refinement patterns.

**Refinement Patterns.** In general, there are many possible ways to refine a particular specification, thus arriving at different implementations. In order to cope with the overall complexity of such formal development, we carry out changes in a well-defined way, applying specific *refinement patterns*. These patterns focus on specific transformations that introduce particular features (like communication or fault tolerance mechanisms) into our software models. As their name suggests, the refinement patterns also enable the specification to be reused. In our formal development, we distinguish the following types of refinement patterns [9]:

- patterns for role decomposition (decoupling), allowing us to modify the data and operations in such a way that they become distributed among the involved roles;
- patterns for introducing communication between roles;
- patterns for introducing fault tolerance mechanisms.

Let us now consider a simple refinement pattern. In our development of the *Filesystem* scope, we start with an abstract specification describing the basic file access operations (specification *fileacess*). This scope describes co-operation of two roles: a file server and a file user. Therefore, our goal is to refine the scope specification in such a way that, at some point of the development, it would be possible to decompose it into the corresponding specifications of a file server (the *FileServer* role) and a file user (the *FileUser* role).

For this purpose, we use one of the role decomposition patterns proposed earlier [9]. It allows us to develop a scope, treating all its data and operations as belonging to the main (managing) role. Then, in a separate refinement step, we introduce new data and operations of a secondary role, tying them together with the corresponding data and operations of the main role. The operation guards and the invariant are modified in such a way that scope activities involving both roles are carried out only following a certain predefined scenario.

In the *Filesystem* scope, the main role is *FileServer* and the secondary role is *FileUser*. The refinement step, which applies the described pattern, takes specification *faremote* and produces specification *fserver*. The structure of the *faremote* and *fserver* specifications is presented in Fig. 7. The resulting specification *fserver* is now ready to be decomposed into the corresponding specifications, specifying the roles of *FileServer* and *FileUser* (see Fig. 8).

The alternative role decomposition pattern allows us to actually split both the data and the abstract operations among the involved roles. More details about the decomposition patterns can be found in [9].

Using the refinement patterns described above, we have developed the identified scopes of the shared editor application, arriving at the specifications of the involved roles. As a result, we have decomposed

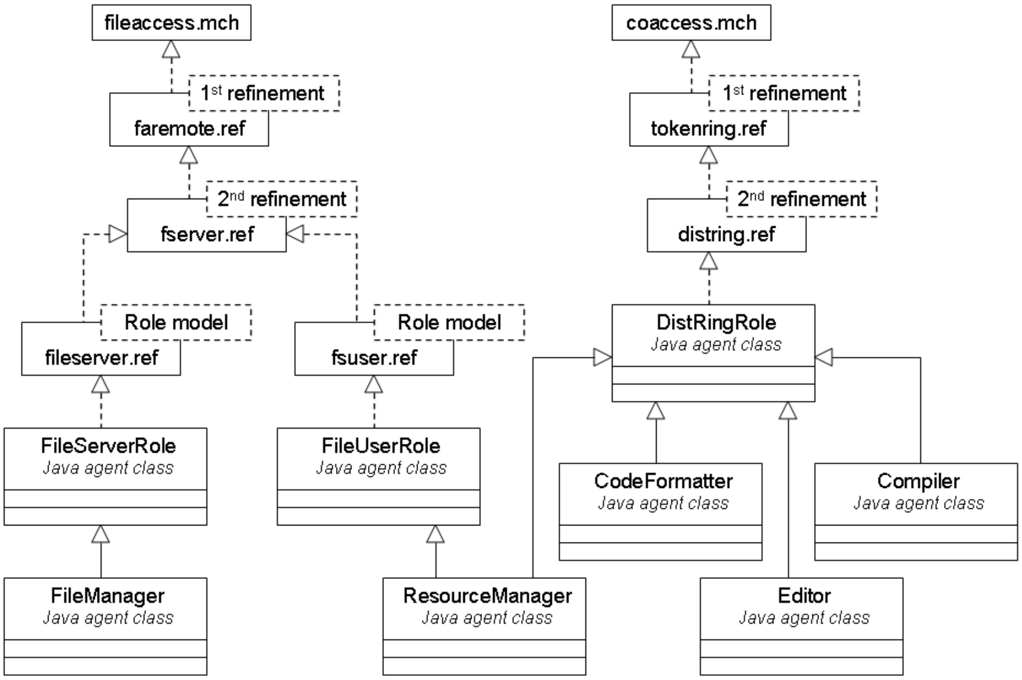


**Fig. 8.** The overall picture of the *shared editor* application development

the *Filesystem* scope into two separate roles – *FileServer* and *FileUser*. The development of the *Editing* scope has produced a single role, *DistRing*, representing the distributed token ring. The whole application development structure is shown in Fig. 8.

In the next section we discuss how we can use these developed models to construct different types of cooperating agents involved in the shared editor application.

### 4.2 Constructing the Agents for the Shared Editor Application

In our formal development, we have designed two scopes (*Filesystem* and *Editing*) with three different roles (*FileServer*, *FileUser*, and *DistRing*). The next stage is to construct a running application based on these scopes and roles models. This resulted in the construction of five agents: *FileManager*, *ResourceManager*, *CodeFormatter*, *Compiler* and *Editor* agents. Fig. 9 illustrates how these agents are structured within the scopes mentioned above.

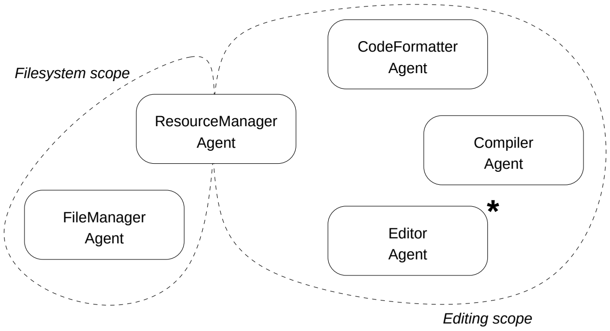


**Fig. 9.** The scopes and agents for the *shared editor* application

The *ResourceManager* agent combines two roles and acts as a link between the two scopes. The *CodeFormatter*, *Compiler* and *Editor* agents implement the same role (*DistRing*), but they each provide different functionality on top of the role model.

Fig. 10 gives the overall structure and some details of implementation of the *DistRing* role model. The methods with the *reaction* prefix are the role events (we do not use 'event' prefix to avoid possible confusion with unrelated Java libraries) and the methods with the *action* prefix are the role actions. The middleware uses Java introspection capability to find all the methods starting with a particular prefix. When an instance of a role is created, the middleware analyses the class and automatically creates event subscriptions for each role event. For example, the declaration of `reactionChangeOwnership` method will result in a subscription of events that match the following LINDA-style pattern: {`'ChangeOwnership', String: *`}. The first field is the event name and the remaining fields are type-constrained wildcards created from the list of formal event arguments. Hence, an event is matched by its name and also by the number and the types of the parameters.

```
public class DistRingRole extends RoleSkeleton {
  private Scope scope;
  private boolean ringready = false;
  private boolean isowner = false;
  private String myLeftNeighbour = null;
  private Listener listener = null;
  public DistRingRole(String myname, Scope s, Listener list) throws CamaException { ... }
  public void actionJoinRing() throws CamaException { ... }
  public void actionRingReady() throws CamaException { ... }
  public void actionChangeOwnership() throws CamaException {
    if (ringready && isowner) {
      isowner = false;
      post("ChangeOwnership", new Record().addString(myLeftNeighbour));
      listener.event(Record.empty, this);
  }}
  public void reactionSetNeighbour(String node, String neighbour) throws CamaException { ... }
  public void reactionRingReady() throws CamaException { ... }
  public void reactionChangeOwnership(String nodename) { ... }
```

**Fig. 10.** Implementation of *DistRing*

An event method is invoked whenever a matching event is posted. The middleware extracts the values supplied with the event and feeds them as the event method arguments. An event is created by an action or a reaction of a role in the current scope. The `actionChangeOwnership` method demonstrates the use of the *post* which creates new events. This particular statement triggers `reactionChangeOwnership` events in all the roles of all the agents in the current scope with the current value of `myLeftNeighbour` as the argument.

Role implementations for the case study use exclusive execution model for reactions and actions. Only one action or reaction can be running at any given moment. This is crucial for preserving the properties of the formal model as they are proved under the assumption of atomic operation execution. Threads and monitors are automatically managed by the middleware, thus the execution model appears very natural to a developer. We allow a role to invoke reaction on itself by creating an event. Actions, however, cannot be invoked within a role due to the requirement of atomic execution of reactions and actions. The middleware uses a special mechanism to avoid role starvation from cyclic reaction invocations.

While reactions are managed by the middleware and invoked externally by other agents, actions must be taken care of by an agent developer. It is possible to have a role without actions at all. For example, the *FileServer* role does not contain any action and thus it can be implemented as a completely autonomous role. In general, however, an agent developer must write a code that manages a role. This is a deliberate methodological decision. We think it is important to have a well defined means of balancing formalisation efforts and implementation freedom. Role actions are like ports to which developers can attach their customised extension code, unspecified but foreseen by the formal design. It is a correct way of extending the role functionality without risking violating the role properties proved during the formal development.

A role action invocation is similar to calling a class method. An action can be invoked at any moment. However, the calling code might be blocked if there are other actions or reactions running at the same

```
public void event(Record values, Object role) {
  if (pred1(ρ)) { ... }
  else if (pred2(ρ)) { ... }
              ...
  else if (predn(ρ)) { ... }
}
```

**Fig. 11.** Specifying reactions on roles' variables change

time. To match the style used in the roles, agent developers will have to implement the role management code in a reactive manner. This requires implementation of a code reacting to the changes in the roles states, or in other words, a code for monitoring the role variables. Procedural languages typically do not support such a feature. It can be reasonably simulated by registering a callback method that requires a role to call it whenever a role variable is updated. The `listener` variable in Fig. 10 is a pointer to a callback procedure (which is an instance of an interface in Java). It calls the `actionChangeOwnership` action to notify the role management code that a role variable has been updated. The callback will usually have a structure as shown in Fig. 11. A callback is made-up of a number of actions, each activated by a predicate over the variables of one or more roles. This is a scalable and structured approach to coordinate any number of roles.

A complex agent is typically composed of more than one role. In many case, the roles must be orchestrated so that they work together towards the same global goal. Some changes in a state of one role can result in an action being called in another role. The *ResourceManager* agent from our case study is an example of an agent with such two roles. When a *ResourceManager* agent gets a token in the *Editing* scope, it can upload or download a file to/from the *FileServer* agent. The content of a downloaded file can be made available to other agents in the *Editing* scope.

### 4.3 Fault Tolerance

During the formal development of the case study we decided against introducing faults and recovery actions as we could not find recovery algorithms for the modelled problems which would be general enough to retain the desired level of abstraction. This means that we had to rely on implementation stage techniques to bring fault tolerance into the case study application. However, we consider it to be natural to approach the problem using the structures introduced by the formal design. Hence, to make the whole application more robust, we individually analyse the two parts of the application – the *Filesystem* scope and the *Editing* scope – and try to define possible faults and the corresponding recovery actions.

**Role-level Recovery.** The *FileSystem* scope has two roles: one for generating requests and reading the results; the other for serving these requests. One possible source of faults is malformed requests. *ResourceManager* might try to download a non-existent file or read beyond a file end. The information needed to validate these is private to the *FileServer* role. Thus, a request can be found invalid only after it triggers a reaction in the server part. The *FileServer* role cannot handle a malformed request in a normal circumstances. It must, however, inform the calling agent in order to avoid deadlocks since the calling agent may be expecting some results from the request. Effectively, an ability to handle malformed requests requires us to introduce new protocols of communication, not described in the formal model. We use the exception propagation technique to deal with new behaviour and new control flow in a disciplined manner. Access to a non-existent file will result in an exception sent back to the agent that generated the request. Each role implementation is extended in such a way that it is ready to accept an exception in place of normal events. The basic communication mechanism is the same for exceptions and normal events. The purpose of these exceptions and the exception handlers is to introduce recovery actions in such a way that the code for normal activities is not affected and the invariant of a role is not violated until an exception happens. Exception detection and the subsequent recovery create a temporary deviation from a normal behaviour which – provided that the recovery succeeds – at some point will rejoin the normal behaviour and restore the role invariant.

Exception detection for the *FileServer* role is based on the guards of the *fserver* model operations. These operation have guards in the form $sreq\_cmd = REQUEST \wedge \rho(v)$. The first part of the conjunction tests for a request type. The implementation of this part is implicit and each reaction is associated with a single request type. $\rho(v)$ is a predicate checking a request on well-formedness (e.g., if a file exists). The

```
public void reactionR$_i$(a) {
  if (error-condition(a, v)) {
    post(Exception$_k$, description);
  } else {
    ... // normal behaviour
  }
}
... // reactions and actions corresponding to normal behaviour
public void reactionException$_j$(d) {
  ... // recovery actions
}
```

**Fig. 12.** Extending role with error detection and recovery actions

negation of the predicate describes all invalid requests. We use this fact to introduce error detection by extending the corresponding role reactions with new branches that handle the same request but with a guard in the following form: $sreq\_cmd = REQUEST \wedge not(\rho(v))$. The new code signals an abnormal execution by sending an exception to the request producer. An exception is an event and the recovery actions can be constructed as standard role reactions (see Fig. 12).

**Agent-level Recovery.** The approach described above cannot be applied to the token ring role. Neither of the role instances knows about the global state. In fact, they know so little about the token ring as a whole that it is not possible to write any error detection predicates at all. Any attempts will result in extra communication with the neighbouring agents. We decided to avoid modifying the role model, so we introduced error detection and recovery at an agent-level.

The situation from which we want to recover-from is the disappearance of an agent from the token ring (*Editing*) scope. When an agent crashes, disconnects or deliberately leaves the scope, the ring is broken and the token will eventually be lost. The middleware has the capability to detect agent disconnections and inform all of the scope participants. Thus error detection is based on the services provided by the middleware. The corresponding recovery action must mend the existing ring or create a new valid ring with all the existing agents. We do the latter as we found it impossible to perform ring repairs without extending the functionality of the role.

Once an agent has learnt that another agent has disappeared, it terminates the current instance of the token ring role and starts a new one. This automatically initiates the construction of a new ring. This approach, although very simple, is a very efficient recovery method. Not only it recovers from the situations where one agent disconnects, but also successfully handles the situations where the ring is partitioned into two parts. The local recovery actions of all agents will result in the creation of two new functional rings, provided there are enough agents in each of them.

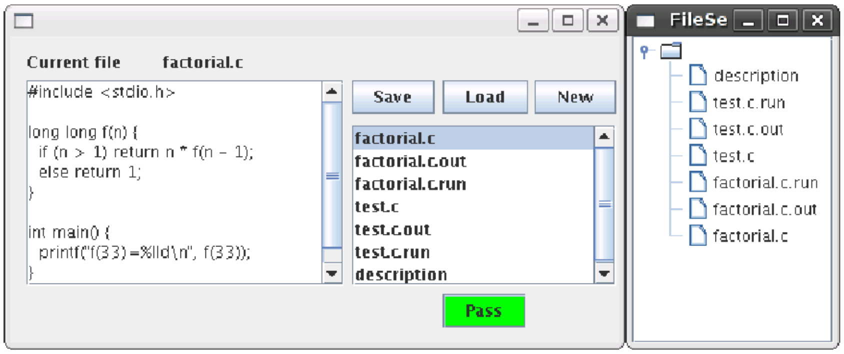
### 4.4 Implementation Details

We implemented the whole case study application in Java using the features provided by `jcama`. There are five different agents implementing three different roles (see Fig. 8 and Fig. 9). In order to simplify the transition from a formal specification into Java code, the implementation was done in two stages. In the first stage, we produced role implementations according to the formal specifications. The implementation of the roles was made generic enough to be reusable by different agent designs. The *FileManager*, *CodeFormatter*, and *Compiler* agents are non-interactive and completely autonomous. There can be multiple instances of the *Editor* agent.

The application was deployed on two PDAs, one smartphone, and two desktop PCs. The PDAs and the smartphone were running the *Editor* agent. Users of this agent (i.e. the students) should be able to move around, capitalising on the wireless connectivity of their devices.

The students – through their *Editor* agent – can type C programs, use an automatic code formatter (the *CodeFormatter* agent is based on the UNIX *indent* utility), compile and run the program, and see the run-time output. The right and the middle screenshots in Fig. 14 show the dialog windows of the *Editor* agent. The left-most screenshot shows a result from executing the C program through this agent.

All other agents reside on standard PCs running linux-2.6 and JDK-1.5. The smartphone is a Sony-Ericsson M600i running under Symbian 9.1 with CDC Java profile. The PDAs run Windows Mobile 2003

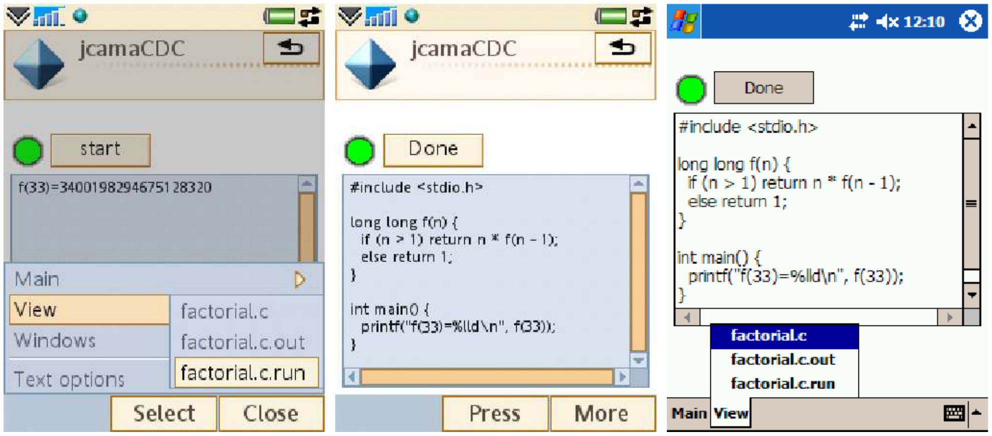


```
#include <stdio.h>

long long f(n) {
  if (n > 1) return n * f(n - 1);
  else return 1;
}

int main() {
  printf("f(33)=%lld\n", f(33));
}
```

**Fig. 13.** *ResourceManager* and *FileManager* agents running on a PC



```
f(33)=3400198294675128320
```

```
#include <stdio.h>

long long f(n) {
  if (n > 1) return n * f(n - 1);
  else return 1;
}

int main() {
  printf("f(33)=%lld\n", f(33));
}
```

```
#include <stdio.h>

long long f(n) {
  if (n > 1) return n * f(n - 1);
  else return 1;
}

int main() {
  printf("f(33)=%lld\n", f(33));
}
```

**Fig. 14.** *Editor* agent on a smartphone and a PDA

SE and IBM J9 Java machine, connected to the location middleware using a wireless LAN infrastructure. Smartphones connect to the location middleware through ad-hoc Bluetooth networking.

## 5 Conclusions and Future Work

The main contribution of this paper is in introducing a novel approach to developing fault tolerant ambient applications by using a combination of a formal method augmented by specialised development patterns and a set of design abstractions supported by a dedicated middleware. This approach has been successfully applied in developing the lecture scenario as part of a larger ambient campus system.

We have found formal methods to be very useful in allowing us to clearly define and rigorously develop in a stepwise fashion the most critical part of the application. Our experience suggests that it is useful to combine formal methods with the more commonly-used ways of building systems. In our work of developing the lecture scenario, we have identified and applied several ways of using them in combination.

Our ongoing and future work focuses on: (i) finalising the set of abstractions and the functionality of the middleware; (ii) building the complete development method supporting – in addition to the B refinement – verification (by model checking) of system properties with a specific focus on the fault tolerance properties; (iii) extending the exception handling mechanism with an ability to involve several scopes, to explicitly state and dynamically modify the exception propagation policies and to use exceptional events (reactions) to further separate normal system behaviour from the abnormal one.

## 6 Acknowledgements

This work is supported by the IST RODIN Project [7].

## References

1. Marzo, G.D., Romanovsky, A.: Designing Fault-Tolerant Mobile Systems. In Guelfi, N., Astesiano, E., Reggio, G., eds.: Scientific Engineering for Distributed Java Applications International Workshop, FIDJI 2002, Luxembourg, LNCS 2604. Springer (2003) 185–201
2. Cristian, F.: Exception Handling. In Anderson, T., ed.: Dependability of Resilient Computers. Blackwell Scientific Publications (1989) 68–97
3. Miller, R., Tripathi, A.R.: The guardian model and primitives for exception handling in distributed systems. IEEE Trans. Software Eng. **30**(12) (2004) 1008–1022
4. Pears, S., Xu, J., Boldyreff, C.: Mobile agent fault tolerance for information retrieval applications: An exception handling approach. In: ISADS, IEEE Computer Society (2003) 115–122
5. Arief, B., Iliasov, A., Romanovsky, A.: On Using the CAMA Framework for Developing Open Mobile Fault Tolerant Agent Systems. In: Proceedings of Software Engineering for Large-Scale Multi-Agent Systems (SELMAS) Workshop at ICSE 2006. (2006) 29–36
6. Gelernter, D.: Generative Communication in Linda. ACM Transactions on Programming Languages and Systems **7**(1) (1985) 80–112
7. Rodin: Rigorous Open Development Environment for Complex Systems. IST FP6 STREP project, http://rodin.cs.ncl.ac.uk/ (Last accessed: 24 Nov 2006)
8. Iliasov, A., Romanovsky, A.: Structured Coordination Spaces for Fault Tolerant Mobile Agents. In Dony, C., Knudsen, J.L., Romanovsky, A., Tripathi, A., eds.: Advanced Topics in Exception Handling Techniques. LNCS-4119 (2006) 182–201
9. Laibinis, L., Iliasov, A., Troubitsyna, E., Romanovsky, A.: Formal Approach to Ensuring Interoperability of Mobile Agents. Technical report, CS-TR-989, School of Computing Science, Newcastle University, UK. October (2006)
10. Bruntink, M., van Deursen, A., Tourwé, T.: Discovering Faults in Idiom-Based Exception Handling. In: Proceedings of ICSE 2006. 20-28 May 2006. Shanghai, China, ACM Press (2006) 242–251
11. Reimer, D., Srinivasan, H.: Analyzing Exception Usage in Large Java Applications. In: Proceedings of ECOOP 2003 Workshop on Exception Handling in Object-Oriented Systems. (2003)
12. Troubitsyna, E., ed.: Rodin Deliverable D18: Intermediate Report on Case Study Development. Project IST-511599, School of Computing Science, Newcastle University, UK (2006)
13. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (2005)
14. Metayer, C., Abrial, J.R., Voisin, L.: Rodin Deliverable 3.2: Event-B Language. Technical report, Project IST-511599, School of Computing Science, University of Newcastle (2005)
15. ClearSy: Atelier B: The industrial tool to efficiently deploy the B method. http://www.atelierb.societe.com/index_uk.htm (Last accessed: 24 Nov 2006)
16. Iliasov, A.: Implementation of Cama Middleware. http://sourceforge.net/projects/cama (Last accessed: 24 Nov 2006)
17. Campbell, R., Randell, B.: Error Recovery in Asynchronous Systems. IEEE Transactions on Software Engineering **12**(8) (1986) 811–826