Step-wise Development of Resilient Ambient Campus Scenarios

Alexei Iliasov, Budi Arief, and Alexander Romanovsky

School of Computing Science, Newcastle University,
Newcastle upon Tyne NE1 7RU, England.
{Alexei.Iliasov, L.B.Arief, Alexander.Romanovsky}@newcastle.ac.uk

Abstract. This paper puts forward a new approach to developing resilient ambient applications. In its core is a novel rigorous development method supported by a formal theory that enables us to produce a well-structured step-wise design and to ensure disciplined integration of error recovery measures into the resulting implementation. The development method, called AgentB, uses the idea of modelling database to support a coherent development of and reasoning about several model views, including the variable, event, role, agent and protocol views. This helps system developers in separating various modelling concerns and makes it easier for future tool developers to design a toolset supporting this development. Fault tolerance is systematically introduced during the development of various model views. The approach is demonstrated through the development of several application scenarios within an ambient campus case study conducted at Newcastle University (UK) as part of the FP6 RODIN project.

1 Introduction

We use the term *ambient campus* to refer to the *ambient intelligence* (AmI)[1] systems deployed in an educational setting (a university campus). Ambient campus applications are tailored to support educational, administrative and research activities typically found in a campus, including delivering lectures, organising meetings, and facilitating collaborations among researchers and students.

This paper reports our work on the development of the ambient campus case study within the RODIN project [1]. This EU-funded project, led by the School of Computing Science of Newcastle University, aimed to create a methodology and supporting open tool platform for the cost-effective rigorous development of dependable complex software systems and services. In the RODIN project, the ambient campus case study acted as one of the research drivers, where we investigated how to use formal methods combined with advanced fault-tolerance techniques in developing dependable AmI applications.

[1] A concept developed by the Information Society Technologies Advisory Group (ISTAG) to the EC Information Society and the Media DG, where humans are surrounded by unobtrusive computing and networking technology to assist them in their activities – http://cordis.europa.eu/ist/istag.htm.

Software developed for AmI applications needs to be able to operate in an unstable environment susceptible to various errors and unexpected changes (such as network disconnection and re-connection) as well as to deliver context-aware services. These applications tend to rely on the *mobile agent paradigm*, which supports system-structuring using decentralised and distributed entities (*agents*) working together in order to achieve their individual aims. Development of multi-agent applications poses many challenges due to their openness, the inherent autonomy of their components (i.e. the agents), the asynchrony and anonymity of their communication, and the specific types of faults they need to be resilient to. To address these issues, we designed a framework called CAMA *(Context-Aware Mobile Agents)*, which encourages disciplined development of open fault-tolerant mobile agent applications by supporting a set of abstractions ensuring exception handling, system structuring and openness. These abstractions are backed by an effective and easy-to-use middleware allowing high system scalability and guaranteeing agent compatibility. More details on CAMA and its abstractions can be found in [2–4].

The rest of this paper discusses the challenges in developing fault tolerant AmI systems (Section 2), describes the theory behind our design approach (Section 3), outlines various approaches to tackling fault-tolerant issues (Section 4), and illustrates how our design approach was applied in the case study scenarios (Section 5).

2 Challenges in Developing Fault-Tolerant Ambient Intelligence Systems

Developers of fault-tolerant AmI systems face many challenging factors, some of the most important ones are:

- *Decentralisation and homogeneity*
  AmI systems are composed of a number of independent computing nodes. However, while traditional distributed systems are *orchestrated* – explicitly, by a dedicated entity, or implicitly, through an implemented algorithm – in order to solve a common task, agents in AmI system make *independent* decisions about *collaboration* in order to achieve their individual goals. In other words, AmI systems do not have inherent hierarchical organisation. Typically, individual agents are not linked by any relations and they may not have the same privileges, rights or capabilities.
- *Weak Communication Mechanisms*
  AmI systems commonly employ communication mechanisms which provide very weak, if any, delivery and ordering guarantees. This is important from the implementation point of view as AmI systems are often deployed on wearable computing platforms with limited processing power, and they tend to use unreliable wireless networks for communication means. This makes it difficult to distinguish between a crash of an agent, a delay in a message delivery and other similar problems caused by network delay. Thus, a

recovery mechanism should not attempt to make a distinction between network failures and agent crashes unless there is a support for this from the communication mechanism.

- *Autonomy*
  During its lifetime, an agent usually communicates with a large number of other agents, which are often developed in a decentralised manner by independent developers. This is very different from the situation in classical distributed system where all the system components are part of a closed system and thus fully trusted. Each agent participating in a multi-agent application tries to achieve its own goal. This may lead to a situation where some agents may have conflicting goals. From recovery viewpoint, this means that no single agent should be given an unfair advantage. Any scenarios where an agent controls or prescribes a recovery process to another agent must be avoided.
- *Anonymity*
  Most AmI systems employ anonymous communication where agents do not have to disclose their names or identity to other agents. This has a number of benefits: agents do not have to learn the names of other agents prior to communication; there is no need to create fresh names nor to ensure naming consistency in the presence of migration; and it is easy to implement group communication. Anonymity is also an important security feature - no one can sense an agent's presence until it produces a message or an event. It is also harder to tell which messages are produced by which agent. For a recovery mechanism, anonymity means that we are not able to explicitly address agents which must be involved in the recovery. It may even be impossible to discover the number of agents that must be involved. Even though it is straightforward to implement an exchange for agents names, its impact on agent security and the cost of maintaining consistency usually outweigh the benefits of having named-agents.
- *Message Context*
  In sequential systems, recovery actions are attached to certain regions, objects or classes which define a context for a recovery procedure. There is no obvious counterpart for these structuring units in asynchronously communicating agents. An agent produces messages in a certain order, each being a result of some calculations. When the data sent along with a message cause an exception in an agent, the agent may want to notify the original message producer, for example, by sending an exception. When an exception arrives at the message producer (which is believed to be the source of the problem), it is possible that the agent has proceeded with other calculations and the context in which the message was produced is already destroyed. In addition, an agent can disappear due to migration or termination.
- *Message Semantics*
  In a distributed system developed in a centralised manner, semantics of values passed between system components is fixed at the time of the system design and implementation. In an open agent system, implementation is decentralised and thus the message semantics must be defined at the stage

of a multi-agent application design. If an agent is allowed to send exceptions, the list of exceptions and their semantics must also be defined at the level of an abstract application model. For a recovery mechanism, this means that each agent has to deal only with the exception types it can understand, which usually means having a list of predefined exceptions.

We have to take these issues into account when developing and implementing fault-tolerant AmI systems. The following section outlines the design approach intended for constructing the ambient campus case study scenarios.

3 Design Approach

Our overall aim is to develop a fairly detailed model which covers a number of issues critical for ambient systems, including communication, networking failures, proactive recovery, liveness, termination, and migration.

No existing formal modelling technique can adequately address all the development stages of a complex, large-scale agent system. The proposed modelling method combines high-level behavioural descriptions, detailed functional specifications and agent-level scenarios for an integral approach addressing the issues of parallelism, distribution, mobility and context.

Different modelling techniques are used to operate on a common general model. A general system model is projected through a number of views, each emphasizing some specific aspect of a model. The choice of views was dictated by the availability of the verification toolkits that can be used to automate analysis of model properties. Another important role of views is to help a designer to better understand a model.

Unlike hybrid methods – where two or more notations are used – in our approach, a whole model is described in single basic notation based on the standard set theory language. This reduces the possibility of consistency problems and makes the method more elegant and flexible. The model notation is deliberately very schematic, no tool is going to support it and no real system can be described in it. Instead we propose to use a graphical modelling tool that can visually layout and manipulate different model parts.

Since it is hard to produce an efficient and scalable verification tool, we try to reuse the well-established formalisms supported by verifications tools. Our approach is based on a combination of a process algebra and a state-based modelling method. A CSP-inspired process algebra is used for verification of high-level system behaviour while the Event-B specification method [5] is employed to construct detailed functional specifications. The Mobility Plugin model checker [6] is used to verify hybrid specifications composed of a process algebraic model and an Event-B machine.

The motivations for this work is the construction of a tool for computer-aided development of agent systems. Such tool would combine simplicity of visual modelling tools such as UML, the expressive power of specifications languages such as B ([7]) and Z ([8]) and systematic step-wise development of the refinement.

The result of such development is a set of specifications of agent *roles*. Due to the top-down development, such specifications are interoperable. Role specifications can be taken apart and implemented or further developed independently without risking losing interoperability with the other agent roles of the system. In many cases, executable code can be generated directly from a role specification.

3.1 The AgentB Modelling Database

The AgentB modelling database concept unites the various model types used in a development. Most of the models are rather simple on their own and thus easy to read and update. Combining these models together produces a powerful modelling tool, and their combination is never written out as a single specification. Instead, a software tool is responsible for maintaining links among model parts to ensure consistency of the whole development.

A refinement of a formal development is constructed by refining different database parts, one at a time. This is possibly the most attractive feature of the approach. Instead of tackling a whole model, a modeller can choose a specific aspect of a model to work on. At any given moment, the focus of a modeller is on a single database part or a synthetic view constructed from a combination of several parts. The modelling database has the following structure:

$$Sys = (S, V, E, R, P, F, C, D, A, I)$$

where

S - collection of carrier sets and variables. They are used in the functional, communication and agent database parts.
V - variable model. This includes variables used by functional, communication and agent models.
E - event model, as a set of possible system events.
R - role model, collection of system roles.
P - a protocol model, in the form of a CSP-like process algebraic expression. A protocol model is a high-level description of observable system behaviour. This model does not refer to or update system state. For this reason it is very convenient to use the protocol part alone to design an initial system abstraction.
F - functional model. It describes the state updates done by events present in the system. Functional model of a single event includes a set of local variables, a guard, and a before-after predicate relating a new system state to an old one.
C - communication model. It is used to describe how information is passed between different agents roles, as roles do not normally share variables. The model helps to distinguish between internal control flow of an agent and external message passing.
D - distribution model. This model relates elements of the event and variable models to roles from the role model. An empty distribution model stands for an implicit single role which contains all the variables and events.

A - agent model. This model describes *locations* and *agents* of a system. The model helps to address the problems of mobility and context-awareness.

I - a system invariant. Properties expressed by a model invariant must be preserved at all stages of a system execution. Typically, an invariant consists of typing predicates for system variables, functional model properties, agent model properties and, possibly, a gluing invariant for linking model refinements.

Model context contains static information used by a development. It declares user carrier sets, constants and a context properties: (S, C, P).

Variable Model The variable model part describes the state space of the modelled system. This model must be accompanied by the invariant part providing typing predicates for all the variables.

In a modelling database, the variable model part is used by four other parts – functional, distribution, communication and agent models. All variables are visible to these parts. The initialisation event of the functional model is responsible for computing the initial state of a system. Agent and functional models are the only parts which can update variable states.

Event Model Event is an observable action of a system, it has no duration and thus only one event can be observed at any given moment. An event model indicates which events may be observed in a correct model implementation. For example, for event model $\{a, b\}$, all the systems with the following observable behaviours are correct implementations:

$\langle a, a, a, ... \rangle$
$\langle b, b, b, ... \rangle$
$\langle a, a, b, a, b, b, ... \rangle$
$\langle \rangle$

where by $\langle \rangle$ we denote a system which stops immediately when started (a system doing nothing is a valid implementation of any event model). Any system with events other than a and b is not a valid model implementation. For instance, $a, a, a, c, ...$ does not implement the model since c is not included in the event model.

Role Model The role model part declares a set of system roles (component types) and for each role, specifies the minimum number of role instances required to construct a system and the maximum number of role instances that is supported by a system. A role model is defined with a tuple made of a set of roles and two functions defining the restriction on role instance number:

$$(R, \mathrm{rmin}, \mathrm{rmax})$$

where R is the set of roles, rmin and rmax are functions specifying the minimum and maximum number of instances for each role:

$$\text{rmin} : R \to \mathbb{N}_1$$
$$\text{rmax} : R \to \mathbb{N}_1 \cup \{\infty\}$$

Protocol Model Behavioural modelling is a natural choice for high-level modelling of parallel and distributed systems. Behavioural specifications focus on temporal ordering of events, omitting details of state evolution. The agent system paradigm is one example where behavioural model is preferable for high-level system abstraction although state-based description may be required at later stages. The protocol model language is based on a subset of CSP notation (Figure 1).

$e \to P$	synchronisation prefix
$P; Q$	sequential composition
$P \| Q$	parallel composition
$P \sqcap Q$	choice
$\mu x \cdot P(x)$	recursion
skip	no-effect process

Fig. 1. The language of protocol model expression.

We will often use the following shortcut notation for describing loops:

$$^*(P) = P; P; P; ... = \mu X \cdot (P; (X \sqcap \mathbf{skip}))$$

Reactions In AgentB, we are interested in the modelling of distributed systems. To make transition into implementation stage easier, we try to achieve distribution at the modelling level. Protocol model is one of the parts that must be split somehow into pieces to faithfully model a distributed system. For this, we represent a protocol model as a collection of several independent protocol models:

$$P_1, P_2, ..., P_n$$

To be able to refer to parts of a protocol model, protocol sub-models are identified with unique labels:

$$l_1 : P_1, l_2 : P_2, ..., l_n : P_n$$

Here P_i are the protocol model parts and l_i are the attached labels. For example, a model of a server providing two different services – reading a file and saving a file – can be described as:

$$readfile : P\|$$
$$savefile : Q$$

Each reaction name has a special meaning. Reaction with label $\star$ is a protocol model of a whole system before it is completely decomposed into models of individual roles. Other reactions labels are only notational decorations and are not given any interpretation.

Functional Model With the functional model part, a modeller specifies how an event updates the state of a system. This is done by formulating predicates relating old and new system states. Such a predicate does not have to describe a unique new state, instead it describes a whole family of possible next states. Functional model of an event is equipped with a guard. A guard defines the states when the event can be enabled. If an event execution is attempted in a state prohibited by its guard, execution is suspended until the system arrives at a state satisfying the guard.

Functional model of an event computes a new system state in a single atomic step. It does not use any intermediate steps or intermediate local results and it is not interleaved with the execution of other events. Functional model always contains initialisation event. This event is special: it cannot be referred-to anywhere in a model (for example, in a protocol model expression) and this event is always prior to any other event. The functional model of an event is described by event guard and event action:

$$F : Ev \nrightarrow (Grd \times Act)$$

where

Ev - event identifier, must be an element of the Event model;
Grd - event guard. In addition to typing predicates for parameters and event enabling conditions, an event guard can also have free variable that must be typed by the guard. These variables are the local variables of the event. They are not seen outside the event and cannot be updated by the event action;
Act - generalised substitution defined on variables from the variable model.

We use generalised substitutions to describe how an action transforms a model state. The table below lists the substitution styles that are used to describe an action:

notation	relation predicate	
$v := F(c, s, v, l)$	$v' = F(c, s, v, l)$	assignment
skip	$v' = v$	no-effect assignment
$v :\in F(c, s, v, l)$	$v' \in F(c, s, v, l)$	set choice
$V :\mid F(c, s, V_0, V_1, l)$	$F(c, s, V_0, V_1, l)$	generalised substitution

where v is a variable, F is an expression, V is a vector of variables and V_0 and V_1 are the old and new values of V. Expression F may refer to constants c, sets s, system variables v and local variables l. The first of the substitution type,

:=, is a simple assignment. The assigned variable becomes equal to the value of expression F. Substitution $v :\in F$ selects a new value for v such that it belongs to set F. The most general substitution operator, $:|$, uses a predicate to link the new and old model states.

Several substitution types can be combined into a single action with the parallel composition operator:

$$s_1 \| s_2 \| ... \| s_k$$

We perceive parallel composition of actions as a simultaneous execution of all the actions. We can always replace the set of parallel substitutions with a single generalised substitution.

Communication Model The communication model allows a modeller to analyse and update communications that may occur in a system. By communication we understand a pair of "a sending event" and "a receiving reaction" (described by the protocol model part) and a predicate-binding parameters that define the communication-enabling conditions. As with the functional model part, the communication model does not have explicit parameters. Parameter passing is modelled by the conjunction of event and communication guards.

The purpose of this model is to keep the information about communication separate from other parts. The reason to do this is because we cannot assign communication to protocol model as it would make it very hard to achieve decomposition into agent models which is important to our method. Communication cannot be described in the functional model part as a functional model is formulated on per-event basis. Introducing communication would destroy this simple architecture.

Communication is introduced when a system has more than one role. To make sure that parts of the system that are to be implemented as independent components are linked in a manner that does not prevent their distribution, we use communication model to describe possible messages exchanged by such components.

A communication model associates a set or sets of communications with a source event (message sender). Each communication is a tuple of a guard and a destination event:

$$C : Ev \nrightarrow \mathcal{P}(Grd \times Rct)$$

where Ev is the message source – the event sending the message. Predicate Grd determines whether a message should be sent and the values for the parameters should be passed to the designation event. The message target is a reaction name.

Distribution Model Distribution model defines how the functionality and the state of a model are partitioned among the roles of a system. This permits a system to be realised as a set of independent components. Each variable and

event of a model is associated with a particular role and additional restrictions are imposed on protocol and functional models. Formally, a distribution model is described as a tuple of functions partitioning events and variables:

$$D = (D_e, D_v)$$

where function $D_e : Ev \twoheadrightarrow \mathcal{P}(R)$ maps an event into a role to which the event belong. Function $D_v : V \twoheadrightarrow \mathcal{P}(R)$ does the same for a model variable.

Agent Model An agent model is a tuple of locations set L and agent specifications A:

$$M = (L, A)$$

An agent specification describes the behaviour of a single agent. At any given moment, an agent is located at some location from the set L. The set L always contains the predefined location *limbo* which is understood as the whole of the 'outside' world. From this location, new agents appear in a system and via this location agents may leave a system.

The role of the location concept is to structure an agent system into disjoint set of communicating agent groups. This addresses the scalability problem of agents-system. A large and complex system can be described as a composition of smaller and simpler sub-systems, well isolated from each other.

An agent may communicate with other agents in the same location. Agents from different locations may also communicate. An agent may decide to change its position by migrating to a new location. This changes the set of agents it sees and can communicate to. The structuring of an agent system and agent grouping is dynamic. An agent can use its current state, produced by interacting with other agents, to compute the next migration destination. This permits the description of dynamic agent systems with complex reconfiguration polices.

Specification of an agent behaviour is a process algebraic expression. The decomposition model makes sure agents are defined in a non-conflicting manner.

The starting point for the construction of an agent system is the assignment of a set of roles to each agent. An agent with roles R is understood to be a component implementing the complete functionality of all the roles from R and is required to provide all the services attributed to these roles.

An agent model is described by two functions: Arl provides the list of roles implemented by an agent and Asp returns an agent specification.

$$Arl : Agt \nrightarrow \mathcal{P}_1(R)$$
$$Asp : Agt \nrightarrow Sp$$

We require that $dom(Asp) = dom(Arl) \neq \oslash$.

An agent specification is described using a small subset of CSP, which features the following constructs:

$Pred?a$	guarded action
$P;Q$	sequential composition
$P \| Q$	parallel composition
$P \sqcap Q$	choice
$Pred?^*(P)$	loop

where action a is:

$\overline{[m,p]}$	invocation of an internal reaction
$\mathbf{go}(l)$	migration
$evt(p_1,p_2,...,p_n)$	execution of an event from a functional model
$[m,p]$	communication process sending event m with parameters p

The migration action changes the current position of an agent. Invocation of an internal reaction results in a creation of a new process within an agent. Such a process is described by a combination of protocol and functional models. An agent can send a message to another agent provided the destination agent can be found at the current location. As a reaction to a message, the receiver creates a new process with the internal reaction invocation action. Finally, an agent model may call an event defined in the functional model of an agent.

To summarize, the proposed modelling framework has been developed to fit well with the major characteristics of the agent systems identified in Section 2. Role, distribution and communication models guarantee agent decentralization and weak communication. The event-based communication between agents ensures their anonymity. Agents are autonomous and do not have to communicate if this does not fit their goals. The framework supports independent development of individual agents in such a way that they are interoperable, function in the distributed settings and can move by changing locations when and where they want to achieve their individual goals.

4 Fault-Tolerance

To ensure fault tolerance of complex ambient applications, we address the fault-tolerance issues through the entire development process starting from eliciting relevant operational and functional requirements. In our approach, system operational and functional requirements – among other information – capture all possible situations which are abnormal from the point of view of the system stakeholders (including, system users, support, developers, distributors and owners). First of all, this allows us to state the high level fault assumptions, which, generally speaking, define what can go wrong with the system – and as such, needs tolerating – and what we assume will never go wrong (the latter is as important as the former as it defines the foundation on which fault tolerance can be built). These requirements guide the modelling of the error detection and system recovery.

Due to the complex nature of large-scale AmI applications – caused by their dynamic nature and openness – the traditional fault tolerance structuring techniques, such as procedure-level exception handling, atomic transactions, conversations and rollback cannot be applied directly as the systems typically need combined approaches used for dealing with different threats in different contexts. Within our modelling approach, fault tolerance becomes a crosscutting concern integrated into a number of model views and at different phases of incremental system development. System structuring, ensuring that potential errors are contained in small scopes (contexts) represented as the first class entities during system modelling, is in the core of this approach (in the same way as it is in the core of providing any application fault tolerance [9]).

Fault tolerance is systematically introduced during the development of various model views. Thus, the event model includes both normal and abnormal events, where the latter represents various situations ranging from detecting errors to successful completion of system recovery. Each role model typically constitutes a simple scope, which becomes the first level of system recovery, to be conducted by the individual role, without involving other agents or other roles of the same agent. In some situations, this type of recovery can be successful considering the agent's autonomy and the decentralized nature of the AmI applications.

Unfortunately, our experience shows that in real systems, we often need to conduct a higher level recovery which involves other agents. There are many reasons for this, including cooperative and interactive nature of these applications, in which agents come together to achieve their goals, so that they often need to cooperate to recover and to ensure that during and after recovery, the whole system is in a consistent state.

Within AgentB, different model views deal with faults and errors in view-specific ways and use specific fault tolerance measures. The protocol model allows us to raise and propagate exceptional messages and to conduct application-specific recovery modelled as a separate part of each protocol (providing a special form of exception handler – see [10]). The abnormal part of the protocol view shows message sequences typically exchanged during system recovery.

In the agent models, we introduce fault-tolerance properties at the level of agents, which are the units of deployment and mobility in the AmI systems, as well as at the level of groups of cooperating agents. This allows us to represent fault tolerance at both: the level of individual agents and the level of groups of agents deployed in the same location. In particular, we can represent the use of redundancy (e.g. to achieve fault handing by spawning an agent copy to survive an agent crash) and diversity (to achieve error recovery by employing the same service provided by independently implemented agents). We can also model fault tolerance of a group of agents (for example, when they need to leave a location in emergency or when we need to conduct load balancing operation to avoid system degradation).

To automate system modelling, we are currently designing a number of fault tolerance patterns to help system developers introduce some common fault-

tolerance techniques when modelling an agent system [11, 12]. These techniques range from abstract system-level patterns to very specific agent-level patterns dealing with specific faults and focus on integrating fault tolerance in the specific modelling views.

Early on, we have extended the blackboard communication pattern [13] with nested scopes and exception propagation [14]. These two extensions are essentially the modelling and the implementation techniques aiming at representing recovery actions. In the implementation of fault tolerance, we extensively rely on the reactive agent architecture. This has two immediate benefits: its implementation style matches the event modelling style, captured by the event and functional model views; and recovery of multi-threaded agents becomes similar to that of the asynchronous reactive architecture.

In spite of some success in modelling different fault tolerance solutions for the AmI systems, we realise that our approach needs further work, in particular, in coherent modelling of fault tolerance represented in different model views. In the work we report here, in most cases we treat fault tolerance in different views as being orthogonal and non interfering, assuming that the erroneous state is always confined to one model view at a time: in this case, error recovery can be localised in this view. To address the more general cases where the same error affects several views or recovery from concurrent errors that need coordinated activities in several views, we will need to define common parts of the views and some rules of their sharing/transformation.

5 Case Study Scenarios

In our previous work, we implemented two scenarios within the ambient campus case study using the CAMA framework as the core component of the applications [15, 16, 4]. The first scenario (*ambient lecture*) deals with the activities carried out by the teacher and the students during a lecture – such as questions and answers, and group work among the students – using various mobile devices (PDAs and smartphones). The second scenario (*presentation assistant*) covers the activities involved in giving and attending a presentation. The presenters uses a PDA to control the slides during their presentation and they may receive 'quiet' questions on the topic displayed on the slide from the audience. Each member of the audience will have the current slide displayed on his/her PDA, which also provides a feature to type in questions relevant to that slide.

In this section we discuss our work on a more challenging scenario which involves greater agent mobility as well as the use of the location specific services. Agents may move physically among multiple locations (rooms), and depending on the location, different services will be provided for them. In this work, we shift our focus from implementation to design and we use this scenario to validate our formal development approach.

In this scenario – we call it the *student induction assistant* scenario – we have new students visiting the university campus for the first time. They need to register to various university departments and services, which are spread

on many locations on campus, but they do not want to spend too much time looking for offices and standing in queues. They much prefer spending their time getting to know other students and socialising. So they can delegate the registration process to their personalised software agent, which then visits virtual offices of various university departments and institutions, obtains the necessary information for the registration, and makes decisions based on the student's preferences. The agent also records pieces of information collected during this process so that the students can retrieve all the details about their registration.

Unfortunately, not all the registration stages can be handled automatically. Certain steps require personal involvement of the student, for example, signing paperwork in the financial department and manually handling the registration in some of the departments which do not provide fully-featured agents able to handle the registration automatically. To help the student to go through the rest of registration process, his/her software agent creates an optimal plan for visiting different university departments and even arranges appointments when needed.

Walking around on the university campus, these new students pass through *ambients* – special locations providing context-sensitive services (see Figure 2). An ambient has sensors detecting the presence of a student and a means of communicating to the student. An ambient gets additional information about students nearby by talking to their software agent. Ambients help students to navigate within the campus, provide information on campus events and activities, and assist them with the registration process. The ambient infrastructure can also be used to guide students to safety in case of emergency, such as fire.

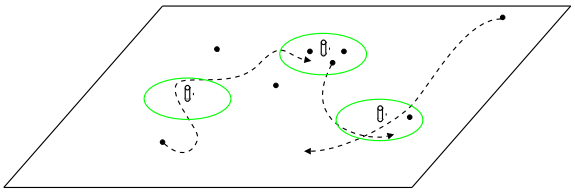

Fig. 2. *Student induction assistant* scenario: the dots represent free roaming student agents; the cylinders are static infrastructure agents (equipped with detection sensors); and the ovals represent *ambients* – areas where roaming agents can get connection and location-specific services.

5.1 Application of Our Approach to the Scenario

To proceed further, we need to agree on some major design principles, identify major challenges and outline the strategy for finding the solution. In order to understand the scenario better, we apply the *agent metaphor*. The agent metaphor is a way to reason about systems (not necessarily information systems) by decomposing it into agents and agent subsystems. In this paper, we use the term *agent* to refer to a component with an independent thread of control and state, and the term *agent system* to refer to a system of cooperative agents.

From agent systems' viewpoint, the scenario is composed of the following three major parts: physical university campus, virtual university campus and ambients. In the physical university campus, there are students and university employees. Virtual campus is populated with student agents and university agents. Ambients typically have a single controlling agent and a number of visiting agents. These systems are not isolated, they interact in a complex manner and information can flow from one part to another.

However, since we are building a distributed system, it is important to get an implementation as a set of independent but cooperative components (agents). To achieve this, we apply the following design patterns:

agent decomposition During the design, we will gradually introduce more agents by replacing abstract agents with two or more concrete agents.

super agent It is often hard to make a transition from an abstract agent to a set of autonomous agents. What before was a simple centralised algorithm in a set of agents must now be implemented in a distributed manner. To aid this transition, we use *super agent* abstraction, which controls some aspects of the behaviour of the associated agents. Super agent must be gradually removed during refinement as it is unimplementable.

scoping Our system has three clearly distinguishable parts: physical campus, virtual campus and ambients. We want to isolate these subsystems as much as possible. To do this, we use the scoping mechanism, which temporarily isolates cooperating agents. This is a way to achieve the required system decomposition. The isolation properties of the scoping mechanism also make it possible to attempt autonomous recovery of a subsystem.

orthogonal composition As mentioned above, the different parts of our scenario are actually interlinked in a complex manner. To model this connections, we use the *orthogonal composition* pattern. In orthogonal composition, two systems are connected by one or more shared agents. Hence, information from one system into another can flow only through the agent states. We will try to constrain this flow as much as possible in order to obtain a more robust system.

locations definition To help students and student agents navigate within the physical campus and the virtual campus, we define location as places associated with a particular agent type.

decomposition into roles The end result of system design is a set of agent roles. To obtain role specifications, we decompose scopes into a set of roles.

5.2 Formulating the Requirements

From the initial description of the scenario, we formulated a set of requirements that would assist us in implementing the student induction assistant system, in particular concerning the registration process. These requirements can also be found in the RODIN Deliverable D27 [17]. We divided the system requirements into the following categories:

ENV	Facts about the operating environment of the system.
DES	Early design decisions captured as requirements.
FUN	Requirements to the system functionality.
OPR	Requirements to the system behaviour.
SEC	Requirements related to the security properties of the system.

Top-Level Requirements First we attempt a high-level description of the system. The description captures different aspects of the system: environment, some design decisions (dictated by the motivation for this case study), and few general functionality and security requirements.

FUN1	*The system helps new students to go through the registration process.*

DES1	*The system is composed of university campus, virtual campus and ambients.*

OPR1	*A student must have a choice between automated and manual registration.*

OPR2	*Malfunctioning or failure of the automated registration support should not prevent a student from manual registration.*

SEC1	*The system should not disclose sensitive information about students.*

SEC2	*The system must prevent malicious or unauthorised software to disguise itself as acting on behalf of a student or an employee.*

University University campus forms the environment for the software-based registration process (Figure 3). The university campus is obviously not something that can be designed and implemented. However it is important to consider it in the development of the scenario as it provides an operating environment for the other two parts (virtual campus and ambients) which can be implemented in software and hardware.

ENV1	*In university campus, students interact with university employees.*

ENV2	*Students can freely move around while employees do not change their position.*

ENV3	*Each university employee is permanently associated with a unique location.*

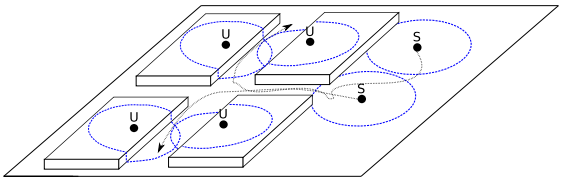

Fig. 3. University campus is modelled as a number of university employees (U) and students (S). Virtual campus has the same structure but is populated with student and university agents.

Virtual Campus Virtual campus uses software-based solution to process student registration automatically. Its organisation is similar to that of a real campus.

DES2	*Virtual campus is composed of university agents and student agents.*

DES3	*In virtual campus, student agents can autonomously change their location.*

DES4	*Each university agent is permanently associated with a unique location.*

Virtual campus is a meeting place for student agents and university agents. During registration, student agent talks to different university agents.

FUN2	*Student agents and university agents can exchange information related to the registration process.*

Some registration steps require intervention from a student.

OPR3	*A registration process may fail due to inability of a particular university agent to handle the registration.*

Before the registration process is initiated, a student agent has to go through several other stages. This results in a tree of dependencies. The root of the tree represents a successful registration and its leaves represent the registration stages without any prerequisites (see Figure 4). Student agent does not know about the tree structure and so it has to explore it dynamically. Reconstructing the tree for each agent makes the system more flexible and robust.

OPR4	*Each registration stage has number of dependencies.*

DES5	*Initially, student agent does not know the dependency tree.*

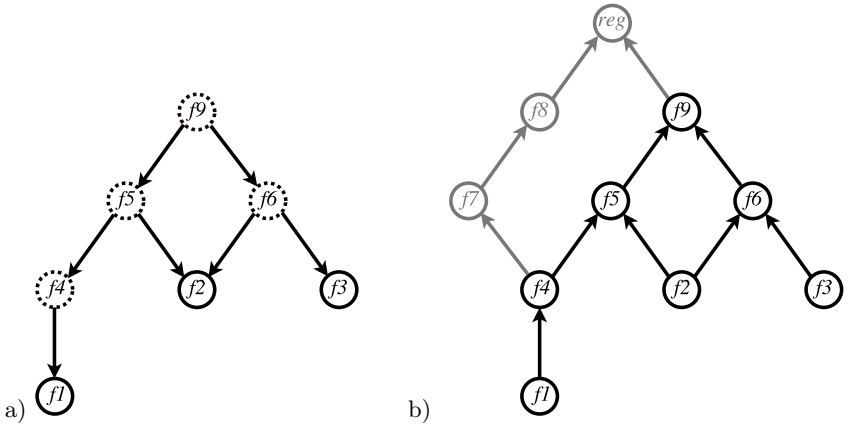

Fig. 4. a) Registration process starts from a random location (*f9* on the figure). The basic requirements *f1, f2, f3* are discovered by tracing back the requirements graph. b) Student agent attempts to do the registration by satisfying each known requirement. It does not yet know the full set of registration requirements (unknown steps are greyed). They are discovered during this process.

DES6	*Student agent autonomously constructs the dependency tree.*

Interacting with university agents, student agent records all the information related to the registration process. This information can be used to restart the registration process or to be passed to the student in order to do manual registration. In the latter case, student agent creates a schedule that helps a student to visit different university offices in the right order and at the right time (see Figure 5).

DES7	*Student agent keeps a history of the registration process that can be used to restart the registration from the point of last completed registration step.*

DES8	*Student agent can create an itinerary for a student to complete the registration manually.*

DES9	*Itinerary must satisfy the registration dependencies.*

Ambient As implied by the scenario, ambients provide services within a predefined physical location. By services we understand an interaction of an ambient with student's software. An interaction is triggered when a student enters a location associated with a given ambient.

OPR5	*Ambients interact with student agents to assist with the registration process.*

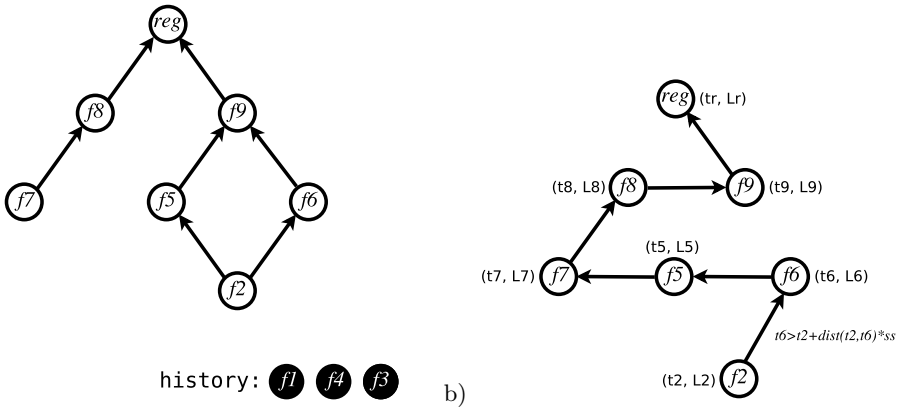

Fig. 5. a) During registration a student agent accumulates registration information. b) Itinerary for manual continuation of a registration is a path covering all the remaining registration graph nodes and satisfying a number of constrains. A node of the path is described by a pair containing when and where should go to resolve a given registration dependency.

FUN3	*Ambient provides services by interacting with student software.*

FUN4	*Interaction with an ambient is triggered when a student enters a location associated with the ambient.*

FUN5	*Interaction with an ambient is terminated when a student leaves the location of the ambient.*

Positioning Service For simplicity, we assume that ambient locations are discreet – a student is either within a location or outside of it – and do not change over time.

FUN6	*Ambient locations are discreet and static.*

Discovery of an ambient by a student (or vice versa) does not come for free. It is achieved using tiny mobile sensor platforms called *smart dust* [18]. Smart dust devices – also known as *motes* – has low-power, short-range radio capability, enabling them to communicate with other motes within range (Figure 6).

ENV4	*Ambients detect students nearby using the mote radio communication.*

Each student carries one such mote which broadcasts student's identity at certain intervals.

ENV5	*Each student carries a mote.*

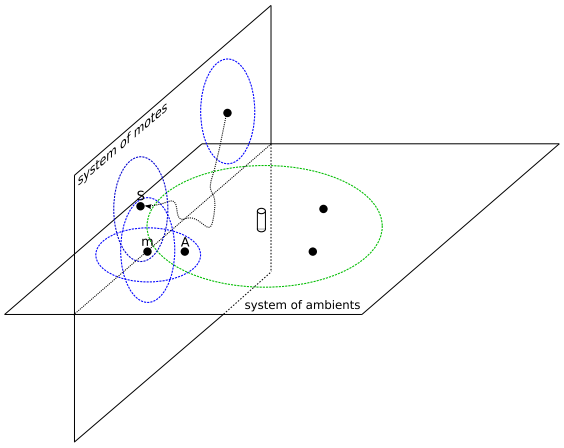

Fig. 6. Composition of motes and ambients system.

FUN7	*Student mote broadcasts student id.*

Student motes' signals are sensed by ambients. Ambient agent is equipped with a mote radio receiver.

ENV6	*Each ambient is equipped with a mote radio receiver.*

When an ambient senses that a student mote is within range, it transmits this information to all other ambients.

FUN8	*Position of a student detected by an ambient is made available to all other ambients.*

We will rely on this functionality to implement recovery in emergency situations.

Student Automated registration must be under the full control of a student. A student should be able to start, stop and inspect the current state of a registration.

FUN9	*Student starts and stops registration process.*

FUN10	*Student may enquire the current state of a registration while registration is in progress.*

FUN11	*When registration is finished or interrupted, a student can access the recorded registration state.*

Student Agent Student agent is a software unit assisting a student in registration.

FUN12	*Student agent assists a student in manual registration by creating a schedule for visiting university employees.*

FUN13	*Student agent records the state of registration process.*

Mobility The scenario includes several types of mobility. There is physical mobility of computing platforms owned by students (e.g. mobile phones and PDAs). Students' agents can migrate to and from a virtual campus world. In this case, agent code and agent states are transferred to a new platform using code mobility. Finally, agents migrate within a virtual campus using virtual mobility.

Different styles of mobility have different requirements. Code mobility is a complex and fail-prone process: it is dangerous to have an agent separated from its state or having an agent with only partially available state or code. There is also a danger of an agent disappearing during the migration: the source of migration, believing that migration was successful, shuts down and removes the local agent copy, while the destination platform fails to initialise the agent due to transfer problems.

OPR6	*Agent either migrates fully to a new platforms or is informed about inability to migrate and continues at a current platform.*

Physical mobility presents the problems of spontaneous context change. A student agent may be involved in a collaboration with an ambient when a student decides to walk away. Clearly, student behaviour cannot be restricted and such abrupt changes of context and disconnections must be accounted for during the design of agents and ambients.

OPR7	*Interaction between an ambient and a student agent can be interrupted at any moment.*

Virtual mobility is the simplest flavour of mobility as it does not involve any networking and nothing is actually moving in space. The only possible failure that can affect virtual migration is a failure or a shut-down of the hosting platform. However, such dramatic failure is unlikely to happen during an agent lifetime and thus we do not consider it at all in this document.

Fault-Tolerance The system we are designing is a complex distributed system with a multitude of possible failure sources. In addition to traditional failures associated with networking, we have to account for failures related to environmental changes which are beyond the control of our system. Below is the list of faults we are going to address and which we believe covers the possible failures in our system:

– disconnections and lost messages:

OPR8	*Agents must tolerate disconnections and message loss.*

– failure of ambients:

OPR9	*Student agents must be able to autonomously recover from a terminal ambient failure.*

also, since ambient services are not critical, it is better to avoid failing or misbehaving ambient:

FUN14	*Student agent drops interaction with an ambient if it suspects that the ambient is malfunctioning.*

– failure of university agents. University agents are critical for the completion of the registration, so it is worth trying to recover cooperatively:

OPR10	*Student and university agents cooperate to recover after failure.*

It does not make sense to remain in virtual campus if one of the university agents is failing to interact:

FUN15	*Student agent leaves virtual campus when it detects a failing university agent.*

– failure of student agents. Failure of a student agent may be detected by a university agent, ambience agent or student.

FUN16	*University agent detecting student agent crash should attempt to notify the agent owner.*

And there is a possibility that a student suddenly terminates without leaving any notice. In this case, we rely on the student to detect this situation and possibly try again by sending another agent.

FUN17	*Student should be able to restart registration process.*

5.3 Refinements

In this section, we demonstrate few initial development steps for the case study. These steps are done in a process-algebraic style, but at a later refinement, the development method changes into state-based modelling using Event-B. More details on our modelling approach can be found in [19].

To ensure interoperability among different agent types in our scenario and also to verify properties (such as eventual termination of the registration process), we use the combination of CSP process algebra [20], AgentB modelling (Event-B with some syntactic sugar – outlined in Section 3), and the Mobility Plugin [6]. The AgentB part of the design is responsible for modelling functional properties of the system; for the verification purposes, it is translatable into proper Event-B models. With the Mobility plugin, we are able to construct scenarios describing typical system configurations and verify properties related

to system dynamics and termination. For example, we can model-check the migration algorithm described in Event-B to verify that the algorithm will never omit a location.

The whole development process is lengthy, so we only show some excerpts here.

Our system is concerned with the registration of a new student. At a very abstract level, the registration process is accomplished in one step:

$$\frac{S_0 \xrightarrow{\text{REF\_PREFIX}} S_1 \text{ sat. } \mathsf{FUN1}}{register.}$$

From the description of the system, we know that the registration process is made of an automatic or manual parts, either of which properly implements the registration process

$$\frac{S_1 \xrightarrow{\text{REF\_ICH}} S_2 \text{ sat. } \mathsf{OPR1}}{\begin{array}{l|l} auto \mapsto register & auto. \sqcap manual. \\ manual \mapsto register & \end{array}}$$

(steps S_3 - S_6 omitted)

At this stage, we are ready to speak about roles of agents implementing the system. We introduce two roles: student (s), representing a human operator using a PDA; and agent (a) which for now stands for all kinds of software in our system.

$$\frac{S_6 \xrightarrow{\text{REF\_ROLE}} S_7 \text{ sat. } \mathsf{DES1}}{\begin{array}{l|l} s, a \in \rho S_7 & \begin{array}{l}(s'send \rightarrow a'move.; a'communicate.; a'automatic.) \sqcap (\\ \quad (auto\_fail \rightarrow manual\_anew.) \sqcap \\ \quad (auto\_part \rightarrow manual\_cont.))\end{array} \end{array}}$$

In the next model, we focus on a sub-model of the system which represents virtual campus (vc) activities: the *communicate* process. The process is refined into a loop where a student agent visits different university agents and speaks to them. The loop alternates between termination (**break**) and the registration process:

$$\frac{communicate. \text{ from } S_7 \xrightarrow{\text{REF\_LOOP}} S_1^{vc} \text{ sat. } \mathsf{FUN3}}{done \mapsto communicate \,\big|\, a'^{+}(auto\_register. \sqcap \mathbf{break}); a'done.}$$

(steps S_2^{vc} - S_6^{vc} omitted)

By adding more details on the interactions between the student and the university agents, we arrive to the following model. The model implements a simple request-reply protocol where the university agent's role is given through a choice from a number of replies. Event *reply_ok* is used when registration is successful, event *reply_docs* indicates that there are missing documents and that student agent must visit some other virtual offices before registration can be completed. In the case when the registration is not possible without the student being present in person, the *reply_pers* reply is used.

$$\frac{S_6^{vc} \xrightarrow{\text{REF\_DCPL}} S_7^{vc}}{sa' \overset{+}{\left(\begin{array}{l} sa'migrate \rightarrow sa'ask.;(\\ \quad (ua'reply\_ok.;sa'save\_repl.)ua' \sqcap \\ \quad (ua'reply\_docs.;sa'doclist.)ua' \sqcap \\ \quad (ua'reply\_pers.;sa'do\_pers \rightarrow sa'\mathbf{break}))ua' \sqcap \\ \quad (ua'fail.;sa'leave\_vc \rightarrow sa'\mathbf{break})) \end{array}\right)} ;sa'done.}$$

(steps S_8^{vc} and S_9^{vc} omitted)

This model prepares the transition to a state-based model with completely decoupled agent roles:

$$\frac{S_9^{vc} \rightarrow S_{10}^{vc}}{\begin{array}{l} ([\psi_1] \sqcap \mathbf{skip}) \| \\ {}^{+}(\psi_1 \rightarrow (sa'(migrate \rightarrow ask.);[\varphi_1])) \| \\ {}^{+}(\varphi_1 \rightarrow ua'((reply\_ok;[\varphi_2]) \sqcap (reply\_docs;[\varphi_3]) \sqcap (reply\_pers;[\varphi_4])) \sqcap (fail;[\varphi_5]))) \| \\ {}^{+}(\varphi_2 \rightarrow sa'save\_repl.;[\psi_1]) \| \\ {}^{+}(\varphi_3 \rightarrow sa'doclist.;[\psi_1]) \| \\ {}^{+}(\varphi_4 \rightarrow sa'do\_pers.) \| \\ {}^{+}(\varphi_5 \rightarrow sa'leave\_vc.) \end{array}}$$

The next two refinement steps add further details to the behavioural model. Refinement step 10 introduces functional model with the modelling decisions taken by the student and the university agents. Further refinements introduce details on how a university agent decides what documents to ask and when the registration process is complete. The student agent keeps track of all visited locations and is able to remember branching points in order not to visit the same university agents twice.

Further details on the refinement process can be found in [17, 21, 19].

5.4 Implemented System and Screenshots

To implement the ambients, we incorporate smart dust devices or motes [18] into the scenario. In particular, we use off-the-shelf MPR2400 MICAz motes (see Figure 7) from Crossbow Technology [22]. These motes communicate with each other using Zigbee radio, and by customising the transmit power of the radio (in this case, reducing the radio range to around 3-5 meters), we can use them as a localisation sensor. This enables us to deliver location-specific information and services to the users.

Fig. 7. MICAz mote used for localisation sensor

Each user carries a mote (programmed with a unique identification number, so that the mote acts as a badge - sort of speak), as well as a PDA as an interaction device. Each room is equipped with a smart dust base station (receiver), which is connected to a controller application. The latter uses the CAMA middleware [3] to communicate with the PDAs through Wi-Fi. When a user enters a particular room, his/her PDA shows the relevant information and/or services available for that room.

A set of rooms can be prepared to be smart dust aware. This can include the reception office, in which the users (i.e. students) can start the registration process or can find out who their tutor is. Figure 8 shows the screen captures of the PDA used by the student ("Alice"). The picture on the left shows the situation where Alice is not in any location that supports the scenario. When she enters the reception room, her PDA adjusts its location and displays the services available in that room (as can be seen in the picture in the middle). In this example, Alice opts to find out who her tutor is (the picture on the right).

6 Conclusion

This paper provides an outline of the work that we had carried out in developing fault-tolerant ambient applications. We introduce a theoretical approach called AgentB, that is based on the modelling database concept, and is composed of several simple modelling methods focusing on various aspects of the system. These modelling techniques allow us to validate the formal development, and

Fig. 8. Screen captures of the registration assistant scenario

to model and build fault tolerant ambient applications. The approach has been demonstrated through a rigorous development of an ambient campus *student induction assistant* scenario, starting from the definition of a set of requirements, the modelling and refinement processes, and finally, the implementation of the system. We developed an agent-based system implementing this scenario, using the Cama framework and the middleware [2] that we have previously developed.

7 Acknowledgements

This work is supported by the FP6 IST RODIN STREP Project [1], the FP7 ICT DEPLOY Integrated Project [23], and the EPSRC/UK TrAmS Platform Grant [24].

References

1. Rodin: Rigorous Open Development Environment for Complex Systems. IST FP6 STREP project, http://rodin.cs.ncl.ac.uk/ (Last accessed: 6 Aug 2008)
2. Arief, B., Iliasov, A., Romanovsky, A.: On Developing Open Mobile Fault Tolerant Agent Systems. In Choren, R., et al., eds.: SELMAS 2006, LNCS 4408. Springer-Verlag (2007) 21–40
3. Iliasov, A.: Implementation of Cama Middleware. http://sourceforge.net/projects/cama (Last accessed: 6 Aug 2008)
4. Iliasov, A., Romanovsky, A., Arief, B., Laibinis, L., Troubitsyna, E.: On Rigorous Design and Implementation of Fault Tolerant Ambient Systems. Technical report, CS-TR-993, School of Computing Science, Newcastle University (Dec 2006)
5. 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)
6. Iliasov, A., Khomenko, V., Koutny, M., Niaouris, A., Romanovsky, A.: Mobile B Systems. In: Proceedings of Workshop on Methods, Models and Tools for Fault Tolerance at IFM 2007, CS-TR 1032, Newcastle University (2007)

7. Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (2005)
8. Abrial, J.R., Schuman, S.A., Meyer, B.: A specification language. In McNaughten, R., McKeag, R., eds.: On the Construction of Programs, Cambridge University Press. (1980)
9. Randell, B.: System Structure for Software Fault Tolerance. IEEE Trans. Software Eng. **1**(2) (1975) 221–232
10. Plasil, F., Holub, V.: Exceptions in Component Interaction Protocols - Necessity. In: Architecting Systems with Trustworthy Components. (2004) 227–244
11. Iliasov, A.: Refinement patterns for rapid development of dependable systems. In: Proceedings of Engineering Fault Tolerant Systems Workshop (at ESEC/FSE, Croatia, ACM Digital Library (2007)
12. Iliasov, A., Romanovsky, A.: Refinement Patterns for Fault Tolerant Systems. In: Technical paper presented at the Seventh European Dependable Computing Conference (EDCC-7), IEEE CS (2008)
13. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System Of Patterns. West Sussex, England: John Wiley & Sons Ltd. (1996)
14. Iliasov, A., Romanovsky, A.: Structured Coordination Spaces for Fault Tolerant Mobile Agents. In Dony, C., Knudsen, J.L., Romanovsky, A., Tripathi, A., eds.: LNCS 4119. (2006) 181–199
15. Arief, B., Coleman, J., Hall, A., Hilton, A., Iliasov, A., Johnson, I., Jones, C., Laibinis, L., Leppanen, S., Oliver, I., Romanovsky, A., Snook, C., Troubitsyna, E., Ziegler, J.: Rodin Deliverable D4: Traceable Requirements Document for Case Studies. Technical report, Project IST-511599, School of Computing Science, University of Newcastle (2005)
16. Troubitsyna, E., ed.: Rodin Deliverable D8: Initial Report on Case Study Development. Project IST-511599, School of Computing Science, University of Newcastle (2005)
17. Troubitsyna, E., ed.: Rodin Deliverable D27: Case Study Demonstrators. Project IST-511599, School of Computing Science, University of Newcastle (2007)
18. Smartdust: Wikipedia definition. http://en.wikipedia.org/wiki/Smartdust (Last accessed: 6 Aug 2008)
19. Iliasov, A., Koutny, M.: A Method and Tool for Design of Multi-Agent Systems. In Pahl, C., ed.: Proceedings of Software Engineering (SE 2008), ACTA Press (2008)
20. Hoare, C.A.R.: Communicating Sequential Processes. Communications of the ACM **21**(8) (1978) 666–677
21. Troubitsyna, E., ed.: Rodin Deliverable D26: Final Report on Case Study Development. Project IST-511599, School of Computing Science, University of Newcastle (2007)
22. CrossbowTechnology: MPR/MIB User's Manual. http://www.xbow.com/Support/Support_pdf_files/MPR-MIB_Series_Users_Manual.pdf (Last accessed: 6 Aug 2008)
23. Deploy: Industrial Deployment of System Engineering Methods Providing High Dependability and Productivity. IST FP7 IP project, http://www.deploy-project.eu/ (Last accessed: 6 Aug 2008)
24. TrAmS: Trustworthy Ambient Systems Platform Grant. http://www.cs.ncl.ac.uk/research/current%20projects?pid=223/ (Last accessed: 6 Aug 2008)