A Refinement-Based Approach to Developing Critical Multi-Agent Systems

Abstract: Multi-agent systems (MAS) are increasingly used in critical applications. To ensure dependability of MAS, we need powerful development techniques that would allow us to master complexity inherent to MAS and formally verify correctness and safety of collaborative agent activities. In this paper we present a rigorous approach to development and verification of critical MAS in Event-B. We demonstrate how to formally specify complex agent interactions and verify their correctness and safety. We argue that refinement approach facilitates structuring complex requirements and formal reasoning about system-level properties. We illustrate our approach by a case study: formal development of a hospital MAS.

Keywords: Event-B; refinement; formal modelling; formal verification; multi-agent system; safety

1 Introduction

Mobile multi-agent systems (MAS) are complex decentralised distributed systems composed of agents asynchronously communicating with each other. Agents are computer programs acting autonomously on behalf of a person or an organisation, while coordinating their activities by communication (OMG Mobile Agents Facility, 1997). MAS are increasingly used in various critical applications such as factories, hospitals, rescue operations in disaster areas, etc. However, widespread use of MAS is currently hindered by the lack of methods for ensuring their dependability.

In this paper we focus on studying complex agent interactions. In a critical MAS, incorrect execution of these activities might have devastating consequences. However, ensuring correctness of complex interactions is a challenging issue due to faults caused by agent disconnections, dynamic role allocation and autonomy of the agent behaviour. To address these challenges, we need the system-level modelling approaches that would support formal verification of correctness and facilitate discovery of restrictions that should be imposed on the system to guarantee its safety.

In this paper we propose a refinement-based approach to developing critical MAS. Our formal modelling and verification framework is Event-B (Abrial, 2010). The main development technique of Event-B is refinement – a top-down approach to formal development of systems that are correct by construction. The system development starts from an abstract specification, which defines the main behaviour and properties of the system. The abstract specification is gradually transformed (refined) into a more concrete specification that is directly translatable into an implementation. Correctness of each refinement step is verified by proofs. These proofs establish system safety (via preservation of safety invariant properties expressed at different levels of abstraction) and liveness (via the provable absence of undesirable system deadlocks). Transitivity of the refinement relation allows us to guarantee that the system implementation adheres to the abstract specifications. The Rodin platform (Rodin Platform, 2006) provides the developers with automated tool support for constructing and verifying system models in Event-B.

The main novelty of our approach is in demonstrating how to gradually *derive* a system implementation that satisfies the desired safety properties. It is different from traditional approaches to verification of MAS that extract a model from a system implementation and verify the desired properties by state-exploration. Our approach is not only free of the state explosion problem but also allows the designers to discover restrictions that should be imposed on the system environment to guarantee system safety. The top-down development approach facilitates structuring of complex requirements and improves comprehensibility of formal models. We argue that the formal development in Event-B offers a useful technique for development and verification of complex critical MAS.

The paper is structured as follows. In Section 2 we describe our formal modelling framework – Event-B. In Section 3 we define the main principles of formal reasoning about MAS and their properties. In Section 4 we present our case study – a hospital MAS. We show here how to abstractly model a MAS, introduce complex collaborative agent interactions by refinement, as well as verify safety properties. Moreover, we describe the last refinement step that models system decomposition, thus achieving derivation of a distributed implementation from a centralised specification. Finally, in Section 5 we overview the related work, discuss the achieved results and outline our future work.

2 Formal Modelling and Refinement in Event B

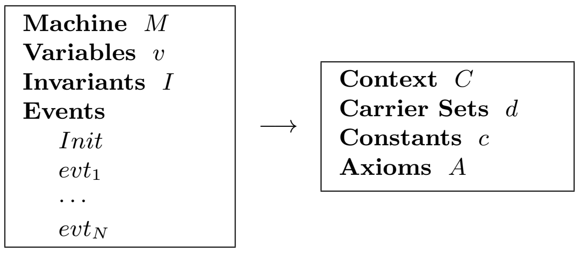
We start by briefly describing our formal development framework. The Event-B formalism is a variation of the B Method (Abrial, 2005), a state-based formal approach that promotes the correct-by-construction development paradigm and formal verification by theorem proving. Event-B has been specifically designed to model and reason about parallel, distributed and reactive systems. Currently Event-B is actively used within the FP7 ICT project Deploy to develop dependable systems from various domains (Deploy Project, 2008).

2.1 Modelling in Event-B

In Event-B, a system specification (model) is defined using the notion of an *abstract state machine* (Abrial, 2010). An abstract state machine encapsulates the model state represented as a collection of model variables, and defines operations on this state, i.e., it describes the dynamic part (behaviour) of the modelled system. A machine may also have the accompanying component, called *context*, which contains the static part of the system. In particular, a context can include user-defined carrier sets, constants and their properties, which are given as a list of model axioms. A general form of Event-B models is given in Figure 1.

The machine is uniquely identified by its name M. The state variables, v, are declared in the **Variables** clause and initialised in the *Init* event. The variables are strongly typed by the constraining predicates I given in the **Invariants** clause.

Figure 1 Event-B machine and context



The invariant clause might also contain other predicates defining properties that should be preserved during system execution.

The dynamic behaviour of the system is defined by the set of atomic events specified in the **Events** clause. Generally, an event can be defined as follows:

$$\textbf{evt} \mathrel{\widehat{=}} \textbf{any } vl \textbf{ where } g \textbf{ then } S \textbf{ end}$$

where vl is a list of new local variables (parameters), the guard g is a state predicate, and the action S is a statement (assignment). In case when vl is empty, the event syntax becomes **when** g **then** S **end**. If g is always true, the syntax can be further simplified to **begin** S **end**.

The occurrence of events represents the observable behaviour of the system. The guard defines the conditions under which the action can be executed, i.e., when the event is *enabled.* If several events are enabled at the same time, any of them can be chosen for execution non-deterministically. If none of the events is enabled then the system deadlocks.

In general, the action of an event is a parallel composition of assignments. The assignments can be either deterministic or non-deterministic. A deterministic assignment, $x := E(x, y)$, has the standard syntax and meaning. A non-deterministic assignment is denoted either as $x :\in Set$, where Set is a set of values, or $x :\mid P(x, y, x')$, where P is a predicate relating initial values of x, y to some final value of x'. As a result of such a non-deterministic assignment, x can get any value belonging to Set or according to P.

2.2 Event-B Semantics

The semantics of an Event-B model is formulated as a collection of *proof obligations* – logical sequents. Below we describe only the most important proof obligations that should be verified (proved) for the initial and refined models. The full list of proof obligations can be found in Abrial (2010).

The semantics of Event-B actions is defined using so called before-after (BA) predicates (Abrial, 2010). A before-after predicate describes a relationship between the system states before and after execution of an event, as shown in Table 1. Here x and y are disjoint lists (partitions) of state variables, and x', y' represent their values in the after-state.

The initial Event-B model should satisfy the event feasibility and invariant preservation properties. For each event of the model, evt_i, its feasibility means that, whenever the event is enabled, its before-after predicate (BA) is well-defined, i.e., exists some reachable after-state:

$$A(d, c),\ I(d, c, v),\ g_i(d, c, v)\ \vdash\ \exists v' \cdot BA_i(d, c, v, v') \qquad \text{(FIS)}$$

Table 1 Before-after predicates

Action (S)	$BA(S)$
$x := E(x,y)$	$x' = E(x,y) \;\wedge\; y' = y$
$x :\in Set$	$\exists z \cdot (z \in Set \wedge x' = z) \;\wedge\; y' = y$
$x :\mid P(x,y,x')$	$\exists z \cdot (P(x,z,y) \wedge x' = z) \;\wedge\; y' = y$

where A is model axioms, I is the model invariant, g_i is the event guard, d are model sets, c are model constants, and v, v' are the variable values before and after the event execution.

Each event evt_i of the initial Event-B model should also preserve the given model invariant:

$$A(d,c),\; I(d,c,v),\; g_i(d,c,v),\; BA_i(d,c,v,v') \;\vdash\; I(d,c,v') \qquad \text{(INV)}$$

Since the initialisation event has no initial state and guard, its proof obligation is simpler:

$$A(d,c),\; BA_{Init}(d,c,v') \;\vdash\; I(d,c,v') \qquad \text{(INIT)}$$

Event-B employs a top-down refinement-based approach to system development. Development starts from an abstract system specification that models the most essential functional requirements. While capturing more detailed requirements, each refinement step typically introduces new events and variables into the abstract specification. These new events correspond to stuttering steps that are not visible at the abstract level. Moreover, Event-B formal development supports data refinement, allowing us to replace some abstract variables with their concrete counterparts. In that case, the invariant of the refined machine formally defines the relationship between the abstract and concrete variables.

To verify correctness of a refinement step, we need to prove a number of proof obligations for a refined model. For brevity, here we show only a few essential ones.

Let us first introduce a shorthand $H(d,c,v,w)$ to stand for the hypotheses $A(d,c)$, $I(d,c,v)$, $I'(d,c,v,w)$, where I, I' are respectively the abstract and refined invariants, and v, w are respectively the abstract and concrete variables. Then the feasibility refinement property for an event evt_i of a refined model can be presented as follows:

$$H(d,c,v,w),\; g_i'(d,c,w) \;\vdash\; \exists w' \cdot\; BA_i'(d,c,w,w') \qquad \text{(REF\_FIS)}$$

where g_i' is the refined guard and BA_i' is a before-after predicate of the refined event.

The event guards in a refined model can be only strengthened in a refinement step:

$$H(d,c,v,w),\; g_i'(d,c,w) \;\vdash\; g_i(d,c,v) \qquad \text{(REF\_GRD)}$$

where g_i, g_i' are respectively the abstract and concrete guards of the event evt_i.

Finally, the *simulation* proof obligation requires to show that the "execution" of a refined event is not contradictory with its abstract version:

$$H(d,c,v,w),\; g_i'(d,c,w),\; BA_i'(d,c,w,w')$$

$$\vdash \exists v' \cdot BA_i(d,c,v,v') \;\wedge\; I'(d,c,v',w') \qquad \text{(REF\_SIM)}$$

where BA_i, BA_i' are respectively the abstract and concrete before-after predicates of the same event evt_i.

The Event-B refinement process allows us to gradually introduce implementation details, while preserving functional correctness. The verification efforts, in particular, automatic generation and proving of the required proof obligations, are significantly facilitated by the Rodin platform (Rodin Platform, 2006). Proof-based verification as well as reliance on abstraction and decomposition adopted in Event-B offers the designers a scalable support for the development of such complex distributed systems as multi-agent systems.

2.3 Modularisation Extension

Recently the Event-B language and tool support have been extended with a possibility to define modules (Iliasov et al., 2010, RODIN Modularisation Plug-in, 2010) – components containing groups of callable atomic operations. Modules can have their own (external and internal) state and invariant properties. An important characteristic of modules is that they can be developed separately and, when needed, composed with the main system.

A module description consists of two parts – *module interface* and *module body.* Let M be a module. A module interface MI is a separate Event-B component. It allows the user of the module M to invoke its operations and observe the external variables without having to inspect the module implementation details. MI consists of external module variables w, constants c, sets s, the external module invariant $M\_Inv(c, s, w)$, and a collection of module operations, characterised by their pre- and postconditions:

```
Interface $MI$
  Sees $MI\_Context$
  Variables $w$
  Invariants $M\_Inv(c, s, w)$
  Initialisation $\cdots$
  Process
      $PE_1 =$ any $vl$ where $g(c, s, vl, w)$ then $S(c, s, vl, w, w')$ end
      $\cdots$
  Operations
      $O_1 =$ any $p$ pre $Pre(c, s, vl, w)$ post $Post(c, s, vl, w, w')$ end
      $\cdots$
```

In addition, a module interface description may contain a group of standard Event-B events under the **Process** clause. These events model the autonomous module thread of control, expressed in terms of their effect on the external module variables. In other words, the module process describes how the module external variables may change between operation calls.

A formal module development starts with the design of an interface. Once an interface is defined, it cannot be altered in any manner. This ensures that a module body may be constructed independently from a model relying on the module interface. A module body is an Event-B machine. It implements the interface by providing a concrete behaviour for each of the interface operations. A set of additional proof obligations are generated to guarantee that each interface operation has a suitable implementation.

When the module M is imported into another Event-B machine, the importing machine may invoke the operations of M and access (read) the external variables of M. To make a specification of a module generic, in $MI\_Context$ we can define some constants and sets (types) as parameters. Their properties then define the constraints to be verified when a module is instantiated.

A general strategy of a distributed system development in Event-B is to start from an abstract centralised specification and incrementally augment it with design-specific details. When a suitable level of details is achieved, certain events of the specification are replaced by the calls of interface operations and variables are distributed across modules. As a result, a monolithic specification is decomposed into separate modules. Since decomposition is a special kind of refinement, such a model transformation is also correctness-preserving. Therefore, refinement allows us to efficiently cope with complexity of distributed systems verification and gradually derive an implementation with the desired properties and behaviour.

In the next section, we outline main principles of formal reasoning about MAS and their properties.

3 Formal Reasoning about Multi-Agent Systems

Let us start by defining a MAS formally.

Definition 1. *A multi-agent system $\mathcal{MAS}$ is a tuple $(\mathcal{A}, \mu, \Sigma, \mathcal{E}, \mathcal{R})$, where $\mathcal{A}$ is a collection of different classes of agents, μ is the system middleware, Σ is the system state space, $\mathcal{E}$ is a collection of system events and $\mathcal{R}$ is a set of dynamic relationships between agents in a MAS.*

Each agent belongs to a particular class or type of agents $A_i,\ i \in 1..n$ such that $A_i \in \mathcal{A}$. This class is dynamic, i.e., new class agents can spontaneously appear or the existing agents may disappear (both normally or abnormally). A particular agent $a_{ij} \in A_i$ is characterised by its local state consisting from agent variables and static agent attributes.

The system middleware μ can be considered as a special agent that is always present in the system. The responsibility of the middleware is to ensure communication between different agents, to detect appearance of new agents or disappearance (both normal and abnormal) of the existing agents, as well as recover from the loss of agent connection.

The system state space Σ contains all the possible states of agents and the middleware. The system events $\mathcal{E}$ include all internal and external system reactions. An execution of an event may change the state of the middleware or agents. In other words, each event is associated with the corresponding relation on Σ, i.e., is of the type $\Sigma \leftrightarrow \Sigma$. Each collaborative activity between different agents (or an agent and the middleware) may be composed of a set of events. Moreover, system events may model appearance or disappearance of mobile agents, sending request from one agent to another, recovery of lost agent connections, etc. The collaborative activity should preserve the following property:

Property 1. *Let $\mathcal{A}_{act}$ and $\mathcal{A}_{ina}$ be sets of active and inactive agents correspondingly, where $\mathcal{A} = \mathcal{A}_{act} \cup \mathcal{A}_{ina}$ and $\mathcal{A}_{act} \cap \mathcal{A}_{ina} = \varnothing$. Let $\mathcal{EAA}$ and $\mathcal{EA}\mu$ be all the collaborative activities (sets of events) between agents and agents and between agents and middleware respectively. Moreover, for each $A \in \mathcal{A}$, let $\mathcal{E}A$ be a set of events in which the agent A is involved. Then*

$$\forall A \cdot\ A \in \mathcal{A}_{act} \Rightarrow \mathcal{E}A \in \mathcal{EAA}$$

and

$$\forall A \cdot\ A \in \mathcal{A}_{ina} \Rightarrow \mathcal{E}A \in \mathcal{EA}\mu$$

For instance, this property implies that while an agent is recovering from a failure it cannot be involved into cooperative activities with other agents.

A collection of system events $\mathcal{R}$ consists of dynamic relationships or connections between active agents of the same or different classes. An agent relationship is modelled as a mathematical relation

$$R(a_1, a_2, ..., a_m) \subseteq C_1^* \times C_2^* ... \times C_m^*,$$

where $C_j^* = C_j \cup \{?\}$. A relationship can be *pending*, i.e., incomplete. This is indicated by question marks in the corresponding places of R, e.g., $R(a_1, a_2, ?, a_4, ?)$. Pending relationships are usually caused by appearance of new agents or disappearance of the existing ones. In addition, an existing agent may initiate a new relationship.

Property 2. *Let $\mathcal{A}_{act}$ be a set of active agents. Let $\mathcal{EAA}$ be all the collaborative activities in which these active agents are involved. Moreover, for each agent $A \in \mathcal{A}_{act}$, let $\mathcal{R}_A$ be all the relationships it is involved in. Finally, for each collaborative activity $CA \in \mathcal{EAA}$, let $\mathcal{A}_{CA}$ be a set of all involved agents. Then, for each $CA \in \mathcal{EAA}$ and $A_1, A_2 \in \mathcal{A}_{CA}$,*

$$\mathcal{R}_{A_1} \cap \mathcal{R}_{A_2} \neq \varnothing$$

This property restricts the interactions between the agents – only the agents that are linked by relationships (some of which may be pending) can be involved into cooperative activities. The system middleware μ keeps track of pending relationships and tries to resolve them by enquiring suitable agents to confirm their willingness to enter into a particular relationships. Additional data structure $Pref_R$ associated with a relationship $R \in \mathcal{R}$ can be used to express a specific preference of one agents over the others. Then middleware enforces this preference by enquiring the preferred agents first. Formally, $Pref_R$ is an ordering relation over the involved agent classes. Thus, for $R \subseteq C_1^* \times ... \times C_m^*$,

$$Pref_R \ \in \ C_1 \times ... \times C_m \ \leftrightarrow \ C_1 \times ... \times C_m.$$

A responsibility of the middleware is to detect situations when some of the established or to be established relationships become pending and guarantee "fairness". Essentially this means that no pending request is ignored forever and middleware tries to enforce the given preferences.

While developing a critical MAS, we should ensure that once initiated, certain cooperative activities are successfully completed. These are the activities that implement safety requirements. The ensure safety we have to verify the following property:

Property 3. *Let $\mathcal{EAA}_{crit}$, where $\mathcal{EAA}_{crit} \subseteq \mathcal{EAA}$, be a subset containing critical collaborative activities. Moreover, let $\mathcal{R}_{pen}$ and $\mathcal{R}_{res}$, where $\mathcal{R}_{pen} \subseteq \mathcal{R}$ and $\mathcal{R}_{res} \subseteq \mathcal{R}$, be subsets of pending and resolved relationships defined for these activities. Finally, let $\mathcal{R}_{CA}$, where $CA \in \mathcal{EAA}$ and $\mathcal{R}_{CA} \subseteq \mathcal{R}$, be all the relationships the activity CA can affect. Then, for each activity $CA \in \mathcal{EAA}_{crit}$ and relationship $R \in \mathcal{R}_{CA}$,*

$$\Box((R \in \ \mathcal{R}_{pen}) \leadsto \ (R \in \ \mathcal{R}_{res})),$$

where $\Box$ designates "always" and $\leadsto$ denotes "leads to".

This property postulates that eventually all pending relationships should be resolved for each critical cooperative activity.

In the next section we will illustrate modelling of MAS in Event-B by a case study – development of a hospital MAS.

4 Abstract Modelling of a Hospital MAS

4.1 A case study description

The hospital MAS consists of two types of agents – patients and medical personnel (called doctors for simplicity). The condition of each patient is monitored by the corresponding medical equipment – an agent representing a patient. The doctor agents are running on Pocket PC-based devices – Personal Digital Assistants (PDA). The hospital provides the wireless connectivity to the doctor agents. Each doctor is associated with one agent. From now on, we will use the terms "patient" and "doctor" to designate both agents and people that they represent.

The medical equipment continuously updates the patient's medical record consisting of different medical measurements as well as detects emergencies – dangerous changes of critical parameters (e.g., blood pressure, pulse rate and etc.). In case of an emergency, the patient agent generates an emergency call that is communicated to the doctor(s) treating the patient. An important safety requirement imposed on the system is that *all emergencies should be promptly handled by the responsible doctors*. In spite of its seeming simplicity, this requirement is hard to ensure. Indeed, a MAS operates in a volatile communication environment, i.e., agents might experience temporal disconnections. Hence the design of our system should incorporate certain fault tolerance mechanisms that would guarantee that each emergency call is eventually handled by some doctor. Moreover, different doctors can be associated with the same patient during different shifts. Therefore, we have to ensure that at the end of a doctor's shift all his/her patients are handed over to another doctor.

Another important safety requirement associated with the system is to guarantee that *a doctor always accesses the most recent patient record and the patient's data are always kept in a consistent state*. We assume that the patient record is stored at the equipment associated with her. To ensure that these requirements are satisfied, we should regulate the access to the patient's data. A specific delivery of a medicine, prescription of a treatment and so on are introduced into the patient's log by the medical personnel via their PDAs. To ensure that the data are updated consistently, we only allow the doctor to modify the patient's data when he or she is in a close (geographical) proximity to the patient. When the doctor arrives to the patient location, the patient data become available at the doctor's PDA and the doctor can modify them. All the modifications are synchronised with the data stored by the patient's equipment. When the doctor finishes examining the patient or delivering a medicine and leaves, the connection to the patient's data is closed. Such a restriction allows us to ensure that only a doctor who is in a close proximity to a patient is allowed to modify the patient's record. Moreover, it also ensures that the doctor has the access to the freshest info about the patient. This precludes, e.g., a possibility of delivering the medicine twice (modelling the security requirements ensuring patient's data integrity is outside of the scope of this paper).

To ensure that the safety-critical requirements imposed on the system are fulfilled, we should put the properties defined in the previous section in the context of doctor-patients interactions. Indeed, each patient should be assigned to a certain doctor. Hence the specification should observe these relationships. While the doctor is inactive (because the shift is over or a failure of the corresponding PDA), doctor-patient interactions cannot occur. Finally, each emergency call should be eventually served.

Next we demonstrate how refinement process in Event-B can facilitate modelling of intertangled agent interactions and verification of their properties.

4.2 Towards modelling agent interdependencies

The main focus of our development is specification of collaborative behaviour of patients and doctors in a hospital MAS. Our abstract specification is very simple. It models the behaviour of the entire hospital MAS in a highly abstract way. We define the variable *med_agents* to present the active doctor agents:

$$med\_agents \subseteq MEDSTAFF$$

The events *Activate* and *Deactivate* model joining and leaving hospital location by the agents. While an agent is active, it can perform certain activities, which is abstractly modelled by the event *Activity*. Here we consider middleware as a special agent that is always present in the system.

In our abstract specification we have merely introduced one type of agents – medical personnel agents. Since these agents perform only engagement and disengagement with the location, they interact with the middleware only.

In our first refinement step we augment our model with a representation of patients and introduce the events that abstractly model interactions between the introduced patients and the doctors. The variable *patients* defines a set of patients admitted to the hospital:

$$patients \subseteq PATIENTS$$

Each patient arriving at the hospital is associated with a doctor who has a primary responsibility for treating the patient. To model this relationship between patients and medical personnel, we introduce the variable *assigned_doctor*, which is defined as a total function associating patients with the active doctor agents:

$$assigned\_doctor \in patients \to med\_agents$$

We omit showing complete specifications of the subsequent refinement steps and merely describe the events and data structures relevant to critical collaborative actions.

At the first refinement step, resulting in the model $Hospital1$, we add the new event *PatientArrival* to model a patient arrival:

```
$PatientArrival \mathrel{\widehat=}$
  **any** $ma, pa$
  **when**
    $pa \in PATIENTS \wedge pa \notin patients \wedge ma \in med\_agents$
  **then**
    $patients := patients \cup \{pa\} \parallel assigned\_doctor(pa) := ma$
  **end**
```

The guard $ma \in med\_agents$ ensures preservation of a specific instance of the *Property 1*: only active medical agents are assigned to the patient agents. We use $\parallel$ to denote a parallel composition of actions.

The dual event *PatientDischarge* models a patient discharge from the hospital in a similar way. Let us observe that the event *PatientArrival* ensures the following property, which can be considered as an instance of the *Property 3*:

$$\Box(\text{new patient} \leadsto \text{assigned doctor}),$$

while, similarly, the event *PatientDischarge* ensures that

$$\Box(\text{a patient leaves the hospital} \leadsto \text{all his/her relationships are removed}).$$

In the refined specification we also elaborate on the event *Activity*. Essentially, the medical personnel should examine the patients and deliver the prescribed medicine. We generalise these actions under the general term "visiting a patient". In our refined model, we define the variable *visited* representing a subset of patients that are currently being examined. The new variable *last_visited* stores for every patient the id of the last doctor agent that has visited her. The interdependencies between the doctor and patient variables are defined by the following invariants:

$$\begin{array}{l} last\_visit \in patients \pprod MEDSTAFF \\ visited \subseteq patients \\ last\_visit[visited] \subseteq med\_agents \\ visited \subseteq dom(last\_visit) \end{array}$$

The events *VisitBegin* and *VisitEnd* refine the abstract event *Activity* and model a visiting procedure. The event *VisitBegin* is specified as follows:

```
$VisitBegin \mathrel{\widehat{=}}$ **Refines** $Activity$
  **any** $ma, pa$
  **when**
    $ma \in med\_agents \wedge pa \in patients \wedge pa \notin visited \wedge$
    $ma \notin last\_visit[visited]$
  **then**
    $last\_visit(pa) := ma \parallel visited := visited \cup \{pa\}$
  **end**
```

Let us note that the event guard ensures preservation of another instance of the *Property 1*: only active medical agents are eligible to interact with the patient agents.

In our abstract specification we have assumed that the doctor agents can leave the hospital at any time. However, formal modelling has uncovered that safety cannot be guaranteed under this assumption. Hence we must impose certain restrictions on the situations when the doctors can actually leave the hospital. Before a doctor agent can leave the hospital, we should reassign his/her patients to another doctor. This is another illustration of the *Property 3*:

$$\Box(\text{a doctor leaves the hospital} \leadsto \text{all his/her patients are reassigned}).$$

We split the abstract event *Deactivate* into two corresponding events: *AgentLeaving* and *ReassignDoctor*. The event *AgentLeaving* models leaving the location by a doctor, who does not have any assigned patients and is not currently involved in examining a patient.

The event *ReassignDoctor* models leaving the location by a doctor who has assigned patients. We must be sure that all his/her patients will be reassigned to the new doctor. Here we again illustrate a specific case of the *Property 1*: only active medical agents are assigned to the patient agents: $ma\_new \in med\_agents$. This event is modelled as follows:

```
ReassignDoctor ≙ Refines Deactivate
  any
    $ma, ma\_new$
  when
    $ma \in ran(assigned\_doctor) \wedge ma \notin last\_visited[visited]$
    $ma\_new \in med\_agents \wedge ma\_new \neq ma$
  then
    $med\_agents := med\_agents \setminus \{ma\} \parallel$
    $assigned\_doctor := assigned\_doctor \lhd\!\!\!- (dom(assigned\_doctor \rhd \{ma\}) \times \{ma\_new\})$
  end
```

4.3 Introducing fault tolerance by refinement

In the specification *Hospital*1, while defining the events *AgentLeaving* and *ReassignDoctor*, we have abstracted away from the reasons behind doctor leaving and patient reassignment. Essentially, a doctor agent might leave the location because the doctor's shift is over or because the doctor agent has irrecoverably failed and should be permanently disconnected. At the second refinement step, *Hospital*2, we introduce a distinction between the normal agent leaving and its disconnection due to a failure. Both cases have a direct impact on the safety requirements. Next we demonstrate how formal reasoning allows us specify handling of these situations in a correct way.

In a MAS, the agents often lose connection only for a short period of time. After the connection is restored, the agent should be able to continue its operations. Therefore, after detecting a loss of connection, the location should not immediately disengage the disconnected agent but rather set a deadline before which the agent should reconnect. If the disconnected agent restores its connection before the deadline then it can continue its normal activities. However, if the agent fails to do so, the location should permanently disengage the agent. Here we deal with a particular instance of the *Property 3*:

$$\Box(\text{an agent's disconnection happens} \leadsto \text{agent reconnects activity or disengages from the location})$$

In the refined specification we define the variable *disconnected* representing the subset of active agents that are detected by location as disconnected:

$$disconnected \subseteq med\_agents$$

Moreover, to model a timeout mechanism, we define the variable *timer* of the enumerated type $\{inactive, active, timeout\}$. Initially, for every active agent, the *timer* value is set to *inactive*. As soon as active agent loses connection with the location, its id is added to the set *disconnected* and its timer value becomes *active*. This behaviour is specified in the new event *DisconnectAgent* as follows:

```
DisconnectAgent ≙
  any $ma$
  when $ma \in med\_agents \wedge ma \notin disconnected$
  then
    $disconnected := disconnected \cup \{ma\} \parallel timer(ma) := active$
  end
```

A temporarily disconnected agent can succeed or fail to reconnect, as modelled by the events *ReconnectionSuccessful* and *ReconnectionFailed* respectively.

If the agent reconnects before the value of timer becomes *timeout*, the timer value is changed to *inactive* and the agent continues its activities virtually uninterrupted. Otherwise, the agent is removed from the set of active agents. The event *ReconnectionSuccessful* is modelled as follows:

```
$ReconnectionSuccessful \mathrel{\widehat{=}}$
  **any** $ma$
  **when** $ma \in disconnected \wedge timer(ma) = active$
  **then**
    $timer(ma) := inactive \parallel disconnected := disconnected \setminus \{ma\}$
  **end**
```

The following invariant ensures that any disconnected agent is considered to be temporarily inactive:

$$\forall ma \cdot (ma \in med\_agents \wedge timer(ma) \neq inactive \Leftrightarrow ma \in disconnected).$$

The introduction of an agent disconnection allows us to make a distinction between two reasons behind leaving the location by a doctor − because of the end of shift or due to the disconnection timeout. To model these two cases, we split the event *AgentLeaving* into two events *NormalAgentLeaving* and *DetectFailedFreeAgent* respectively.

While modelling failure of a doctor agent, we should again have to deal with the *Property 3* :

$$\Box(\text{a doctor abnormally disconnects} \leadsto \text{all his/her patients are reassigned}).$$

In both cases we need to reassign all the patients of the disconnected doctor to another doctor. In a similar way as above, the event *ReassignDoctor* is decomposed into two events *NormalReassignDoctor* and *DetectFailedAgent.* The event *DetectFailedAgent* is specified as follows:

```
$DetectFailedAgent \mathrel{\widehat{=}}$ **Refines** $ReassignDoctor$
  **any** $ma, ma\_new$
  **when**
    $ma \in ran(assigned\_doctor) \wedge ma \notin last\_visited[visited] \wedge$
    $ma\_new \in med\_agents \wedge ma\_new \neq ma \wedge$
    $ma \in disconnected \wedge timer(ma) = timeout \wedge$
    $(ma\_new \notin disconnected \vee (ma\_new \in disconnected \wedge timer(ma\_new) = active))$
  **then**
    $med\_agents := med\_agents \setminus \{ma\} \parallel$
    $assigned\_doctor := assigned\_doctor \mathbin{<\!\!+} (dom(assigned\_doctor \rhd \{ma\}) \times \{ma\_new\}) \parallel$
    $disconnected := disconnected \setminus \{ma\} \parallel timer := \{ma\} \mathbin{\lhd\!\!\!-} timer$
  **end**
```

As a result of the second refinement step we have ensured that no patients are left unattended neither because of the doctors shift change nor because of a doctor agent failure.

4.4 Ensuring Safety of Cooperative Agent Actions

Our next refinement step introduces abstract modelling of emergency calls, which are generated by patient monitoring equipment. We must guarantee that each call will be properly handled by a corresponding doctor. Hence our next refinement step should transform the specification to ensure the following property:

$$\Box(\text{an emergency call by a patients} \leadsto \text{a medical visit happens})$$

In this refinement step, we introduce the variable *emergency_calls*, which is defined as a partial function associating the emergency calls with the patients:

$$emergency\_calls \in ALARMS \rightarrowtail\!\!\!\!\!\!+\; patients$$

Moreover, we define the variable *accepted_calls* that establishes the correspondence between the emergency calls and the doctors that answer them:

$$accepted\_calls \in ALARMS \nrightarrow med\_agents$$

At this refinement step we abstract away from an actual implementation of how a doctor handling an emergency call is chosen. A detailed model of this will be introduced at the next refinement step. Here we add new events *EmergencyCall* and *HandlingEmergencyCall* to abstractly model the occurrence of an emergency and finding a responsible doctor to handle it:

```
$EmergencyCall \mathrel{\widehat{=}}$
  Status $convergent$
  any $pa, ec$
  when
    $pa \in patients \wedge ec \in ALARMS \wedge$
    $ec \notin dom(emergency\_calls) \wedge pa \notin ran(emergency\_calls)$
  then
    $emergency\_calls := emergency\_calls \cup \{ec \mapsto pa\}$
  end
$HandlingEmergencyCall \mathrel{\widehat{=}}$
  Status $convergent$
  any $ec, ma$
  when
    $ec \in dom(emergency\_calls) \wedge ec \notin dom(accepted\_calls) \wedge$
    $ma \in med\_agent \wedge ma \notin disconnected$
  then
    $accepted\_calls := accepted\_calls \cup \{ec \mapsto ma\}$
  end
```

While refining a system, the developer can mistakenly introduce new (undesirable) deadlocks. To prevent this we use model variants – natural number expressions on the model variables. The additional proof obligation requires an explicit demonstration of convergence (termination) of the newly introduced events. Essentially, a developer should prove that the variant expression is decreased after an event execution.

In our case, we define a specific system variant to ensure that the newly introduced events *EmergencyCall* and *HandlingEmergencyCall* do not take the control forever. Those events have status *convergent*. We define the variant as follows:

$$card(ALARMS \setminus dom(emergency\_calls)) + card(ALARMS \setminus dom(accepted\_calls)),$$

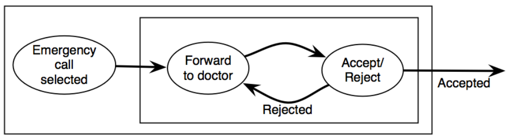
and prove that it is decreased by new events. The variants play an important role in guaranteeing, e.g., that error recovery terminates, service requests are eventually served etc. In our case the variant allows us to guarantee that eventually some doctor is chosen to handle an emergency. Obviously, this has important safety implications.

Let us mention that at this refinement step we also distinguish two types of a patient visit – a regular visit (a scheduled examination or a delivery of a medicine) and a visit for handling an emergency call. To model this, we decompose the event *VisitBegin* into the events *RegularVisitBegin* and *EmergencyVisitBegin*. The event *EmergencyVisitBegin* is specified as follows:

```
$EmergencyVisitBegin \mathrel{\widehat{=}}$ **Refines** $VisitBegin$
  **any** $ec$
  **when**
    $ec \in dom(accepted\_calls) \land emergency\_calls(ec) \notin visited \land$
    $accepted\_calls(ec) \notin last\_visit[visited]$
  **with**
    $ma = accepted\_calls(ec)$
    $pa = emergency\_calls(ec)$
  **then**
    $last\_visit(emergency\_calls(ec)) := accepted\_calls(ec) \parallel$
    $visited := visited \cup \{emergency\_calls(ec)\} \parallel$
    $emergency\_calls := emergency\_calls \setminus \{ec \mapsto emergency\_calls(ec)\} \parallel$
    $accepted\_calls := accepted\_calls \setminus \{ec \mapsto accepted\_calls(ec)\}$
  **end**
```

In current refinement step we have introduced modelling of emergency calls and non-deterministic assignment of responsible doctors to handle them. The goal of our next refinement step is to introduce a detailed procedure of selecting a doctor in the case of an emergency call. It follows the steps graphically depicted in Figure 2.

Figure 2 Procedure of choosing a doctor for a certain call



The proposed procedure can be described as follows. We start by selecting an emergency call to handle. Then we model the process of finding a suitable candidate and sending a request to him/her. If the doctor rejects it then we choose the next candidate. The procedure is repeated until we get an acceptance of the request.

To model the described procedure, we introduce a number of new variables and events to specify the corresponding steps of the selection procedure. The event *ChooseCurrentCall* models the beginning of handling of a particular emergency call. The event *CallFeed* directs the call to the assigned doctor, while *ForwardCall* forwards the call to next suitable candidate:

```
$CallFeed \mathrel{\widehat{=}}$
  **when**
    $ec\_handling = TRUE \land candidate\_found = FALSE \land$
    $assigned\_doctor(emergency\_calls(current\_call)) \notin disconnected \land$
    $assigned\_doctor(emergency\_calls(current\_call)) \notin occupied$
  **then**
    $directed(current\_call) := assigned\_doctor(emergency\_calls(current\_call)) \parallel$
    $candidate\_found := TRUE$
  **end**
$ForwardCall \mathrel{\widehat{=}}$
  **any** $ma\_new$
  **when**
    $ec\_handling = TRUE \land candidate\_found = FALSE \land$
    $(assigned\_doctor(emergency\_calls(current\_call)) \in disconnected \lor$
    $assigned\_doctor(emergency\_calls(current\_call)) \in occupied) \land$
    $ma\_new \in med\_agents \land ma\_new \notin disconnected \land ma\_new \notin occupied$
  **then**
    $directed(current\_call) := ma\_new \parallel candidate\_found := TRUE$
  **end**
```

Let us note that a doctor agent can refuse to accept a call. For instance, by checking the doctor schedule it might discover that she is currently performing a scheduled surgery.

The events *AcceptCall* and *RejectCall* model acceptance and rejection of the call respectively. The event *AcceptCall* is specified as follows:

```
$AcceptCall \mathrel{\widehat{=}}$ **Refines** $HandlingEmergencyCall$
  **when**
    $ec\_handling = TRUE \wedge candidate\_found = TRUE$
    $current\_call \in dom(emergency\_calls) \wedge current\_call \notin dom(accepted\_calls)$
    $directed(current\_call) \notin disconnected$
  **with**
    **ec:** $ec = current\_call$
    **ma:** $ma = directed(current\_call)$
  **then**
    $accepted\_calls(current\_call) := directed(current\_call) \parallel ec\_handling := FALSE$
    $candidate\_found := FALSE \parallel occupied := \varnothing$
  **end**
```

We define a special event *ForcedAcceptCall* to "force" the last available doctor to accept the call. To make this decision, the variable *occupied* is used to accumulate the id's of doctors that have already refused the call. We assume here that the whole procedure of finding a doctor in respond to a certain emergency call takes a short period of time and during this period no disconnection of agents can occur. As a result, we strengthen the guards in the event *DisconnectAgent* to disallow any disconnection while an emergency call is handled.

While verifying correctness of this refinement step, we encounter a problem with the system requirements – we cannot guarantee safety unless we assume that during the search for a doctor no disconnection of agents can occur. Hence, our system should ensure (e.g., by implementing a certain protocol) that finding a doctor takes a very short period of time. Moreover, we define an additional system variant $card(med\_agents \setminus occupied)$ to ensure that the event *RejectCall* is convergent, which means that eventually we should get an acceptance from a doctor to serve the call.

4.5 Data integrity

To ensure that a patient gets a correct treatment, we should guarantee that the medical personnel always access the most recent patient record. As we discussed in Section 4.1, we allow a doctor to access and modify the patient's data only when he/she is in a close proximity to the patient. We implement this requirement via the scoping mechanism (Laibinis et al., 2006, 2008, 2009). A scope provide a shared data space for a doctor and a patient. We assume that each patient agent has the scope associated with it. As soon as a doctor agent appears at a close vicinity of the patient agent, it automatically joins the scope. While in the scope, the doctor can modify the patient record (e.g., prescribe a new medicine, log the information about the delivered medicine, prescribe a new procedure and so on).

To model this behaviour, we refine the abstract events *RegularVisitBegin*, *EmergencyVisitBegin*, *VisitEnd* by the events *RegularEnterScope*, *EmergencyEnterScope*, *LeaveScope* and add a new event *ModifyRecord*. Moreover, we define the variable *scopes*, which is defined as a partial function associating the active scopes with the doctors participating in them:

$$scopes \in ScopeName \rightarrowtail\!\!\!\!\!\!\!\shortmid\;\; med\_agents$$

Furthermore, we define an invariant to illustrate preservation of the *Property 1*: only active medical agents are eligible to enter the scope:

$$\forall ma \cdot ma \in disconnected \Rightarrow ma \notin ran(scopes)$$

Also we introduce the variable *record* that represents the medical history for every patient:

$$record \in patients \rightarrow \mathbb{P}(DATA)$$

The variable *ma_data* stores the data that appear on the doctor's PDA screen:

$$ma\_data \in med\_agents \nrightarrow \mathbb{P}(DATA)$$

When the doctor agent is in a close vicinity of a patient, its *ma_data* becomes equal to the value of the patient data. The event *ModifyRecord* models an update of the patient record by a doctor, when he/she is in the scope of a patient:

```
$ModifyRecord \widehat{=}$
  any $ma, sn, pa, da\_new$
  when
    $(sn \mapsto ma) \in scopes \wedge pa \in dom(last\_visit) \wedge pa \in visited \wedge$
    $last\_visit(pa) = ma \wedge da\_new \in \mathbb{P}(DATA) \wedge da\_new \neq \varnothing$
  then
    $ma\_data(ma) := da\_new \parallel record(pa) := da\_new$
  end
```

Hereby we ensure that the patient record is always up-to-date. The corresponding safety property stating that the medical personnel always access the most recent record is formulated as the invariant:

$$\forall ma, pa \cdot (pa \mapsto ma) \in (visited \lhd last\_visit) \Rightarrow ma\_data(ma) = record(pa).$$

4.6 Decomposition

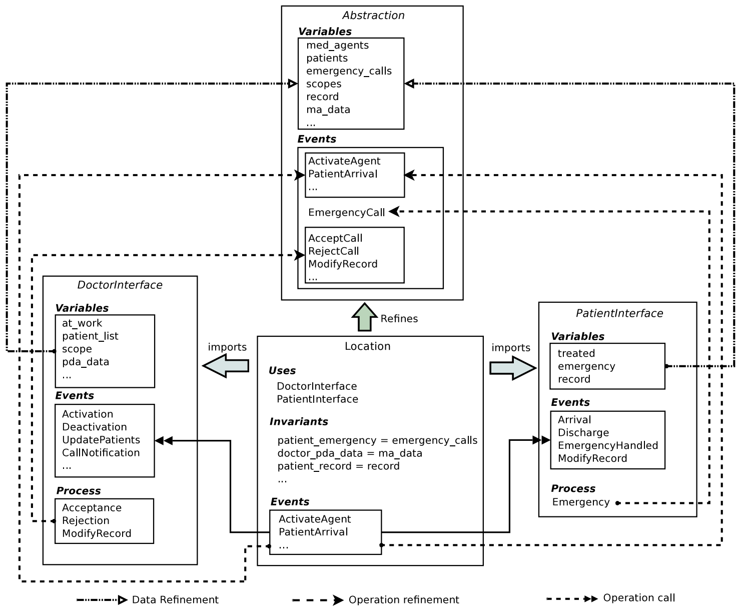
As a result of our previous refinement steps, we have arrived at a centralised model – *Hospital*5 of the Hospital MAS. In the final refinement step we aim at deriving a distributed architecture. Our goal is to decompose *Hospital*5 into separate modules representing the middleware, the doctor and patient agents and explicitly model communication by message passing.

A graphical representation of the system after decomposition refinement is shown in Figure 3. The abstract model is refined by a model representing the *Location* part – the middleware, and two separate modules – *Doctor* and *Patient*. Each modules contain callable operations and both internal and external data. The *Location* part accesses the modules via the provided generic interfaces.

All the variables and most of operations of the interface corresponding to the doctor and patient agents appear with the prefixes *doctor* and *patient* respectively. This is a feature of the modularisation extension that helps to avoid name clashes when importing several interfaces.

The *Location* model imports two module interfaces – *Doctor* and *Patient*. The majority of the events of the machine *Hospital5* are now refined by the location events. The remaining events, including *AcceptCall*, *RejectCall*, *EmergencyCall* become a part of the autonomous processes of the doctor or patient modules. The abstract event *EmergencyCall* is refined by the patient's interface event *Emergency* implementing the occurrence of emergency calls. Similarly, the abstract events *AcceptCall*, *RejectCall*, *ModifyRecord* are refined by the corresponding doctors's interface events *Acceptance*, *Rejection* and *ModifyData*.

Figure 3 Illustration of decomposition refinement



The variables of the centralised model *Hospital5* are now refined by the variables of the Location, patient and doctor modules. For instance, the abstract variable *emergency_calls* is replaced by a new variable from the patient interface *patient_emergency*. Similarly, the abstract variable *doctor_pda* is refined by a new variable from the doctor interface *doctor_ma_data*. An extract from the *PatientInterface* is shown below:

```
Interface $PatientInterface$
  Variables $treated, emergency, record$
  Invariants
    $inv_1 : treated \in PATIENTS \rightarrow BOOL$
    $inv_2 : emergency \in ALARMS \rightarrowtail PATIENTS$
    $inv_3 : record \in PATIENTS \nrightarrow \mathbb{P}(DATA)$
  Process
    $Emergency$
      any $pa, ec$
      when
        $pa \in dom(treated) \wedge treated(pa) = TRUE \wedge ec \in ALARMS$
        $ec \notin dom(emergency) \wedge pa \notin ran(emergency)$
      then
        $emergency := emergency \cup \{ec \mapsto pa\}$
      end
  Operations
    $Arrival$
      any $pa, da$
      pre $pa \in PATIENTS \wedge treated(pa) = FALSE \wedge da \subseteq DATA$
      return $void$
      post
        $void' \in VOID \wedge treated' = treated \lhd\!\!- \{pa \mapsto TRUE\} \wedge record' = record \lhd\!\!- \{pa \mapsto da\}$
      end
      ...
```

Let us describe communication between the agents by an example – a patient arrival. At our previous refinement step this activity is modelled by the event *PatientArrival*. Essentially a patient arrival corresponds to registering a patient by the location (middleware). By performing certain internal computations, the location assigns a doctor to a patient and confirms a successful registration by sending the corresponding messages to the patient and the doctor. This behaviour is specified in the refined event *PatientArrival* is presented below:

```
$PatientArrival \;\widehat{=}\;$ **Refines** $PatientArrival$
  **any** $ma, pa, da$
  **when**
    $pa \in PATIENTS \wedge pa \notin patients \wedge ma \in med\_agents \wedge da \subseteq DATA$
  **then**
    $patients := patients \cup \{pa\} \parallel assigned\_doctor(pa) := ma \parallel$
    $void := patient\_Arrival(pa \mapsto da) \parallel void2 := doctor\_UpdatePatients(ma \mapsto \{pa\})$
  **end**
```

The calls to the corresponding operations from the patient and doctor interfaces *patient_Arrival(...)* and *doctor_UpdatePatients(...)* model the communication between the location and agents respectively. The other events are decomposed in a similar way.

As a result of the decomposition we arrive at a specification of distributed hospital MAS. Each type of agents – doctors and patients – are represented by the corresponding module. The location serves as a communication infrastructure.

5 Conclusion

In this paper we have presented a formal approach to developing MAS by refinement in Event-B. We formalised the main properties of MAS and demonstrated how refinement process can facilitate their preservation. Finally, we illustrated the proposed approach by a large case study – development of a hospital MAS. In our case study we focused on modelling and verification of safety of two central critical activities – handling emergencies and consistent updates of patient data. Ensuring correctness of these activities was especially challenging due to highly dynamic nature of a hospital, volatile error-prone communication environment and autonomous agent behaviour.

In our development we have explicitly modelled the fault tolerance mechanism that guarantee correct system functioning in the presence of agent disconnections. We have verified by proofs the correctness and safety of these two activities. Formal verification process has not only allowed us to systematically capture complex requirements but also facilitated derivation of the constraints that should be imposed on the system to guarantee its safety. Indeed, for instance, while proving convergence of the emergency handling procedure, we had to explicitly state the assumptions that the system must fulfil. These assumptions can be seen as a contract that should be checked during system deployment to guarantee its safety. In our development we have also demonstrated that the scoping mechanism provides a useful abstraction for ensuring consistent updates of the patient data.

5.1 Related work

The work presented in this paper is inspired by a related work on modelling context-aware mobile agent systems by Laibinis et al. (2006, 2008, 2009) in the CAMA framework (Iliasov & Romanovsky, 2005, Iliasov et al., 2006). Similarly to Laibinis et al. (2006, 2008, 2009), we rely on a timeout mechanism to tolerate agent disconnections and employ a scoping mechanism to provide shared data space for patient and doctor agents. However, in this paper we have focused on modelling and verification of safety properties of complex agent interactions rather than on reasoning about general mechanisms for agent interaction with middleware.

A problem of modelling fault tolerance MAS has been addressed by Ball & Butler (2009). They illustrated how certain typical fault tolerance mechanisms can be incorporated into a MAS specification. In our approach we consider fault tolerance as a part of ensuring safety of MAS.

Formal modelling of MAS has been undertaken by Roman et al. (2007, 2004, 1997). The authors have proposed an extension of the Unity framework to explicitly define such concepts as mobility and context-awareness. In our approach we also have studied the problem of ensuring access to the fresh context. However, in Roman et al. (2007) it is solved at the level of the matching agent attributes while in our approach we rely on the scoping mechanism to achieve this.

A formal modelling of MAS for the health care in Z has be undertaken by Gruer et al. (2002). The work has focused on specifying a multi-agent system for a medical help system. The authors aimed at studying how to formally represent agent interactions, e.g., during negotiations. In our approach we not only model the agent interactions but also formally prove their properties. Hence, our approach is especially suitable for developing critical MAS.

Our approach is different from numerous process-algebraic approaches used for modelling MAS. Firstly, we have relied on proof-based verification that does not impose restrictions on the size of the model, number of agents etc. Secondly, we have adopted a system's approach, i.e., we modelled the entire system and extracted specifications of its individual components by decomposition. Such an approach allows us to express and formally verify safety of the overall system, i.e., we achieve verification of safety as a *system level* property. Finally, the adopted top-down development paradigm has allowed us to efficiently cope not only with complexity of requirements but also with complexity of verification. We have build a large formal model of a complex system by a number of rather small increments. As a result, verification efforts have been manageable because we merely needed to prove refinement between each two adjacent levels of abstraction. Hence, we conclude that refinement in Event-B constitutes a suitable technique for formal modelling and verification of critical multi-agent systems.

As a future work we are planning to generalise the proposed approach and extract modelling patterns that can be used to automate formal development. Moreover, we will investigate how to model adaptive agent behaviour that depends on the surrounding context and various reconfiguration mechanisms.

References

Abrial, J.-R. (2005), *The B-Book: Assigning Programs to Meanings*, Cambridge University Press.

Abrial, J.-R. (2010), *Modeling in Event-B*, Cambridge University Press.

Ball, E. & Butler, M. (2009), Event-B Patterns for Specifying Fault-Tolerance in Multi-Agent Interaction, *in* C. J. M. Butler & A. Romanovsky, eds, 'Methods, Models and Tools for Fault Tolerance', Vol. 5454 of *LNCS*, Springer, pp. 104–129.

Deploy Project (2008). [online] Available at http://www.deploy-project.eu/ (Accessed 11 November 2011).

Gruer, P., Hilaire, V., Koukam, A. & Cetnarowicz, K. (2002), A formal framework for multi-agent systems analysis and design, *in* ‘Expert Systems with Applications’, Vol. 23, pp. 349–355.

Iliasov, A. & Romanovsky, A. (2005), CAMA: Structured Coordination Space and Exception Propagation Mechanism for Mobile Agents, *in* ‘ECOOP 2005, Workshop on Exception Handling in Object Oriented Systems: Developing Systems that Handle Exceptions’.

Iliasov, A., Khomenko, V., Koutny, M. & Romanovsky, A. (2006), On Specification and Verification of Location-based Fault Tolerant Mobile Systems, *in* A. R. M. Butler, C. Jones & E. Troubitsyna, eds, ‘Rigorous Development of Complex Fault-Tolerant Systems’, Vol. 4157 of *LNCS*, Springer, pp. 168–188.

Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A., Varpaaniemi, K., Ilic, D. & Latvala, T. (2010), Supporting Reuse in Event B Development: Modularisation Approach, *in* ‘Proceedings of Abstract State Machines, Alloy, B, and Z (ABZ 2010)’, Vol. 5977, Lecture Notes in Computer Science, Springer, pp. 174–188.

Laibinis, L., Troubitsyna, E., Iliasov, A. & Romanovsky, A. (2006), Rigorous Development of Fault-Tolerant Agent Systems, *in* A. R. M. Butler, C. Jones & E. Troubitsyna, eds, ‘Rigorous Development of Complex Fault-Tolerant Systems’, Vol. 4157 of *LNCS*, Springer, pp. 241–260.

Laibinis, L., Troubitsyna, E., Iliasov, A. & Romanovsky, A. (2008), Formal Development of Cooperative Exception Handling for Mobile Agent Systems, *in* ‘SERENE 2008, International Workshop on Software Engineering for Resilient Systems’.

Laibinis, L., Troubitsyna, E., Iliasov, A. & Romanovsky, A. (2009), Fault Tolerant Middleware for Agent Systems: A Refinement Approach, *in* ‘EWDC 2009, European Workshop on Dependable Computing’.

OMG Mobile Agents Facility (1997), [online] Available at www.omg.org (Accessed 11 November 2011).

RODIN Modularisation Plug-in (2010). [online] Available at www.omg.org http://wiki.event-b.org/index.php/Modularisation_Plug-in (Accessed 11 November 2011).

Rodin Platform (2006). [online] Available at http://www.event-b.org/ (Accessed 11 November 2011).

Roman, G.-C., Julien, C. & Payton, J. (2004), A Formal Treatment of Context-Awareness, *in* ‘FASE’2004’, Vol. 2984 of *LNCS*, Springer.

Roman, G.-C., Julien, C. & Payton, J. (2007), Modeling adaptive behaviors in Context UNITY, *in* ‘Theoretical Computure Science’, Vol. 376, pp. 185–204.

Roman, G.-C., P.McCann & Plun, J. (1997), Mobile UNITY: Reasoning and Specification in Mobile Computing, *in* ‘ACM Transactions of Software Engineering and Methodology’.