Fault Tolerant Middleware for Agent Systems: A Refinement Approach

Linas Laibinis, Elena Troubitsyna, Alexei Iliasov, Alexander B. Romanovsky

Fault Tolerant Middleware for Agent Systems: A Refinement Approach

Linas Laibinis, Elena Troubitsyna
Åbo Akademi University, Finland
{Linas.Laibinis,Elena.Troubitsyna}@abo.fi

Alexei Iliasov, Alexander Romanovsky
University of Newcastle upon Tyne, UK
{Alexei.Iliasov,Alexander.Romanovsky}@ncl.ac.uk

***Abstract*—Agent technology offers a number of advantages over traditional distributed systems, such as asynchronous communication, anonymity of individual agents and ability to change operational context. However, it is notoriously difficult to ensure dependability of agent systems. In this paper we present a formal approach for the top-down development of fault tolerant middleware for agent systems. We demonstrate how to develop the middleware that besides providing agent coordination is also able to cope with their failures. We focus on handling agent crushes and transient faults caused by volatile communication environment. We argue that formal development of middleware with integrated fault tolerance mechanisms has potential to enhance dependability of an agent system.**

***Index Terms*—middleware, formal development, fault tolerance, refinement**

I. Introduction

Mobile agent systems are complex decentralised distributed systems composed of agents asynchronously communicating with each other. Usually agents – computer programs acting autonomously on behalf of a person or organisation [11] and implementing autonomous communicating functionality – are designed independently from each other by different developers. The agent technology naturally solves the problem of partitioning complex software into smaller parts that are easier to analyse, design and maintain. Currently mobile agent systems are rapidly spreading into business- and safety-critical domains. However, complexity caused by openness, mobility, dynamic and distributed nature of such systems poses a serious challenge to ensuring system dependability. Therefore, there is clear need for structured rigorous approaches for designing dependable agent systems.

In this paper we present a formal approach to developing fault tolerant middleware for agent system. We demonstrate how to formally specify and develop by refinement middleware that provides support not only to normal agent activities but also handles agent crushes, temporal agent disconnections, cooperative work of agents during error recovery etc. We demonstrate that by developing system by refinement in Event B [2] we can introduce fault tolerance mechanisms into the system specification in a structured way. Formal verification allows us to guarantee such essential properties, e.g., as termination of error recovery. We argue that rigorous model-driven development by refinement in Event B enables formal verification of complex fault tolerant agent systems and hence allows the developers to enhance system dependability.

II. CAMA systems

In this paper we model middleware supporting CAMA (Context Aware Mobile Agent) systems [3], [8]. The CAMA inter-agent communication is based on the LINDA paradigm [7], which provides a set of language-independent coordination primitives that can be used for coordination of several independent agents. These primitives allow agents to put tuples (vectors of values) in a shared tuple space, remove them from it, and test the shared space for tuple presence.

The major contribution of CAMA is a novel mechanism to structure inter-agent communication, allowing groups of communicating agents to work in isolated subspaces called *scopes*. A scope is a dynamic container for tuples. The tuples contained within a scope are visible only to the agents participating in this particular scope. Hence, a scope provides an isolated coordination space for its agents.

Besides isolation of a communication space, scopes also provide a basis for dynamic type-checking of multi-agent applications. More precisely, each agent carries special attributes describing the functionality it implements. Such an abstract description of agent functionality is called *role*. Each role is associated with some abstract scope model and the scoping mechanism permits collaboration between agents playing compatible roles. An agent might implement a set of roles and can also take several roles within the same or different scopes.

In its turn, a *location* is a container for scopes. In addition to supporting scopes as the means of agent communication, locations may also offer support for logical mobility of agents, hosting of an agent or an agent backup.

Finally, *platform* provides an execution environment for agents. It is composed of a virtual machine for code execution, networking support, and client middleware for interacting with a location.

A typical behaviour of an agent can be described as follows: an agent connects to a location and then joins an existing scope. In a scope it can cooperate with agents participating in the same scope. When an agent leaves the scope, it either joins

another scope or disconnects from the location. To support this behaviour the CAMA middleware provides three categories of operations: location engagement, scoping mechanism, and communication. The communication operations implement the standard LINDA [7] coordination paradigm. The location engagement operations associate or disassociate an agent with a location. The scoping mechanism operations allow an agent to enquiry for available scopes, create new scopes, destroy previously created scopes, join and leave existing scopes.

Agents systems operate in volatile error-prone communication environment. Hence CAMA middleware should be able to handle not only normal agent activities but also deal with agent failures. This includes tolerating transient faults caused by temporal agent disconnections, agent crushes, maintaining correct operation within scopes during error recovery etc. Obviously, such a complex behaviour should be modelled and verified in a structured and rigorous way. Next we briefly present Event-B – our framework for formal modelling and development of fault tolerant CAMA middleware.

III. FORMAL MODELLING AND REFINEMENT IN EVENT B

The Event B framework [2] extends the B Method [1] to enable development of distributed, parallel and reactive systems. Tool support available for Event B – Atelier B [4] – provides us with the facilities to specify a system, verify it and formally develop by stepwise refinement Atelier B uses theorem proving as a main verification technique. It has achieved a high degree of automation in generating proofs and discharging them while verifying correctness of system specification and refinement. This improves scalability of formal modelling and speeds up development.

B adopts the top-down approach to system development. The development starts from creating a formal system specification. A formal specification is a mathematical model of the required behaviour of a system, or a part of a system. In B, a specification is represented by a collection of modules, called Abstract Machines. The Abstract Machine Notation (AMN), is used in constructing and verifying them. An abstract machine encapsulates a state (the variables) of the machine and provides operations on the state. A simple abstract machine has the following general form:

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

The machine is uniquely identified by its name *AM*. The state variables of the machine, *v*, are declared in the **VARIABLES** clause and initialised in *INIT* as defined in the **INITIALISATION** clause. The variables in B are strongly typed by constraining predicates of the machine invariant *I* given in the **INVARIANT** clause. The invariant is usually defined as a conjunction of the constraining predicates and the predicates defining the properties of the system that should be preserved during system execution. All types in B are represented by non-empty sets. Local types can be introduced by enumerating the elements of the type, e.g., $TYPE \quad = \quad \{element1, element2, \ldots\}$, or by defining them as subsets of already existing types or sets.

The events specified in the **EVENTS** clause define the dynamic behaviour of the system. The events are atomic meaning that, once an event is chosen, its execution will run until completion without interference. An event is defined as follows:

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

where the guard *g* is a predicate over the state variables *v*, and the body *S* is a B statement specifying how *v* are affected by execution of the event.

Several events can be grouped together in an *array of events*.

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

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

The occurrence of events represents the observable behaviour of the system. The guard defines the conditions under which the body can be executed, i.e., when the event is *enabled*. If several events are enabled simultaneously then one of them is non-deterministically chosen for execution. If no event is enabled (the guard of each event evaluates to *false*) then the system deadlocks, i.e., stops its execution.

B statements that we will use to describe the body of the events have the following syntax:

$$\begin{array}{lll} S & == & x := e \mid \textbf{IF } cond \textbf{ THEN } S1 \textbf{ ELSE } S2 \textbf{ END} \mid \\ & & x \;::\; T \mid \textbf{ANY } z \textbf{ WHERE } Q \textbf{ THEN } S \textbf{ END} \mid \\ & & S1 \;\textbf{;}\; S2 \mid S1 \parallel S2 \mid \ldots \end{array}$$

The first three constructs - an assignment, a conditional statement and sequential composition have the standard meaning. Sequential composition is disallowed in abstract specifications but permitted in refinements. The remaining constructs allow us to model non-deterministic or parallel behaviour in a specification. Usually they are not implementable so they have to be refined (replaced) with executable constructs at some point of program development. We use two kinds of non-deterministic statements - the non-deterministic assignment and the non-deterministic block. The non-deterministic assignment $x \;::\; T$ assigns the variable *x* an arbitrary value from the given set (type) *T*. The non-deterministic block **ANY** z **WHERE** Q **THEN** S **END** introduces the new local variable z which is initialised (possibly nondeterministically) according to the predicate *Q* and then used in the statement *S*. Finally, $S1 \parallel S2$ models parallel (simultaneous) execution of $S1$ and $S2$ provided $S1$ and $S2$ do not have a conflict on state variables. The special case of the parallel execution is a multiple assignment which is denoted as $x, y \;:=\; e1, e2$.

The B statements are formally defined using the weakest precondition semantics [5]. The weakest precondition semantics provides us with a foundation for establishing correctness

of specifications and verifying refinements between them. For instance, we verify correctness of specification by proving that initialization and all events establish the invariant.

The basic idea underlying formal stepwise development by refinement is to design the system implementation gradually, by a number of correctness preserving steps, called *refinements*. The refinement process starts from creating an abstract, albeit unimplementable, specification and finishes with generating executable code. The intermediate stages yield the specifications containing a mixture of abstract mathematical constructs and executable programming artifacts. At each refinement step we define a gluing invariant that connects state spaces of more abstract and more concrete models. We use a gluing invariant to verify correctness of refinement, i.e., to prove that each refined model adheres to the more abstract one.

In general, a refinement process can be seen as a way to reduce non-determinism of the abstract specification, to replace abstract mathematical data structures by data structures implementable on a computer, and, hence, gradually introduce implementation decisions.

Next we discuss modelling and refinement of the CAMA middleware using the Event B formalism.

IV. FORMAL SPECIFICATION OF AGENT SYSTEMS

A. The System Approach to Modelling Agent Systems

In this paper we demonstrate formal development of CAMA middleware. We focus on developing a part of CAMA middleware supporting activities of an agent in a location, though the part of middleware supporting inter-location activities can also be developed using our approach.

Our development starts from an abstract specification given in the machine *Cama*, which models the entire agent system, i.e., the agents and the location together. By adopting such a *systems* approach we can define the essential properties of the overall agent system, derive the properties to be satisfied by a location and each agent, and ensure that they are preserved in the agent and location development, i.e., we can address the problem of ensuring interoperability of agents.

The variable *agents* represents the set of agents that joined the location. The operations *Engage* and *Disengage* model joining and leaving the location correspondingly. While an agent is in the location, it performs some computations, as modelled by the operation *NormalActivity*. To express that these computations are performed locally within the agent and hence do not affect the abstract state of the system, we model them by the statement *skip*.

MACHINE *Cama*
SETS *Agents*
VARIABLES *agents*
INVARIANT $agents \subseteq Agents$
INITIALISATION $agents := \emptyset$
EVENTS
Engage = ANY *aa* **WHERE** $aa \in Agents \wedge aa \notin agents$
THEN $agents := agents \cup \{aa\}$ **END**;
NormalActivity = ANY *aa* **WHERE** $aa \in Agents \wedge aa \in agents$
THEN skip END ;
Disengage = ANY *aa* **WHERE** $aa \in Agents \wedge aa \in agents$
THEN $agents := agents - \{aa\}$ **END**
END

We aim at creating a methodology enabling systematic integration of the fault tolerance mechanisms into the development of CAMA systems. The most typical faults that these systems encounter are temporal losses of connection, which can cause failures of communication between cooperating agents or between an agent and the location.

In our initial specification we abstracted away from explicit modelling of the system behaviour in the presence of faults. Although, the result of failure – disengagement of an agent from the location – is implicitly modelled in the operation *Disengage*. In our first refinement step we introduce an explicit representation of the system behaviour in the presence of temporal losses of connection.

Let us observe that in most cases an agent loses connection only for a short period of time. After connection is restored, the agent is willing to continue its activities virtually uninterrupted. Therefore, after detecting a connection loss, 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 activity. However, if the agent fails to do it, the location should disengage the agent.

Such a behaviour can be adequately modelled by the timeout mechanism. Upon detecting a disconnection the location activates a timer. If the agent reconnects before the timeout then the timer is stopped. Otherwise, the location forcefully disengages the disconnected agent. To model this behaviour, in the first refinement step we introduce the variable *timers* representing the subset of agents that have disconnected but for which the timeouts have not expired yet. Moreover, we introduce the variable ex_agents to model the subset of agents that missed their reconnection deadline and should be disengaged from the location. Finally, we add the new events *Disconnect*, *Connect* and *Timer* to model agent disconnection, reconnection and timeout correspondingly.

To ensure that the refined system does not introduce additional deadlocks, we define the variant, which constrains the number of successive disconnections and reconnections. The constant $Disconn\_limit$ defines the maximal number of successive disconnections. The variable $disconn\_limit$ obtains the value $Disconn\_limit$ in the initialisation. Each newly introduced event decreases the value of the variant either by decreasing the value of $disconn\_limit$ (when an agent disconnects) or by removing elements from the set *timers* (when a disconnected agent either reconnects or misses the reconnection deadline). The value of the variant is restored by executing the *NormalActivity* event.

In our specification we assume that an agent failure due to the loss of connection is detected by the location. However, an agent might by itself detect an error in its functioning and leave the location. Therefore, the agent might get disengaged from the location due to the following three reasons:

- because it has successfully completed its activities in the location,
- due to the disconnection timeout,
- due to a spontaneous failure detected by the agent itself.

In the refined specification given in the machine $Cama1$ below, we model all these different types of leaving by splitting the operation *Disengage* into three corresponding operations: *NormalLeaving*, *TimerExpiration* and *AgentFailure*.

REFINEMENT *Cama1*
REFINES *Cama*
CONSTANTS *Disconn_limit*
PROPERTIES *Disconn_limit* $\in$ **NAT** $\wedge$ *Disconn_limit* $>$ *1*
INITIALISATION
agents := $\emptyset$ || *timers* := $\emptyset$ ||
ex_agents := $\emptyset$ || *disconn_limit* := *Disconn_limit*
VARIABLES *agents*, *timers*, *ex_agents*, *disconn_limit*
INVARIANT
timers $\subseteq$ *agents* $\wedge$
ex_agents $\subseteq$ *agents* $\wedge$
timers $\cap$ *ex_agents* = $\emptyset$ $\wedge$
disconn_limit $\in$ **NAT**
VARIANT
card(*timers*) + *2*disconn_limit*

EVENTS

Engage = **ANY** *aa* **WHERE** *aa* $\in$ *Agents* $\wedge$ *aa* $\notin$ *agents*
THEN *agents* := *agents* $\cup$ $\{aa\}$ **END**;

NormalActivity = **ANY** *aa* **WHERE** *aa* $\in$ *agents*
THEN *disconn_limit* := *Disconn_limit* **END**;

NormalLeaving `ref` **Disengage** = **ANY** *aa* **WHERE**
(*aa* $\in$ *agents)* $\wedge$ *(aa* $\notin$ *timers)* $\wedge$ *(aa* $\notin$ *ex_agents)*
THEN *agents* := *agents* - $\{aa\}$ **END**;

TimerExpiration `ref` **Disengage** = **ANY** *aa* **WHERE**
(*aa* $\in$ *agents)* $\wedge$ *(aa* $\in$ *ex_agents)*
THEN *agents* := *agents* - $\{aa\}$ ||
ex_agents := *ex_agents* - $\{aa\}$ **END**;

AgentFailure `ref` **Disengage** = **ANY** *aa* **WHERE**
(*aa* $\in$ *agents)* $\wedge$ *(aa* $\notin$ *timers)* $\wedge$ *(aa* $\notin$ *ex_agents)*
THEN *agents* := *agents* - $\{aa\}$ **END**;

Connect = **ANY** *aa* **WHERE** (*aa* $\in$ *agents)* $\wedge$ *(aa* $\in$ *timers)*
THEN *timers* := *timers* - $\{aa\}$ **END**;

Disconnect = **ANY** *aa* **WHERE**
(*aa* $\in$ *agents)* $\wedge$ *(aa* $\notin$ *ex_agents)* $\wedge$ *(aa* $\notin$ *timers)* $\wedge$
disconn_limit $>$ 1
THEN *timers* := *timers* $\cup$ $\{aa\}$ ||
disconn_limit := *disconn_limit* - 1 **END**;

Timer = **ANY** *aa* **WHERE** (*aa* $\in$ *agents)* $\wedge$ *(aa* $\in$ *timers)*
THEN *ex_agents* := *ex_agents* $\cup$ $\{aa\}$ ||
timers := *timers* - $\{aa\}$ **END**
END

The refined specification $Cama1$ is a result of refinement of the abstract specification $Cama$. This refinement step allowed us to introduce both error detection and error recovery into the system specification. Hence, already at a high level of abstraction we specify fault tolerance as an intrinsic part of the system behaviour.

B. Introducing Scoping Mechanism

In the abstract specification and the first refinement step we mainly focused on modelling interactions of agents with the location. Our next refinement step introduces an abstract representation of the scopes as an essential mechanism that governs agent interactions while they are involved in cooperative activities.

The creation of a scope is initiated by an agent, which consequently becomes the scope owner. The other agents might join the scope and become engaged into the scope activities. The agents might also leave the scope at any instance of time. The scope owner cannot leave the scope but might close it (this action is not permitted for other agents). When the scope owner closes the scope, it forces all agents participating in the scope to leave.

The introduction of the scoping mechanism also enforces certain actions to be executed when an agent decides to leave a location. Namely, to leave the location an agent should first leave or close (if it is the scope owner) all the scopes in which it is active.

The scoping mechanism has deep impact on modelling error recovery in agent systems. For instance, if the scope owner irrecoverably fails, then, to recover the system from this error, the location should close the affected scope and force all agents in this scope to leave.

We refine the machine $Cama1$ to specify the scoping mechanism described above. In the refinement machine $Cama2$, we introduce the variable *scopes*, which is defined as a relation associating the active scopes with the agents participating in them. Moreover, we add the variable *sowner* to model scope owners. It is defined as a total function from the active scopes to agents.

We define the new events *Create*, *Join*, *Leave* and *Delete* to model creating a scope by the owner, joining and leaving it by agents, as well as closing a scope. In the excerpt[1] from the refinement machine $Cama2$, we demonstrate the newly introduced variables and events as well as the effect of the refinement on the events *AgentFailure* and *TimerExpiration*. The guard of the event *NormalLeaving* is now strengthened to disallow an agent to leave the location when it is still active in some scopes.

REFINEMENT *Cama2*
REFINES *Cama1*
SETS *ScopeName*
CONSTANTS *ScopeLimit*

[1]In this paper we are not presenting the B specifications in full length. The complete development can be found at [14].

PROPERTIES $ScopeLimit \in$ **NAT1**
DEFINITIONS $activeAgent(aa) == (aa \notin ex\_agents \wedge aa \notin timers)$
VARIABLES ... , *scopes, sowner, slimit*
INVARIANT
$scopes \in ScopeName \leftrightarrow agents \wedge$
$sowner \in ScopeName \nrightarrow agents \wedge$
dom(*sowner*) = **dom**(*scopes*) $\wedge$
$sowner \subseteq scopes \wedge$
$slimit \in$ **NAT**
VARIANT *slimit*

EVENTS
...
Create = **ANY** *aa*, *nn* **WHERE**
$(aa \in agents) \wedge (activeAgent(aa)) \wedge$
$(nn \in ScopeName) \wedge (nn \notin$ **dom**$(scopes)) \wedge$
$slimit > 0$
THEN
CHOICE
$scopes, sowner :=$
$scopes \cup \{nn \mapsto aa\}, sowner \cup \{nn \mapsto aa\}$
OR skip END ||
$slimit :=$ slimit - 1
END;

Join = **ANY** *aa*, *nn* **WHERE**
$(aa \in agents) \wedge (activeAgent(aa) \wedge$
$(nn \in$ **dom**$(scopes)) \wedge ((nn \mapsto aa) \notin scopes) \wedge$
$slimit > 0$
THEN
CHOICE
$scopes := scopes \cup \{nn \mapsto aa\}$
OR skip END ||
$slimit :=$ slimit - 1
END;

Leave = ...
Delete ...
NormalLeaving = ...

TimerExpiration = **ANY** *aa* **WHERE**
$aa \in agents \wedge aa \in ex\_agents$
THEN
$agents := agents - \{aa\}$;
$scopes := scopes \rhd \{aa\}$;
$scopes := sowner^{-1}[\{aa\}] \lhd scopes$;
$ex\_agents := ex\_agents - \{aa\}$;
$sowner := sowner \rhd \{aa\}$
END;

AgentFailure = **ANY** *aa* **WHERE**
$aa \in agents \wedge activeAgent(aa)$
THEN
$agents := agents - \{aa\}$;
$scopes := scopes \rhd \{aa\}$;
$scopes := sowner^{-1}[\{aa\}] \lhd scopes$;
$sowner := sowner \rhd \{aa\}$
END
END

Here R^{-1} denotes relational inverse for a given relation R, $a \mapsto b$ is mapping between elements $a \in A$ and $b \in B$, $U \lhd R$ is domain subtraction for a given relation R and a set U (i.e., all pairs in which the first element belongs to U are removed from R), and $R \rhd U$ is range subtraction correspondingly.

Let us observe that an agent does not always successfully create or join a scope. This is modelled by the *skip* statements in bodies of operations *Create* and *Join*. This might be caused by an attempt to join the scope with an incorrect role, as will be elaborated at the later refinement steps.

Termination of the added new events *Create*, *Join*, *Leave* and *Delete* is guaranteed by introducing the new variable *slimit*, which serves as the variant expression for the new event operations. As in the previous refinement step, the value of the variant expression is reset in the *NormalActivity* event.

This refinement step resulted in introduction of the general representation of the scoping mechanism in the system specification.

C. Introducing Error Recovery by Refinement

In our current specification the event *AgentFailure* treats any agent failure as an irrecoverable error. Indeed, upon detecting an error, the failed agent is removed from the scopes in which it participates and then disengaged from the location. However, usually upon detecting an error the agent at first tries to recover from it (possibly involving some other agents in the error recovery). If the error recovery eventually succeeds, then the normal operational state of the agent is restored. Otherwise, the error is treated as irrecoverable.

In our next refinement step, we introduce error recovery into our specification. We define the the variable *astate* to model the current state of the agent. The variable *astate* can have one of three values: *OK, RE* or *KO*, designating a fault free agent state, a recovery state, and an irrecoverable error correspondingly. We introduce the event *AgentRecoveryStart*, which is triggered when an agent becomes involved in the error recovery procedure. Observe that *AgentRecoveryStart* implicitly models two situations:

- when an agent itself detects an error and subsequently initiates its own error recovery,
- when an agent decides to become involved into cooperative recovery from another agent failure.

In both cases the state of the agent is changed from *OK* to *RE*.

The event *AgentRecovery* abstractly models the error recovery procedure. Error recovery might succeed and restore the fault free agent state *OK*, or continue by leaving an agent in the recovery state *RE*. Finally, error recovery might fail, as modelled by the event *AgentRecoveryFailure*. The event *AgentRecoveryFailure* enables the event *AgentFailure*, which removes the irrecoverably failed agent from the corresponding scopes and disengages it from the location.

The introduction of agent states affects most of the events – their guards become strengthened to ensure that only fault free agents can perform normal activities, engage into a

location and disengage from it, as well as create and close scopes. In the excerpt from the refinement machine $Cama3$, we present only the newly introduced events and the refined event *AgentFailure*.

REFINEMENT *Cama3*
REFINES *Cama2*
SETS *STATE* = {*OK*, *KO*, *RE*}
DEFINITIONS *activeAgent*(*xx*) == (*xx* $\notin$ *ex_agents* $\wedge$ *xx* $\notin$ *timers*)
VARIABLES ..., *astate*, *recovery_limit*
INVARIANT
...
astate $\in$ *agents* $\rightarrow$ *STATE* $\wedge$
recovery_limit $\in$ *agents* $\rightarrow$ **NAT**
VARIANT $\sum$ *aa*.(*aa* $\in$ *agents* | *recovery_limit*(*aa*))

EVENTS
...
AgentFailure = **ANY** *aa* **WHERE**
aa $\in$ *agents* $\wedge$ *activeAgent*(*aa*) $\wedge$ *astate*(*aa*) = *KO*
THEN
agents := *agents* - {*aa*};
scopes := *scopes* $\rhd$ {*aa*};
scopes := *sowner* $^{-1}$ [{*aa*}] $\lhd$ *scopes*;
sowner := *sowner* $\rhd$ {*aa*};
astate := *aa* $\lhd$ *astate*;
recovery_limit := *aa* $\lhd$ *recovery_limit*
END;

AgentRecovery = **ANY** *aa* **WHERE**
aa $\in$ *agents* $\wedge$ *activeAgent*(*aa*) $\wedge$
astate(*aa*) = *RE* $\wedge$ *recovery_limit*(*aa*) $>$ 0
THEN
recovery_limit(*aa*) := *recovery_limit*(*aa*) - 1 ||
ANY *vv* **WHERE** *vv* $\in$ {*OK*, *RE*}
THEN *astate*(*aa*) := *vv* **END**
END;

AgentRecoveryStart = **ANY** *aa* **WHERE**
aa $\in$ *agents* $\wedge$ *activeAgent*(*aa*) $\wedge$
astate(*aa*) = *OK* $\wedge$ *recovery_limit*(*aa*) $>$ 0
THEN
recovery_limit(*aa*) := *recovery_limit*(*aa*) - 1 ||
astate(*aa*) := *RE*
END;

AgentRecoveryFailure = **ANY** *aa* **WHERE**
aa $\in$ *agents* $\wedge$ *activeAgent*(*aa*) $\wedge$ *astate*(*aa*) = *RE*
THEN
astate(*aa*) := *KO*
END
END

As before, in this refinement step we define the system variant to ensure that the newly introduced events converge, i.e., do not take the control forever. To guarantee this, we introduce the variable $recovery\_limit$, which limits the amount of error recovery attempts for each agent. Each attempt of error recovery decrements $recovery\_limit$. As soon as for some agent $recovery\_limit$ becomes zero, error recovery of this agent terminates and the error is treated as irrecoverable. We define the variant as the sum of $recovery\_limit$ of agents.

While specifying the error recovery procedure, it is crucial to ensure that error recovery terminates, i.e., does not continue forever. In this refinement step the variant also serves as the means to express this essential property of the system.

In the subsequent refinement steps we can introduce representation of agent roles as well as define conditions of agent compatibility while creating and joining the scopes. Due to the lack of space we omit the description of further development here.

The final specification contains a sufficiently detailed model of middleware for CAMA systems. It has been used as a basis for producing the actual C code of CAMA middleware, which can be found at [10]. The translation from Event B to C was straightforward. In the implementation, the global state of the middleware is distributed between agents and a location. Namely, a location can access the state of an agent and an agent can access the names of the available scopes hosted by a location.

The presented formal development has been completely verified with the automatic support of AtelierB [4]. The use of AtelierB has significantly eased verification of the refinement process, since the tool generated all required proofs and discharged most of them automatically. Approximately 250 non-trivial proofs were generated and about 80 % of them were proved automatically by the tool. The remaining proof obligations have been discharged using the interactive prover provided by AtelierB. We observed that the most difficult to prove were the properties relating the scope status with the status of collaborating agents. Also the later refinement steps required significant efforts for proving that newly introduced events converge, i.e., do not introduce additional deadlocks.

Stepwise development process allowed us to introduce the implementation details gradually, which permitted exhaustive verification of the complex final specification by proof. This case study demonstrates that the refinement process is suitable for development of complex agent systems.

V. CONCLUSIONS

In this paper we presented a rigorous approach to developing fault tolerant middleware for agent systems. The top-down development paradigm adopted by Event B allowed us to carry out the system development from a highly abstract specification till the specification directly translatable into C code. Such an approach avoids a typical problem of many formal techniques, which are unable to bridge the gap between an abstract system model and its implementation. The developed formal specification has significantly simplified the implementation process, which is usually cumbersome and error-prone due to a distributed nature of mobile agent systems.

Ensuring a high degree of dependability of mobile agent systems is unfeasible without addressing one of the fundamental issues of distributed systems, the possibility of partial

failure. Therefore, integration of the means for fault tolerance should be an intrinsic part of the development of such systems. In our approach we proposed formal patterns for specifying the fault tolerance mechanisms for mobile location-based agent systems. We demonstrated how to formally define the mechanisms for tolerating agent disconnections, typical for mobile systems, as well as agent crashes. We believe that formally verified middleware has a potential to enhance dependability of systems dynamically composed of independently developed mobile agents.

A formalisation of mobile agent systems has been proposed by Roman et al. [13]. In their approach agent systems are specified and verified within the UNITY framework. The latest extension, called ContextUNITY [12], also captures the essential characteristics of context-awareness in mobile agent systems. The approach proposed by Roman et al. is especially suitable for treating context-awareness. However, it leaves a gap between a formal specification and implementation. The use of refinement in our approach allows us to overcome this limitation.

Fisher and Ghidini have presented a formal logic for describing agent activities [6]. They proposed either to deduce agent correctness from an agent specification via a number of transformations or verify it using a model checker. In their work, communication is modelled very abstractly by representing a list of external messages for each agent. This restricts reasoning about agent inter-operability, which is supported in our approach.

There is a lot of work on using model checking for verifying agent systems. However, model checking approaches typically suffer from the state space explosion problem, which is especially acute for large systems, such as mobile agent systems. The major advantage of our approach that it avoids this problem.

It is widely recognised that application of formal techniques is desirable to obtain assurance in correctness of mobile agent systems. However, complexity of such systems makes construction of a formal model, adequately representing the whole variety of middleware design principles and patterns, very challenging. In our approach, the use of stepwise refinement as a main development technique has alleviated this problem. Indeed, by starting the development from a high level of abstraction and progressively introducing representation of implementation details, we gradually increased complexity of a system model. Since each refinement step was formally verified, the final system model was proved to be correct by construction. Therefore, though the obtained model is complex because it realistically represents middleware for mobile agent systems, it is, nevertheless, exhaustively verified with the automatic support provided by Atelier B.

In our future work we are planning to extend the proposed approach in two directions. On the one hand, it would be interesting to investigate the use of decomposition to derive role-structured agent software from the overall system specification. On the other hand, it would be also useful to explore the formal specification of cooperative recovery as a basic mechanism for fault tolerance in agent systems.

ACKNOWLEDGEMENT

The work presented here is supported the EU research project ICT 214158 DEPLOY (Industrial deployment of system engineering methods providing high dependability and productivity) – www.deploy-project.eu.

REFERENCES

[1] J.-R.Abrial. *The B-Book*. Cambridge University Press, 1996.
[2] J.-R.Abrial and L.Mussat. Introducing Dynamic Constraints in B. In *Proc. of Second International B Conference*, LNCS 1393, Springer-Verlag, 1998.
[3] B.Arief, A.Iliasov and A.Romanovsky. On Using the CAMA Framework for Developing Open Mobile Fault Tolerant Agent Systems. In *Proc. of the International Workshop on Software Engineering for Large-scale Multi-agent Systems (SELMAS'06)*. International Conference of Software Engineering (ICSE'06), Shanghai, China. ACM Press, 2006.
[4] Clearsy. *AtelierB: User and Reference Manuals*. Available at http://www.atelierb.societe.com/index_uk.html.
[5] E.W.Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.
[6] M.Fisher and C.Ghidini. The ABC of Rational Agent Modelling. In *Proc. of the First International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS02)*. Bologna, Italy, July 2002. ACM Press, 2002.
[7] D.Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1): 80-112, 1985.
[8] A.Iliasov and A.Romanovsky. Exception Handling in Coordination-based Mobile Environments. In *Proc. of the 29th Annual International Computer Software and Applications Conference (COMPSAC 2005)*, pp.341-350, IEEE Computer Society Press, 2005.
[9] A. Iliasov and A. Romanovsky. CAMA: Structured Coordination Space and Exception Propagation Mechanism for Mobile Agents. *Proceedings of ECOOP-EHWS 2005*, Glasgow, UK, July 2005.
[10] A.Iliasov. Implementation of CAMA Middleware. Available online at http://sourceforge.net/projects/cama.
[11] OMG MASIF. Available at www.omg.org
[12] G.-C.Roman, C.Julien and J.Payton. A Formal Treatment of Context-Awareness. In *Proc. of FASE'2004*, pp.12–36, LNCS 2984, Springer-Verlag, 2004.
[13] G.-C.Roman, P.McCann and J.Plun. Mobile UNITY: Reasoning and Specification in Mobile Computing. *ACM Transactions of Software Engineering and Methodology*, Vol.6(3), pp.250–282, 1997.
[14] The full B specifications of middleware development. Available at http://www.abo.fi/∼Linas.Laibinis/MiddlewareSpecs.zip