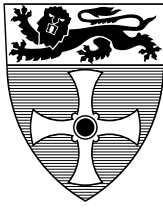


UNIVERSITY OF
NEWCASTLE



University of Newcastle upon Tyne

COMPUTING SCIENCE

Formal Approach to Ensuring Interoperability of Mobile Agents

L. Laibinis, A. Iliasov, E. Troubitsyna, A. Romanovsky.

TECHNICAL REPORT SERIES

No. CS-TR-989

November, 2006

Formal Approach to Ensuring Interoperability of Mobile Agents

L. Laibinis, A. Iliasov, E. Troubitsyna, A. Romanovsky.

Abstract

Mobile agent systems are complex distributed systems that are dynamically composed from autonomous agents. It is difficult to guarantee agent interoperability since they are often developed independently. We propose a formal approach to ensuring agent interoperability which relies on decomposition of a specification of the multiagent application into a set of specifications of the agent roles. We use refinement in the Event-B framework to formally define the process of decomposition.

Bibliographical details

LAIBINIS, L., ILIASOV, A., TROUBITSYNA, E., ROMANOVSKY, A..

Formal Approach to Ensuring Interoperability of Mobile Agents
[By] L. Laibinis, A. Iliasov, E. Troubitsyna, A. Romanovsky.

Newcastle upon Tyne: University of Newcastle upon Tyne: Computing Science, 2006.

(University of Newcastle upon Tyne, Computing Science, Technical Report Series, No. CS-TR-989)

Added entries

UNIVERSITY OF NEWCASTLE UPON TYNE
Computing Science. Technical Report Series. CS-TR-989

Abstract

Mobile agent systems are complex distributed systems that are dynamically composed from autonomous agents. It is difficult to guarantee agent interoperability since they are often developed independently. We propose a formal approach to ensuring agent interoperability which relies on decomposition of a specification of the multiagent application into a set of specifications of the agent roles. We use refinement in the Event-B framework to formally define the process of decomposition.

About the author

Alexei Iliasov is a research associate within the School of Computing Science at Newcastle University.

Alexander (Sascha) Romanovsky is a Professor in the CSR. He received a M.Sc. degree in Applied Mathematics from Moscow State University and a PhD degree in Computer Science from St. Petersburg State Technical University. He was with this University from 1984 until 1996, doing research and teaching. In 1991 he worked as a visiting researcher at ABB Ltd Computer Architecture Lab Research Center, Switzerland. In 1993 he was a visiting fellow at Istituto di Elaborazione della Informazione, CNR, Pisa, Italy. In 1993-94 he was a post-doctoral fellow with the Department of Computing Science, the University of Newcastle upon Tyne. In 1992-1998 he was involved in the Predictably Dependable Computing Systems (PDCS) ESPRIT Basic Research Action and the Design for Validation (DeVa) ESPRIT Basic Project. In 1998-2000 he worked on the Diversity in Safety Critical Software (DISCS) EPSRC/UK Project. Prof Romanovsky was a co-author of the Diversity with Off-The-Shelf Components (DOTS) EPSRC/UK Project and was involved in this project in 2001-2004. In 2000-2003 he was in the executive board of Dependable Systems of Systems (DSoS) IST Project. Now he is coordinating Rigorous Open Development Environment for Complex Systems (RODIN) IST Project (2004-2007).

Suggested keywords

FORMAL METHODS,
MOBILE COMPONENTS,
SCOPES,
FAULT TOLERANCE,
REFINEMENT

Formal Approach to Ensuring Interoperability of Mobile Agents

Linus Laibinis¹, Alexei Iliasov², Elena Troubitsyna¹, Alexander Romanovsky²

¹Åbo Akademi University, Finland {Linus.Laibinis, Elena.Troubitsyna}@abo.fi

²University of Newcastle upon Tyne, UK
{Alexei.Iliasov,Alexander.Romanovsky}@ncl.ac.uk

Abstract. Mobile agent systems are complex distributed systems that are dynamically composed of autonomous agents. Since agents are often developed independently of each other, they may lack inter-operability, i.e., do not communicate in a correct manner. In this paper we propose a formal approach to ensuring inter-operability of agents implemented by independent developers. The essence of our approach is decomposition of a specification of overall multi-agent application into a set of specifications of agent roles. While decomposing the specification, we explicitly define communication between the agents. To ensure inter-operability, the implementation of each agent should adhere to its formal specification. However, each agent can be implemented completely independently, i.e., without knowing the specifications of other agents. We use refinement in the Event-B framework to formally define the process of decomposing a multi-agent application into a set of interacting roles. Our approach is illustrated by a case study – development of an electronic auction.

1 Introduction

Mobile agent systems has emerged as a result of the integration of computing and communication. Such systems are examples of complex distributed systems that are dynamically composed of independent agents. The complexity associated with these systems makes designing mobile agent software notoriously error prone. Since agents are usually developed independently of each other, one of the major problems is a lack of agent interoperability, i.e., mismatch in agent communication. In this paper we propose a formal approach to designing mobile agent systems which allows to ensure interoperability.

In our approach, agent development starts with a formal specification of a scope – the logical representation of the behaviour of a multi-agent application. Each agent plays a certain role in a scope. We formally specify the behaviour in the scope and decompose it into a set of role specification by refinement. We use the Event-B framework [3, 2] – an extension of the B Method to reason about reactive and distributed systems – as our formal basis. The communication between agent is introduce in the process of decomposition. As a result

of decomposition all the information about the role functionality and the communication mechanisms is contained within a role specification. Hence, if each agent implementation adheres to the role specification then the implementations of the agents are compatible with each other, i.e., inter-operability is achieved.

Our approach enables truly decentralised development of agent systems. Developers of individual agents do not have to communicate with each other or disclose agent source code in order to produce agents which are able to collaborate. We see this as the main prerequisite for building and deploying agent systems where new agents with different but compatible functionality can appear at any moment and engage into collaboration.

With the lack of certainty about behaviour of other agents developers often have to rely on a defensive strategy of extensive use of checks on incoming messages. With our approach most of such checks and assertions can be eliminated. This results in simpler and more efficient agents which are easier to implement and understand. Moreover, our approach supports reuse, since a specification of a role should be developed once but can be reused many times. Furthermore, the formal specifications are language neutral and in many aspects implementation-neutral. Any specification is equally well suited to any combination of a platform, middleware and a programming language.

We proceed as follows: in Section 2 we introduce CAMA – Context-Aware Mobile Agents middleware – for which our approach is adapted. In Section 3 we present Event-B – our formal development method. In Section 4 we present our main contribution – the methodology for formal specification and decomposition of multi-agent applications. In Section 5 we demonstrate by a case study – an electronic auction – an application of our approach in decentralized development of agents from formal role specifications. We discuss the proposed approach, overview the related and future work in Section 6.

2 CAMA Middleware

Context-Aware Mobile Agents systems – CAMA systems [10, 5, 4] – are defined via a set of abstractions and operations on them modelling inter-agent communication and operability. The primary goal of defining a CAMA system is to offer programmers a formally-verified basis for rapid development of mobile agent software in a disciplined and structured way.

CAMA inter-agent communication is based on the LINDA [9] paradigm. LINDA provides a set of coordination primitives that can be used for coordination of several independently running pieces of software. Since LINDA is language independent, it became quite popular and its coordination primitives have been implemented in many programming languages. Moreover, LINDA supports asynchronous and anonymous agent communication and hence is well-suited for mobile agent systems.

A CAMA system consists of a set of *locations*. The main role of a location is to provide the inter-agent communication service to its client agents. The communication service is based on a shared blackboard supporting LINDA operations.

One of the major contributions of CAMA is a novel mechanism to structure a shared blackboard so that groups of communicating agents can work in

isolated coordination spaces, called *scopes* [10]. In addition to isolation of the coordination space, the scoping mechanism also provides a dynamic type-checking facility insuring agent inter-operability for multi-agent applications. As a result, the scoping mechanism only permits collaboration of the agents with compatible functionality, which is defined by their attributes.

The main structuring units of CAMA applications are *agents* which are pieces of software conforming to some formal *specification*. To distinguish between various functionalities of individual agents, and to match compatible agents, CAMA uses agent *roles* as units of functionality structuring. The role-based type-checking allows dynamic composition of multi-agent applications that ensure agent inter-operability and isolation.

Let us now describe the CAMA abstractions which we will also later use in our formal modelling of agent systems.

2.1 CAMA Abstractions

Scope is an abstraction designating an isolated coordination space for compatible agents. A scope creation is initiated by an agent. A scope is defined by a set of roles and restrictions on roles. The restrictions on roles dictate how many agents can play any given role in a scope. A location tracks the number of currently taken roles in all created scopes.

Role is an abstract description of an agent functionality. Each scope supports a predefined number of different roles. An agent may implement a number of roles and can also take several roles within different scopes. In this paper we assume that an agent can play at most one role in each scope.

Location is an abstraction defining inter-agent communication. It is the core part of the system because it provides the means of communication and coordination between agents. We assume that each location can be uniquely identified. A location keeps track of the connected agents and their properties in order to update scope states and ensure isolation. A location by itself might also provides some additional services to agents. In addition to supporting scopes as means of agent communication, locations may also offer a support for logical mobility of agents, hosting of agent and agent backup.

Agent is a piece of software implementing a set of roles which allows it to participate in certain scopes. All agents must implement some minimal functionality which allows them to engage in or disengage from a location.

Next we present a brief overview of the essential operations defined over the CAMA abstractions.

2.2 CAMA Operations

The CAMA operations can be grouped together into the following three categories: location engagement, scoping mechanism, and communication. The communication operations implement the standard LINDA coordination paradigm. The location engagement operations associate or disassociate an agent with a location. Finally, the scoping mechanism operations allow an agent to enquiry

for available scopes, create new scopes, destroy previously created scopes, join and leave existing scopes.

In a typical scenario, 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. At some moment agents leaves the scope and joins another scope or disconnects from the location.

Agents interact by sending messages to each other. The message order must be the same for the message consumer and the message producer. In our approach all the requirements to message ordering are embedded into a role specification.

3 Formal Modelling and Refinement in EventB

In this section we present the background for our formal development of mobile agent systems. Namely, we explain the principles of formal modelling and refinement in the EventB framework [3, 2, 1].

3.1 Modelling in EventB

A formal specification is a mathematical model of the required behaviour of a (part of) system. In B, a specification is represented by a collection of modules, called Abstract Machines. The common pseudo-programming notation, called Abstract Machine Notation (AMN), is used in constructing and formally verifying them. An abstract machine encapsulates a local state (local variables) of the machine and provides operations on the state. The operations form externally visible interface of the machine. A simple abstract machine has the following general form:

```

SYSTEM AM
SETS TYPES
VARIABLES v
INVARIANT I
INITIALISATION INIT
OPERATIONS
  E1 = ...
  ...
  EN = ...
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.

The operations E_1, \dots, E_N of the machine are defined in the **OPERATIONS** clause. The operations are atomic meaning that, once an operation is chosen, its execution will run until completion without interference.

In this paper we take the *event-based approach* to specifying mobile agent systems. The operations of event-based systems are called *events* and defined in the **EVENTS** clause which replaces **OPERATIONS** clause. An event is defined as follows:

$$E = \mathbf{WHEN} \ g \ \mathbf{THEN} \ S \ \mathbf{END}$$

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

Several events can be grouped together in *array of events*. It has the following syntax:

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

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., the event is *enabled*. If no events 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:

$$S \quad ::= \quad x := e \mid \mathbf{IF} \ cond \ \mathbf{THEN} \ S1 \ \mathbf{ELSE} \ S2 \ \mathbf{END} \mid S1 ; S2 \mid \\ \mathbf{LET} \ x, y, \dots \ \mathbf{BE} \ x = expr1 \wedge y = expr2 \dots \ \mathbf{IN} \ S \ \mathbf{END} \mid S1 \parallel S2 \mid \dots$$

The first three constructs - an assignment, a conditional statement and a sequential composition have the standard meaning. Sequential composition is disallowed in abstract specifications but permitted in refinements. The **LET** construct allows us to declare new local variables, initialise them according to the given expressions and then use them 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 [8]. Intuitively, for a given statement S and a postcondition P , the weakest precondition $wp(S, P)$ describes the set of all such initial states from which execution of S is guaranteed to establish P . The weakest precondition semantics is a foundation for establishing correctness of specifications and verifying refinements between them. To show correctness (consistency) of an event-based system, we should demonstrate that its invariant is *true* in the initial state (i.e., after the initialisation is executed) and that every event preserves the invariant:

$$wp(INIT, I) = true, \quad \text{and} \\ g_i \wedge I \Rightarrow wp(E_i, I)$$

3.2 Refinement of Event-based Systems

The basic idea underlying formal stepwise development 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 an executable

code. The intermediate stages yield the specifications containing a mixture of abstract mathematical constructs and executable programming artefacts. In general, refinement process can be seen as a way to reduce nondeterminism of the abstract specification, to replace abstract mathematical data structures by data structures implementable on a computer, and, hence, gradually introduce implementation decisions.

In the AMN the results of intermediate development stages - the refinement machines - have essentially the same structure as the more abstract specifications. In addition, the refinement machine explicitly states which specification it refines. For instance, assume that the refinement machine AM' is a result of refinement of the abstract machine AM :

```

REFINEMENT  $AM'$ 
REFINES  $AM$ 
VARIABLES  $v'$ 
INVARIANT  $I'$ 
INITIALISATION  $INIT'$ 
OPERATIONS
   $E_1 = \dots$ 
   $\dots$ 
   $E_N = \dots$ 
END

```

In AM' we replace the abstract data structures of the abstract machine AM with the concrete ones. The invariant of $AM' - I'$ - defines now not only the invariant properties of the refined specification, but also the connection between the newly introduced variables and the abstract variables that they replace. For a refinement step to be valid, every possible execution of the refined machine must correspond (via I') to some execution of the abstract machine. To demonstrate this, we should prove that $INIT'$ is a valid refinement of $INIT$, each event of AM' is a valid refinement of its counterpart in AM and that the refined specification does not introduce additional deadlocks, i.e.,

$$\begin{aligned}
 wp(INIT', \neg wp(INIT, \neg I')) &= true, \\
 I \wedge I' \wedge g'_i &\Rightarrow g_i \wedge wp(S', \neg wp(S, \neg InvC)), \quad \text{and} \\
 I \wedge \bigvee_i^N g_i &\Rightarrow g'_i
 \end{aligned}$$

At a certain stage of formal development we often need to decompose a specification into a set of specifications modelling separate parts of the system under construction. Decomposition allows us to decentralize the development and reduce its complexity, since the subsystem specifications can be developed further independently.

In Event-B decomposition is defined as a special refinement step [1]. Assume that we need to decompose the (specification of) system M into the subsystems N and P . The system M has the variables $v1, v2, v3$ and the operations $e1, e2, e3, e4$. Suppose also we can partition the variables into the variables to be used as internal variables of N (say, $v1$), the variables to be used as internal variables of P ($v3$), and the variables shared between N and P ($v2$).

Assume that the events $e1, e2$ update only the internal and shared variables of N ($v1, v2$). Therefore, these events can be put into the specification of N . Similarly, the events $e3, e4$ update only the internal and shared variables of P ($v3, v2$) and go into the specification of P . To make the specifications of N and P independent of each other, we also have to abstractly specify how the shared variables $v2$ can be changed by their counterparts (i.e., the environment). In other words, we have to add additional events (called *external events*) modelling possible changes of the shared variables by the environment. Such events should be abstract enough to show that the actual event operations changing these variables in the environment are refinements of these external events.

4 Scope Specification and Decomposition into Roles

In this section we present a specification pattern that we propose for describing scope functionality. Moreover, we show how we can use formal refinement to introduce communication mechanisms into a specification of a multi-agent application. In our final refinement step the scope specification is decomposed into separate role specifications that can be then used to implement compatible cooperative agents.

4.1 Structure of Scope Specification

A scope specification defines the scope state (program variables) and the scope events (operations) that update the state. At some stage of specification refinement we eliminate abstract scope variables and introduce roles. The scope state is partitioned by distributing the program variables among the scope roles so that for each variable there is exactly one role responsible for updating it. Similarly, for operations, we specify the scope events in such a way that each event updates variables of only one role. As a result, we attribute each scope operation with a single role. At the same time, a scope operation can read variables of other roles. This gives us an abstract way for modelling coordination among the scope roles in the initial specification.

We propose the following specification pattern for a scope:

<p>SYSTEM<i>Scope</i></p> <p>SETS R_1, R_2, \dots, R_n</p> <p>VARIABLES</p> <p>$var_1^1, var_2^1, \dots, var_{k_1}^1,$ $var_1^2, var_2^2, \dots, var_{k_2}^2,$ \dots $var_1^n, var_2^n, \dots, var_{k_n}^n$</p> <p>INVARIANT</p> <p>$var_1^1 \in R_1 \leftrightarrow Type_1^1 \wedge$ $var_2^1 \in R_1 \leftrightarrow Type_2^1 \wedge$ \dots $var_1^2 \in R_2 \leftrightarrow Type_1^2 \wedge$ \dots $var_1^n \in R_n \leftrightarrow Type_1^n \wedge$ \dots /* other invariant properties */</p>	<p>EVENTS</p> <p>/* 1st role */</p> <p>$role^1_local_reaction_1 = \dots$ $role^1_local_reaction_2 = \dots$ \dots $role^1_local_reaction_{o_1} = \dots$ \dots $role^1_external_reaction_1 = \dots$ $role^1_external_reaction_2 = \dots$ \dots $role^1_external_reaction_{r_1} = \dots$ /* nth role */</p> <p>$role^n_local_reaction_1 = \dots$ \dots $role^n_external_reaction_1 = \dots$ \dots END</p>
--	--

In the **SETS** clause we introduce abstract sets, one for each scope role. These sets abstractly represent all agents that can possibly join a scope in these roles. For example, in a scope which allows buying or selling items among agents, the set **SELLER** would represent all agents that could possibly participate in the scope and play the seller role.

The **VARIABLES** clause introduces the scope state as a list of program variables, which are distributed among the involved roles. Each role variable is defined as a partial function from the corresponding role set to some concrete type. The function is partial because it is defined only for currently active agents playing this particular role in a scope. For example, $in_stock \in \text{SELLER} \rightarrow \mathcal{P}(\text{ITEM})$ represents information about “in-stock” items of all active sellers in a scope. In a special case, a scope may require strictly single instance of a particular role. For such a role, the role variables are simply defined as elements of some concrete type.

In the scope operations, such role variables are always used to access or update the values of some particular role instance (agent), i.e., they used in the form $var_{index}^{role\ instance}$. From now on, we use the term “role variables” to refer to the variables of a role instance.

Let us now discuss the role operations defined in the **EVENTS** section of a specification. In general, we assume that there are two important classes of scope events: the events reacting on changes in the role state (local reactions) and the events reacting on requests generated by other roles (external reactions).

The local role reactions can be of the following two forms:

<pre> <i>role_local_reaction</i> = WHEN <i>condition(role variables)</i> THEN /* update role variables */ END </pre>	<pre> <i>role_local_reaction</i> = ANY v_1, \dots, v_n WHEN <i>condition(role variables)</i> THEN /* update role variables */ END </pre>
---	---

Note that the role variables here are variables of a particular role instance. An event is a local reaction if a role relies exclusively on its local state when computing new values of roles variables.

A scope describes coordinated actions of several roles. Coordinated here means that often the role reactions must be executed in a specific order, i.e., they should follow a certain predefined scenario. To model coordination between different role instances, we need to describe role reactions to specific changes that occurred in other roles.

The role external reactions are of the following form:

```

role_external_reaction =
  ANY  $v_1, \dots, v_n$ 
  WHERE
    /*  $v_1$  is external value of some type */  $\wedge$ 
    ...
    /*  $v_n$  is external value of some type */  $\wedge$ 
    Condition( $v_1, \dots, v_n, \text{external role variables}$ )
  THEN
    /* update local role variables using  $v_1, \dots, v_n$  */
  END

```

An external reaction describes how a role instance reacts on an external event. An external event is specified as a condition on the external role variables. Therefore, an external reaction needs the read access to the variables of other scope roles. We model this by introducing the external values v_1, \dots, v_N , which represent locally available information about the state of other roles.

In the next section we show how to introduce communication among role instances. Once the communication mechanism is in place, the external values v_1, \dots, v_N will be replaced by the corresponding parameters of requests sent by collaborating agents.

Let us consider a small example of a role external reaction. We can describe the following reaction of a buyer when it gets an offer from a seller for some item. As a result of the operation, a buyer updates its local role variable *cheapest_offer*.

```

reaction_of_buyer_to_seller =
  ANY seller, item, price
  WHERE
    seller ∈ SELLER ∧
    item ∈ ITEM ∧
    price ∈ NAT ∧
    item ∈ in_stock(seller) ∧
    price = item_price(seller, item)
  THEN
    IF price < cheapest_offer(buyer, item)
    THEN
      cheapest_offer(buyer, item) := price
    END
  END

```

where *in_stock* and *item_price* are the variables of the seller role.

4.2 Introducing Communication

In this section we show how we can refine our scope specification by introducing communication between role instances. The role communication is expressed in terms of sending and receiving requests.

This refinement step helps us in achieving our main goal – decomposition of a scope specification into a number of independent specifications, one for each role of the scope. In our scope specification, roles are still tied together because the role external reactions can refer to the variables of other roles. To make scope decomposition possible, we have to decouple role operations, i.e., remove any references to external role variables in all the external reactions of each role.

Our refined specification is presented on the next page. We assume that each role has its own buffer for incoming requests. For the sake of simplicity, we model sending requests by allowing roles to put their requests into the corresponding buffers of other roles. We also assume that implicitly present middleware is responsible for transporting these requests across scope roles.¹

¹ The communication model we described is very simple. However, it can be easily extended and made more realistic by introducing separate buffers for both incoming and outgoing requests. In the extended model the middleware would be explicitly modelled as an additional entity which constantly monitors the buffer states and transfers requests.

<p>REFINEMENT<i>Scope_ref</i></p> <p>REFINES<i>Scope</i></p> <p>SETS</p> <p><i>Request</i>; <i>Request_Type</i> = $\{RequestType_1, \dots, RequestType_m\}$</p> <p>CONSTANTS</p> <p><i>ReqType</i>, <i>Request</i>₁, ..., <i>Request</i>_{<i>M</i>}, <i>Req_param</i>₁, ..., <i>Req_param</i>_{<i>k</i>}</p> <p>PROPERTIES</p> <p>$ReqType \in Request \rightarrow Request_Type \wedge$ $Request_1 \in Type_1^1 * .. Type_k^1 \rightarrow Request \wedge$... $Request_m \in Type_1^m * .. Type_k^m \rightarrow Request \wedge$ $Req_param_1 \in Request \leftrightarrow Type_1 \wedge$... $Req_param_{s_k} \in Request \leftrightarrow Type_k$</p> <p>VARIABLES</p> <p>..., <i>role</i>₁-<i>buffer</i>, <i>role</i>₂-<i>buffer</i>, ... , <i>role</i>_{<i>n</i>}-<i>buffer</i></p>	<p>INVARIANT</p> <p>...</p> <p>$role_1_buffer \in seq(Request) \wedge$ $role_2_buffer \in seq(Request) \wedge$... $role_n_buffer \in seq(Request) \wedge$ $\forall rr \cdot rr \in ran(role_1_buffer) \wedge$ $ReqType(rr) = RequestType_a \Rightarrow$ $Condition_1(\dots, role_1_variables) \wedge$... $\forall rr \cdot rr \in ran(role_n_buffer) \wedge$ $ReqType(rr) = RequestType_b \Rightarrow$ $Condition_n(\dots, role_n_variables) \wedge$ /* other invariant properties */</p> <p>EVENTS</p> <p>/* 1st role */ /* local role operations */ ... /* role reactions */ ... /* n - th role */ /* local role operations */ ... /* role reactions */</p> <p>END</p>
---	--

Different request types are used to indicate different external events. We introduce the abstract set *Request* to model all possible requests that can be generated in the current scope. In addition, the function *ReqType* is used to classify a request into one of the predefined request types: $\{RequestType_1, \dots, RequestType_m\}$. The request type also defines the number and types of its parameters. We introduce the request constructor functions *Request*₁, ..., *Request*_{*m*}, which construct requests of a certain type. Similarly, we define the projection (destructor) functions *Req_param*₁, ..., *Req_param*_{*k*}, which extract the request parameters from a request depending on its type.

The buffers of incoming requests *role*₁-*buffer*, ..., *role*_{*n*}-*buffer* are modelled as B sequences². The requests are handled in the order they arrive by reading the first request from a role buffer.

Both local and external reactions can send requests to other roles instances (agents). A request generation corresponds to the following assignment (appending the corresponding role buffer)

$$role_i_buffer := role_i_buffer <-< Request_j(param_{-j_1}, \dots, param_{-j_n})$$

in a role reaction.

² A sequence in B is the special case of a function. It maps natural numbers from the internal $0 \dots n$ into values of some base type.

Only external reactions can read and handle incoming requests. In this refinement step we transform (refine) all role external reactions described in the previous section into reactions of the following form:

```

rolek_external_reaction =
  WHEN not(rolek_buffer = <>) ∧
    ReqType(first(rolek_buffer)) = Requestitypei
  THEN
    LET v1 = Reqparam1(first(rolek_buffer)) ∧ ...
      vn = Reqparamn(first(rolek_buffer))
    THEN
      /* update rolek variables using v1, ..., vn */
      rolek_buffer := tail(rolek_buffer)
    END
  END
    
```

A role external reaction describes now the actions of a role instance when a request of a particular type arrives. The operation extracts arguments from the request and uses them to initialise the values v_1, \dots, v_n , which are used for computations in the operation body. The number of arguments and the argument types are fixed for each external reaction.

Let us note that there is no condition tying the role reaction with a predicate on other role variables. By removing this condition, we make a role reaction decoupled from the reactions of the other roles. By repeating this for all external reactions of a role we can obtain a role specification completely decoupled from the other scope roles.

To show that such operation transformation is indeed a refinement, we have to prove that, whenever a request of this type is generated, the following condition holds:

$$\forall rr \cdot rr \in \text{ran}(\text{buffer}) \wedge \text{ReqType}(rr) = \text{Request}_{type_i} \Rightarrow$$

$$\text{Condition}_k(\text{Req}_{param_1}(rr), \dots, \text{Req}_{param_n}(rr), \text{variables of other roles})$$

This condition should be added to the gluing invariant of the refined specification. For example, the buyer external reaction to a seller offer presented in the previous section can be refined in the following way:

```

reaction_of_buyer_to_seller =
  WHEN not(buyer_buffer = <>) ∧
    ReqType(first(buyer_buffer)) = Selling_Offer
  THEN
    LET
      seller = Seller(first(buyer_buffer)) ∧
      item = Item(first(buyer_buffer)) ∧
      price = Price(first(buyer_buffer))
    THEN
      IF price < cheapest_offer(buyer, item)
        THEN
          cheapest_offer(buyer, item) := price
        END;
      buyer_buffer := tail(buyer_buffer)
    END
  END
    
```

Here $Seller, Item, Price$ are the projection functions extracting the corresponding parameters from $Selling_Offer$ request.

However, the buyer reaction relies on the condition that the offered item is in the seller stock and the offered price is the official seller price for this item. To show operation refinement, we have to guarantee that the following invariant property

$$\begin{aligned} \forall rr \cdot rr \in \text{ran}(\text{buyer_buffer}) \wedge \text{ReqType}(rr) = \text{Selling_Offer} \Rightarrow \\ \text{in_stock}(\text{Seller}(rr), \text{Item}(rr)) \wedge \\ \text{Price}(rr) = \text{item_price}(\text{Seller}(rr), \text{Item}(rr)) \end{aligned}$$

is true, whenever $Selling_Offer$ request is generated. Such a request, for example, can be a seller reaction to the $Item_enquiry$ request sent earlier by a buyer:

```

reaction_of_seller_to_buyer =
  WHEN not(seller_buffer = <>) ∧
    ReqType(first(seller_buffer)) = Item_enquiry
  THEN
    LET
      seller = Seller(first(seller_buffer)) ∧
      item = Item(first(buyer_buffer))
    THEN
      IF item ∈ in_stock(seller) THEN
        buyer_buffer := buyer_buffer ←
          Seller_OfferReq(seller, item, price(seller, item))
      END;
      seller_buffer := tail(seller_buffer)
    END
  END

```

We can formally demonstrate that a request generated in this way satisfies the coupling condition for $reaction_of_buyer_to_seller$.

4.3 Scope Decomposition into Roles

As a result of introducing communication between role instances, we achieved that the operations belonging to different roles became completely decoupled. The only way one role instance can affect another is via sending and receiving requests. This means that we can safely decompose the refined scope specification, separating the program variables and operations of different roles and putting them into different role specifications. To show that this is a valid refinement, we use novel ideas of B decomposition refinement introduced in [1].

The specification presented in the previous section can be decomposed into a number of role specifications, one per each role. For example, the resulting role specification for the role i is of the following form:

REFINEMENT $Role_i$	
VARIABLES	EVENTS
$var_1^i, var_2^i, \dots, var_k^i,$	$role_i_local_reaction_1 = \dots$
$role_1_buffer, role_2_buffer, \dots, role_n_buffer$	$role_i_local_reaction_2 = \dots$
INVARIANT	\dots
$var_1^i \in R_1 \mapsto Type_1^1 \wedge$	$role_i_local_reaction_{o_i} = \dots$
$var_2^i \in R_1 \mapsto Type_2^1 \wedge$	$role_i_external_reaction_1 = \dots$
\dots	\dots
$var_k^i \in R_1 \mapsto Type_k^1 \wedge$	$role_i_external_reaction_{r_i} = \dots$
$role_1_buffer \in seq(Request) \wedge$	$external_event_1 = \dots$
$role_2_buffer \in seq(Request) \wedge$	\dots
\dots	$external_event_i = \dots$
$role_n_buffer \in seq(Request) \wedge$	END
$/* other invariant properties */$	

The variables and operation of $role_1$ are just copied from the previous specification. The additional events $external_event_1, \dots, external_event_l$ are needed to abstractly model possible changes to the shared variables $role_1_buffer, \dots, role_n_buffer$ made by other roles. We know that other roles can either read the buffers $role_1_buffer, \dots, role_n_buffer$ (except $role_i_buffer$) to handle requests from $role_i$ or write their own requests to the role i into the buffer $role_i_buffer$. Therefore, these operations can be either of the form

WHEN $not(role_i_buffer = \langle \rangle)$ **THEN** $role_i_buffer := tail(role_i_buffer)$ **END**

or of the form

ANY rr

WHERE $rr \in Request \wedge ReqType(rr) : \{RequestType_1^i, \dots, RequestType_n^i\}$

THEN $role_i_buffer := role_i_buffer \leftarrow rr$ **END**

where $\{RequestType_1^i, \dots, RequestType_n^i\}$ is the set of request types that the role i expects from other roles.

5 Case Study: Auction

In this section we illustrate the proposed approach by a simple case study of an auction. We start with a scope specification for an auction, then introduce simple communication mechanism for sending and receiving different types of requests, and, finally, decompose the specification into specifications of the involved roles.

5.1 Scope Specification

The auction specification describes activities of agents of three different roles: a seller, a buyer, and a manager. There could any number of sellers and buyers participating in a scope. However, there should be exactly one agent playing the role of a manager, which keeps the bulk of information about the current auction state and controls validity of auction operations.

The auction specification describes the following allowed scenario for a seller, a buyer, and a manager. The scenario starts by a seller, which puts an item to be sold at the auction. Once the item is received by the manager, the bidding process for this particular item starts. Any active buyer can make its bid. However,

only higher bids (for a particular item) are accepted by the manager. After a predefined number of bids, the bidding process is stopped and the winner (i.e., a bidder with the highest bid) is decided by the manager. Once the payment from the winner is received, the item is declared officially sold, and the corresponding seller and buyer are notified about this.

Graphically, we can present the scenario as the following chain of operations (events) involving a seller, a buyer, and the manager:

$$\begin{aligned}
& Put_new_item(Seller) \longrightarrow Get_new_item(Manager) \longrightarrow Make_bid(Buyer) \\
& \longrightarrow Take_bid(Manager) \cdots \longrightarrow Make_bid(Manager) \longrightarrow Take_bid(Manager) \\
& \longrightarrow Determine_winner(Manager) \longrightarrow Make_payment(Buyer) \\
& \longrightarrow Receive_payment(Manager) \longrightarrow Item_sold(Manager) \\
& \longrightarrow Buying_confirmed(Buyer) \longrightarrow Selling_confirmed(Seller)
\end{aligned}$$

We specify all these steps as separate operations in our B specification of the auction scope. Some of these operations are local reactions for the corresponding roles, while the others are the role external reactions to some external events. Moreover, we distribute the program variables among the involved roles as well. Most variables are the role variables for the manager, except for `solditems` and `boughtitems` that belong to a seller and a buyer correspondingly.

SYSTEM *Auction*

SETS *BUYER; SELLER; ITEM*

VARIABLES

bids, bids_left, winners, payments, item_seller, buyer_log,
seller_log, solditems, boughtitems

INVARIANT...

INITIALISATION...

EVENTS

Put_new_item = .../ * Seller, local */
Get_new_item = .../ * Manager, external */
Make_bid = .. / * Buyer, local */
Take_bid = / * Manager, external */
ANY *bb, ii, pp* **WHERE**
bb : *BUYER* \wedge *ii* : *ITEM* \wedge *pp* : *NAT*
THEN
IF *ii* : *dom(bids_left)* \wedge
bids_left(ii) > 0 \wedge
pp > *max(ran(bids(ii)))* \wedge
not(ii : dom(winners))
THEN
bids(ii) := *bids(ii)* <+ {*bb* \mapsto *pp*} ||
bids_left(ii) := *bids_left(ii)* - 1
END
END;

Determine_winner = / * Manager, local */
ANY *ii* **WHERE**
ii : *dom(bids)* \wedge *bids_left(ii)* = 0 \wedge
not(ii : dom(winners))
THEN ... **END;**
Make_payment = / * Buyer, external */
ANY *bb, ii, pp* **WHERE**
bb : *BUYER* \wedge ...*bb* = *winners(ii)*
THEN ... **END;**
Receive_payment = .../ * Manager, external */
Item_sold = ... / * Manager, local */
Buying_confirmed = / * Buyer, external */
ANY *bb, ii* **WHERE**
bb : *BUYER* \wedge ...*bb* = *buyer_log(ii)*
THEN ... **END;**
Selling_confirmed = .../ * Seller, external */
END

We omitted most of the details about the scope operations and the invariant for the sake of simplicity. The full specification can be found in the accompanying technical report [12].

5.2 Introducing Communication

In the first refinement step we introduce simple communication mechanism for sending and receiving requests between the involved roles. The scope operations are transformed according to the methodology described in Section 4. We present the excerpt of the refined scope specification focusing on one role - a buyer. The operations of the other two roles are refined in a similar way.

A buyer has three operations – *Make_bid*, *Make_payment* and *Buying_confirmed*. *Make_bid* is a local reaction of a buyer. However, as a result of this operation, the buyer sends the corresponding request to the manager informing it about the buyer bid for a particular item.

Both *Make_payment* and *Buying_confirmed* are the buyer's reactions to the corresponding manager requests. In case of *Make_payment*, the buyer is declared the winner for a particular item and is asked to pay. As a result of the operation, a special paying request is sent to the manager. In case of *Buying_confirmed*, the buyer is informed that an item is now officially sold to it. Consequently, the buyer updates its local variable containing information about the bought items.

To make operations of one role decoupled from the other roles, we introduce different request types and then require (as an invariant property) that, whenever requests of a particular type are generated by the sending role, their parameters satisfy certain conditions expected by the receiving role. For example, for a buyer, it is important that it receives the payment requests from the manager only after the buyer was declared the winner for a particular item. This property is formulated as the first invariant property in the following specification.

```

REFINEMENT Auction_ref
REFINES Auction
SETS Request; RType = {Bidding, Payment, BuyConfirm, ... }

VARIABLES...man_buffer, buy_buffer, sel_buffer
INVARIANT
  (∀rr. (rr : ran(buy_buffer) ∧ ReqType(rr) = Payment =>
    Buyer(rr) = winners(Item(rr))) ∧
  (∀rr. (rr : ran(buy_buffer) ∧ ReqType(rr) = BuyConfirm =>
    Buyer(rr) : buyer_log(Item(rr))) ∧ ...
EVENTS
...
Make_bid =
  ANY bb, ii, pp WHERE bb : BUYER ∧ ii : ITEM ∧ pp : NAT
  THEN
    man_buffer := man_buffer ← BiddingReq(bb, ii, pp)
  END;
Make_payment =
  WHEN not(buy_buffer =<>) ∧ ReqType(first(buy_buffer)) = Payment
  THEN
    LET bb, ii, pp BE bb = Buyer(first(buy_buffer)) ∧ ii = Item(first(buy_buffer))
    ∧ pp = Price(first(buy_buffer))
  IN

```

```

    man_buffer := man_buffer ← PayingReq(bb, ii, pp);
    buy_buffer := tail(buy_buffer)
  END
END;
Buying_confirmed = / * Buyer * /
  WHEN not(buy_buffer =<>) ∧ ReqType(first(buy_buffer)) = BuyConfirm
  THEN ... END
END

```

5.3 Decomposing into Roles

In the final refinement step we decompose the specification into separate specifications of a buyer, a seller, and the manager. Below we present the specification of a buyer. The specifications for a seller and the manager are created in a similar way.

The specification *Buyer* contains the three buyer operations derived as a result of the previous refinement step. In addition, it has two events *ext_input* and *ext_output* specifying how other roles can affect the variables shared with a buyer – *buy_buffer* and *man_buffer*.

```

REFINEMENT Buyer
REFINES Auction_ref
VARIABLES man_buffer, buy_buffer, boughtitems
EVENTS
  Make_bid = ...
  Make_payment = ...
  Buying_confirmed = ...
  ext_input =
    ANY rr WHERE rr : Request ∧ ReqType(rr) : {Payment, BuyConfirm}
    THEN buy_buffer := buy_buffer ← rr END;
  ext_output =
    WHEN not(man_buffer =<>)
    THEN man_buffer := tail(man_buffer) END
END

```

6 Building Agents from Role Specifications

In the previous section we formally developed a set of B specifications for the auction case study. However, these specifications are not directly executable. In this section we present the final step of our development in which an executable code of a role is produced from its formal specification. Even though currently this is a manual process, we define strict translation rules that can be used for automated code generation.

The discussion is based on the specification of the Manager role from the auction case study. The code examples are given in Java. The CAMA adaptation layer for Java provides coordination and other utility functions. The presented translation steps, however, are language neutral.

A typical role specification has a number of guarded events and a set of variables updated by the events. The role invariant specifies the desirable properties of a system. In the implementation phase the invariant of a proved specification does not play any role since each event has been already proved to preserve it. This results in simpler implementations which are mostly free from assertions (used for expressing invariants) and the pieces of code for tolerating unexpected inputs and events.

To match the event-based style of a role specification, we use the reactive programming paradigm [6]. Reactive programs are composed entirely of reactions triggered by events which are executed asynchronously. In our approach a role implementation is composed of a number of subroutines (methods) implementing the bodies of the corresponding abstract events. We found it necessary to separate the event bodies and the event guards during our translation. This is dictated by the limitations of procedural languages which are mostly oriented to sequential execution while the reactive programming model is highly concurrent.

Fig.1 shows a Java class implementing the Manager role. Event bodies are contained in the methods which have the same names as in the role specification. Each method has a number of typed arguments. This class skeleton can be easily obtained from the role specification. The arguments of a local reaction are the same as in the corresponding **ANY** construct. The arguments of an external reaction are extracted from the top-level **LET** statement of the corresponding role event. The argument types are deduced by analysing the predicates attached to **ANY** or **LET** statement. The special `Configure` method on Fig.1, line 1 is used to associate the event bodies with the event guards.

```

1: public class ManagerRole extends RoleSkeleton {
2:   public void Configure() {...}
3:   private void Get_new_item(Sellerss, Itemii, Integer nn) {...}
4:   private void Take_bid(Buyer bb, Item ii, Integer pp) {...}
5:   private void Determine_winner(Item ii) throws CamaException {...}
6:   private void Receive_Payment(Buyer bb, Item ii, Integer pp) {...}
7:   private void Item_sold(Buyer bb, Item ii) throws CamaException {...}
8: }

```

Fig. 1. The overall structure of Manager role implementation

Fig.2 shows how abstract B notation is mapped (translated) into executable Java code. Once concrete representation for abstract role variables is decided, B statements can be easily translated into their Java counterparts.

The role specification variables should be represented in the chosen implementation language which means that they must have implementable types. A partial function, the type used for the most objects in our example, can be represented as a hash map or a sparse array. A set can be a simple array or a special set class, such as `java.util.Set`. As already mentioned in the connection with role invariant, we do not have to worry about most type restrictions, such as a

```

1: private void Take_bid
2: (Buyer bb, Item ii, Integer pp) {           Take_bid = ...
3: if (items.have(ii) &&                       IF ii : dom(items) ∧
4: items.getNumber(ii) > 0 &&                   items(ii) > 0 ∧
5: pp > bids.getBids(ii).getMaxBid() &&       pp > max(ran(bids(ii))) ∧
6: !winners.have(ii)) {                       not(ii : dom(winners)) THEN
7: bids.add(ii, bids.getBids(ii).add(bb, pp)); bids(ii) := bids(ii) <+{bb ↦ pp}
8: items.set(ii,                               items(ii) :=
9: new Integer(items.getNumber(ii) - 1)); items(ii) - 1
10: announceLocalEvent(
11: ManagerInternalEvent.Take_bid, ii);
12: }
13: // request is removed automatically       man_buffer := tail(man_buffer)
14: }                                         END;

```

Fig. 2. Translation of a reaction body

set being a strict subset or a function type being an injection. As a result, we have to deal only with the two basic data types: sets and functions.

When an event triggering some external reaction occurs, the corresponding reaction is invoked and the data are passed as the reaction arguments. Special utilities (part the CAMA middleware) are responsible for unpacking arguments, checking guard conditions and delegating control to the subroutine implementing the reaction body. The argument types, along with the request type, are associated with the reaction body during the configuration phase. For example, the event Bid (Fig.3) is associated (by special method `addReaction`) with the request type `NewItem`.

```

1: addReaction(Request.NewItem, WHEN not(man_buffer =<>)
2: "Take_Bid");                               ReqType(first(man_buffer)) = Bidding ...
3: .                                           bb = Buyer(first(man_buffer)) ∧
4:                                           ii = Item(first(man_buffer)) ∧
5:                                           nn = INumber(first(man_buffer))
6:

```

Fig. 3. Registering an external reaction

Both local and external reactions can send events that will trigger a reaction in another agent. In a formal specification this is expressed by storing a request in the corresponding output buffer. This is implemented by a special dedicated method used to send a request. The request arguments are passed as a tuple (Fig.4, line 5). The request itself is also a tuple with the additional fields representing the request type and the destination role.

The procedure for invocation of local reactions is slightly different. In a role specification, local reactions are invoked when the operation guard expressed over a subset of role variables is true. Evaluation of the guard predicate in **ANY** statement is combined with the selection of arguments for a reaction. For

```

post("Buyer", Request.PaymentReq, new Tuple().
    add(winners.getBuyer(ii)).add(ii));

```

Fig. 4. Sending a message

example, **ANY** v **WHERE** $f = 1 \wedge v \in S$ **THEN** ... enables a local reaction when the local role variable f equals 1 and exists such argument v that it belongs to the set S . There is no direct counterpart for this construct in procedural languages and it can be prohibitively expensive to analyse the whole program state to compute the operation guard. Analysis of role specifications, however, suggests a solution specific for our approach. The solution is based on the observation that the arguments for local reactions are determined by updates of the related variables in one of the recently executed reactions. This results in a translation rule that any reaction updating a variable, which is mentioned in the guard of a local reaction, must signal the corresponding local event.

The local reaction guard is implemented as a special routine called when the corresponding event is signaled. Such routine, called *event filter*, formulates restrictions on the state of the role variables (Fig.5, line 3-11) and the candidate arguments. The event filter body is obtained from a formal specification of the corresponding event guard. In the example on Fig.5, the call to `addLocalReaction` associates the local reaction `Determine_Winner` with the internal event `ManagerInternalEvent.Take_Bid`. The event filter is a translation of the formal event guard (Fig.5, lines 8-10). The external reaction `Take_Bid` announces the internal event `ManagerInternalEvent.Take_Bid` (Fig.2, line 10) whenever the object `bids` is updated (Fig.2, line 10). The event filter checks whether the bidding process for the item has finished and, if it is so, the local reaction `Determine_Winner` is invoked.

```

1: addLocalReaction(ManagerInternalEvent.Take_Bid,
2: "Determine_Winner", this,
3: new LocalEventFilter() {
4: public boolean filter(Object o, Object c) {
5: Item ii = (Item) o;
6: ManagerRole manager = (ManagerRole) c;
7: if (manager.bids.have(ii) &&                               ii : dom(bids)
8: manager.items.getNumber(ii) == 0 &&                       items(ii) = 0
9: !manager.winners.have(ii))                                not(ii : dom(winners))
10: return true; else return false; });

```

Fig. 5. Registering a local reaction and setting its guard

Though code generation for the case study has been done manually, it is clear that this process can be automated. AtelierB toolkit[7] supports automated code generation for a number of programming languages. Unfortunately, it does not support the languages popular in agent systems, such as Java, Python and C#. Also, the AtelierB code generation requires very detailed input specifications

that are sometimes very hard to construct. The other limitation stems from the fact that it is a general-purpose code generator and it does not account for peculiarities of agent systems. We are planning to design and implement our own tool for generating executable agent code from formally designed specifications.

From the implementation viewpoint, the application of the formal development procedure presents a number of advantages. Implementations derived from formal specifications tend to be more compact, elegant and simple in structure. This is also true for our auction case study. The discussed ManagerRole contains just 125 lines of code (excluding comments) from which 50 LOC account for reaction bodies and 20 LOC for reaction guards. This favorably compares to the conventional development technique employed for the same auction application. The most noticeable differences from the formally designed implementation are: a significant increase in the use of coordination operations, more lines of code (about 4 times more), poor code readability and complex code structure (several times more branching statements) .

7 Discussion

In this paper we presented a formal approach to ensuring inter-operability of mobile agent systems. Our approach uses the specification and refinement techniques of Event-B as the basis for ensuring system correctness. Essentially, we have started from a formal specification of an multi-agent application and by refinement decomposed it into a set of specifications defining the behaviour of an agent in a particular role together with its communication mechanisms. We demonstrated that the implementation of an agent from such a formal specification is rather straightforward and has advantages over ad-hoc programming of agents. This was illustrated by excerpts from the code generated for the electronic auction. The further extension of our approach could result in a tool allowing automatic generation of the agent code from a formal specification.

One of the major concept used in our approach is an agent role. Our definition of a role is close to one proposed by Kendall [11]. In this work she emphasises the need of a formalism for role specification and proposes the aspect-oriented programming as a candidate to solve this problem. However, in her approach she does not address the problem of interoperability of agent roles, a solution to which we have proposed in this paper.

The formal reasoning about mobile agent systems has been also conducted by [13, 14]. While their formalisation is more expressive from the point of view of context handling, they focus more on specifying agent systems rather than on demonstrating how such specifications can be used in the development process, for instance, to address the problem of agent interoperability. Our approach is more advantageous from this point of view.

As a future work we are planning to develop a set of explicit specification and communication patterns which can be used for ensuring agent interoperability in the development of mobile agent systems.

Acknowledgments

This work is supported by IST FP6 RODIN Project.

References

1. Event-B Language. RODIN Deliverable 3.2, online at <http://rodin.cs.ncl.ac.uk/>.
2. J.-R. Abrial. Event Driven Sequential Program Construction. 2000. Available at <http://www.matisse.qinetiq.com>.
3. J.-R. Abrial and L.Mussat. Introducing Dynamic Constraints in B. Second International B Conference, LNCS 1393, Springer-Verlag, April 1998.
4. A.Iliasov, L.Laibinis, A.Romanovsky, and E.Troubitsyna. Rigorous Development of Fault Tolerant Agent Systems. TUCS Technical Report. March 2006.
5. A. B.Arief and A.Romanovsky. On Using the CAMA Framework for Developing Open Mobile Fault Tolerant Agent Systems. University of Newcastle. 2006.
6. N. Busi, A. Rowstron, and G. Zavattaro. State- and event-based reactive programming in shared dataspace. In *Coordination Models and Languages*, pages 111–124, 2002.
7. Clearsy. AtelierB: User and Reference Manuals. Available at http://www.atelierb.societe.com/index_uk.html.
8. E. Dijkstra. *A Discipline of Programming*. Prentice-Hall International, 1976.
9. D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.
10. A. Iliasov and A. Romanovsky. CAMA: Structured Coordination Space and Exception Propagation Mechanism for Mobile Agents. Presented at ECOOP 2005 Workshop on Exception Handling in Object Oriented Systems: Developing Systems that Handle Exceptions. July 25, 2005. Glasgow, UK, 2005.
11. E. A. Kendall. Role modeling for agent system analysis, design, and implementation. *IEEE Concurrency*, 8(2):34–41, 2000.
12. L.Laibinis, A.Iliasov, E.Troubitsyna, and A.Romanovsky. Formal Approach to Ensuring Interoperability of Mobile Agents. TUCS Technical Report. June 2006.
13. G.-C. Roman, C.Julien, and J.Payton. A Formal Treatment of Context-Awareness. Proceedings of FASE'2004, LNCS 2984, Springer-Verlag, March 2004.
14. G.-C. Roman, P.McCann, and J.Plun. Mobile UNITY: Reasoning and Specification in Mobile Computing. *ACM Transactions of Software Engineering and Methodology*, July 1997.