# Towards Formal Development of Mobile Location-Based Systems

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

[1]Newcastle University, UK. {Alexei.Iliasov, Alexander.Romanovsky}@ncl.ac.uk
[2]Åbo Akademi, Finland. {Linas.Laibinis, Elena.Troubitsyna}@abo.fi

## 1. Introduction

### 1.1 Motivation

Mobile agents have many attractive features to offer and they are often mentioned as a future mainstream industry-level software technology. The agent technology naturally solves the problem of decoupling complex software into smaller parts that are easier to design, code and maintain. It helps to use distributed computing power effectively while hiding many of the details and complexities of a hosting environment. Recent advances in mobile computing and wireless networks lead to introduction of host (physical) mobility that offers totally new opportunities and as well raises new problems. Though substantial research has been conducted on developing middleware solutions supporting mobile agents, the mobile agent technology is still not mature enough to become a practice in industrial software development. There are several areas in which no general solutions have been found yet. One of them is ensuring interoperability of independently designed agents and correctness of the overall mobile system. In this work we will present a background for building a formal development methodology that addresses this problem.

Agent software is designed to interact with other agents during its lifetime. Most research in the area discusses only centralized development process, when all the participating pieces of software (code of the agents) are created at the same site to solve common problems. In this case agents are mostly useful as a replacement of conventional client-server scheme with migrating clients or/and servers. However the application area of the mobile agents is much broader and, to make full use of their communication and migration capabilities, we need to assume systems are composed dynamically out of agents developed independently at different sites and for different purposes. Such configurations are impossible if agents are merely anonymous black boxes. In our view, to cooperate, agents must be based upon some common specification of their functionality. This specification should be formally developed and verified to ensure the desired properties of the application composed of agents. Developers of individual agents can independently extend the specification (using a refinement method) to add unique features without losing compatibility with other agents derived from the same specification.

The specification should be minimal in a sense that it does not have to provide many design details but it should be complete enough to identify what services the agent has to offer and what services it is looking for. This information should describe how to communicate with the particular class of agents, what such agents expect as input, and what output they produce.

## 1.2 Background

Mobile agent systems are often symmetric in a sense that each system participant roughly carries the same middleware implementation. Agents can dynamically and autonomously form new groups and communicate. However in this paper we explore an asymmetric approach in which different parts of the system carry different basic functionality. One particular example of such view is *a location-based* scheme. In this model locations provide services to the agents, such as connectivity and coordination space. Agents are not able to communicate with each other without a location support. The choice of the scheme is supported by the fact that the majority of the mobile applications assume that agents meet in physical or logical locations providing a set of designated services to them. Hence, the asymmetric scheme is closer to the traditional service provision architectures. It can support large-scale mobile agent networks in a very predictable and reliable manner. It makes better use of the available resources since most of the operations are executed locally. Moreover, location-based architecture eliminates the need for employing complex distributed algorithms or any kind of remote access. This allows us to guarantee atomicity of certain operations without sacrificing performance and usability. This scheme also provides a natural way of introducing context-aware computing by defining location as a context. The main disadvantage of the location-based scheme is that an additional infrastructure is always required to support mobile agent collaboration.

The *coordination paradigm* (originated in Linda [4]) has become the dominating environment in which a number of mobile systems are built (including Lime [7], Klaim [2], etc.). Linda is a set of language-independent coordination primitives that can be used for communication and coordination between several independent pieces of software. First used for parallel programming, it later became a core component of many mobile software systems because it fits nicely the main characteristics of the mobile systems: openness, dynamicity, anonymity of agents and their loose coordination. Linda-based coordination systems specifically designed for mobile applications supporting both physical mobility, such as a device with running application travelling along with its user across network boundaries, and logical mobility, when a software application changes its hosting environment.

The rest of the paper is organized as follows. Section 2 introduces a number of basic abstractions to be used in development of mobile systems. Sections 3 describes a rigorous development process supporting these abstractions. Section 4 presents a formal abstract specification of the middleware. Finally, the last section presents conclusions and outlines our future work.

## 2. System structure

The CAMA (**context-aware mobile agents)** system consists of a set of locations. Active entities of the system are agents. An agent is a piece of software that meets a number of requirements. Each agent is executed on its own platform. The platform provides execution environment interface to the location middleware. Agents communicate only with other agents in the same location. Agents can migrate from location to location logically (connections and disconnection) or physically (e.g. movement of a PDA on which the agent is hosted on). They can also logically migrate from platform to platform using weak code mobility. Compatible agents collaborate through a scoping mechanism. A scope defines a joint activity of several agents. The scoping mechanism also isolates non-compatible agents from each other. Below are the details of the introduced concepts.

A ***location*** is a container for scopes. It can be associated with a particular physical location and can have certain restrictions on the types of supported scopes. It is the core part of the system as it is provides means of communications and coordination between agents. Location is a named entity and for simplicity we assume that each location has a unique name in the given context. This roughly corresponds to IP addresses of hosts on network which are often unique in some local sense. Location must keep track of present agents and their properties in order to be able to automatically create new scopes and restrict access to existing ones. The more detailed location description is presented in the form of a formal specification (see Section 4).

Certain locations may prevent agents from entering without an authorization. To be allowed to enter a location, an agent must have a key issued by it. Keys may be permanent or have a validity period determined by the issuing location Agent must have to acquire a key on a different location before entering a protected location.

Locations may provide special services, like access to a service from a variety of devices connected to the location, making enquires and so on. Each Location may have its own unique set of services and provided operations. They are made available to agents via what appears to agents as a normal scope though some roles in these scopes are implemented by the location system software. As with all scopes, agents are required to implement specific interfaces in order to connect to a location-provided scope. An example of such services includes printing on a local printer, access to Internet, making a backup to a location storage, migration and etc. In addition to supporting scopes as mean of agent communication, location may also support logical mobility of agents, hosting of platforms and agent backup. Hosting of platform on a location allows agent to execute without a PDA. For example, a user may decide to move an agent from his PDA to a location before leaving the location with his PDA. In addition to the above, location, by a request from an agent, may play in certain types of scopes a role of a trusted third party that is neutral to all the participating agents. This facilitates implementation of various transaction schemes.

A ***platform*** provides an execution environment for an agent. It is composed of a virtual machine for code execution, networking support and middleware for

interaction with location. A platform may be supported by PDA, smart-phone, laptop or a location server. The concept of platform is important to clearly differentiate between a location providing coordination services to agents and middleware that only supports agent execution. In other approaches no such distinction is made.

An ***agent*** is a piece of software implementing a set of roles which allow it to take part in certain scopes. All agents must implement the minimal functionality called the default role, which specifies activities outside scopes.

A ***scope*** is a dynamic container for tuples. It provides an isolated coordination space for compatible agents by restricting visibility of the tuples contained in the scope to the participants of the scope. Scopes are initiated by an agent and then atomically created by Location when all the participants are ready. Scopes can be nested and scope participants can create new contained scopes. Scope is defined by the set of roles and a set of logical restrictions.

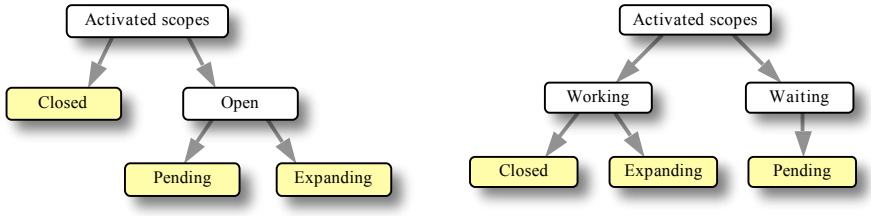


**Fig. 1.** Scope classification a) according to the availability of scopes for new agents and b) according to the agent activity in a scope.

A scope becomes *activated* after some agent creates it with the *CreateScope* operation. A scope is *open* when there are some vacant roles in it, and is *closed* when all the roles in it are taken. A scope is *pending* if some required roles are not taken yet and *expanding* if all the required roles are taken but there still some vacant roles. Closed and expanding states correspond to working scopes, where agents can communicate. All participants of a pending scope are blocked until the scope state is changed into closed or expanding.

A ***role*** is an abstract description of agent functionality. Each role is associated with some scope type. An agent may implement a number of roles and can also play several roles in the same scope or different scopes. There is formal relationship between a scope and a role of a scope.

Introduction of scopes and roles offers agents an entirely new way to discover each other and to collaborate with each other. After arrival to a new location, an agent looking for partners, initiates scope creation or join protocol. They are implemented as a request to the controlling system (middleware) to find appropriate partners ready for certain type of activity. In a request agent specifies type of scope it wants to work in and a role it is going to take. The system then creates a scope or finds an existing matching scope with available role for the agent. This procedure is executed

atomically. As soon as all the required roles are taken, the system creates a separate coordination space for the group of agents participating in the scope. Isolation achieved this way greatly simplifies agent design since while in a scope agent may safely assume reasonable behaviour of their partners. In a scope agents remain anonymous as long as they need and procedures of scope joining or creation do not change this.

The CAMA approach supports the context-awareness of mobile agents. The context of an agent in CAMA systems consists of is composed of the following parts: a set of locations the agent is connected to, the state of scopes in which the agent is currently participating (including tuples contained in these scopes) and role attributes of other agents in collaborating with the agent.

## 3. Formal Development Process

Formal development process of the CAMA system consists of several steps. First, we create abstract specifications of the middleware (location) and the scopes that will be supported by the system. Then we develop (by the stepwise refinement method) specifications of different roles participating in scopes. Finally, we compose an agent specification as a combination of several developed roles (i.e., agent interfaces) and the default functionality defining the agent behaviour outside scopes.

The agent specification can be further refined adding more details and custom functionality. Compatibility of different agents is ensured by the fact that all agents have been developed by the formal refinement method from the same abstract specifications of different roles and the middleware. Therefore, agents can collaborate making safe assumptions about the functionality of their peers.

In the next subsection we give a brief introduction into our formal framework – the B Method, which we will use to formalise the development process described above.

### 3.1 The B Method

The B Method [1] (further referred to as B) is an approach for the industrial development of highly dependable software. The method has been successfully used in the development of several complex real-life applications [6]. The tool support available for B provides us with the assistance for the entire development process. For instance, Atelier B [8], one of the tools supporting the B Method, has facilities for automatic verification and code generation as well as documentation, project management and prototyping. The high degree of automation in verifying correctness improves scalability of B, speeds up development and, also, requires less mathematical training from the users.

The development methodology adopted by B is based on stepwise refinement [3]. While developing a system by refinement, we start from an abstract formal specification and transform it into an implementable program by a number of correctness preserving steps, called *refinements*. A formal specification is a

mathematical model of the required behaviour of a (part of) system. In B a specification is represented by a set of modules, called Abstract Machines. An abstract machine encapsulates state and operations of the specification and as a concept is similar to a module or a package.

Each machine is uniquely identified by its name. The state variables of the machine are declared in the VARIABLES clause and initialized in the INITIALISATION clause. The variables in B are strongly typed by constraining predicates of the INVARIANT clause. All types in B are represented by non-empty sets. We can also define local types as *deferred sets*. In this case we just introduce a new name for a type, postponing actual definition until some later development stage.

The operations of the machine are defined in the OPERATIONS clause. In this paper we use Event B extension of the B Method. The operations in Event B are described as guarded statements of the form SELECT cond THEN body END. Here cond is a state predicate, and body is a B statement. If cond is satisfied, the behaviour of the guarded operations corresponds to the execution of their bodies. However, if cond is false, then the execution of the corresponding operation is suspended, i.e., the operation is in waiting mode until cond becomes true.

The generalised version of the guarded operation is ANY operation. The syntax of ANY operation is ANY vars WHERE cond THEN body END. The operation corresponds to a family of events or a parameterised event operation. It is triggered by any acceptable values of the variables vars satisfying the condition cond. The variables vars are then used as local variables in the operation body.

B statements that we are using to describe a state change in operations have the following syntax:

S == x := e | IF cond THEN S1 ELSE S2 END | S1 ; S2 | x :: T |
S1 || S2 | ANY z WHERE cond THEN S END | ...

The first three constructs – assignment, conditional statement and sequential composition (used only in refinements) have the standard meaning. The remaining constructs allow us to model nondeterministic 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. The detailed description of the B statements can be found elsewhere [1].

### 3.2 Development of Scopes and Roles

The specification of a scope describes general functionality of several collaborating agents (in particular roles). The task of formal development is to use the specification as the starting point for the derivation of specifications of the corresponding agent roles (interfaces). To guarantee correctness of the resulting role specifications, we use formal refinement and decomposition techniques. For example, Fig.2 shows that the Lecture scope is decomposed into roles Student and Teacher defining functionality of the corresponding agents.

On the other hand, we have to take into account scope nesting, when scopes have embedded subscopes providing some extended functionality. Subscope specifications can be naturally derived from the original scope specification via refinement. After verifying the correctness of refinement, we can continue the development process by decomposing the specification into corresponding roles as described above. In Fig.2, we show how scope Lecture is refined by subscope Group work, which is consequently decomposed into roles Student' and Teacher'.

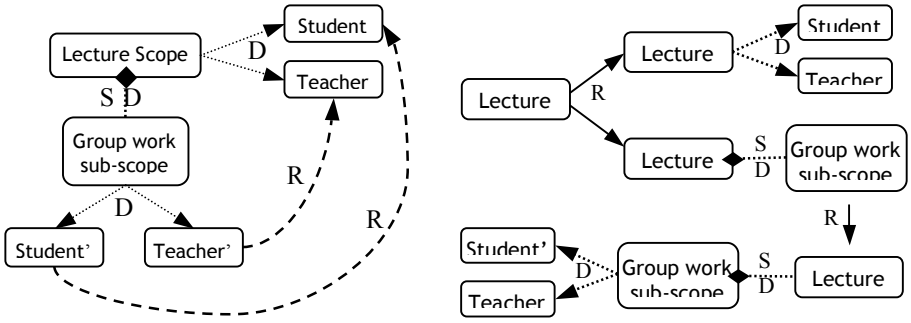


**Fig. 2.** a) Orthogonal decomposition diagram b) its representation as a parallel refinement. SD is scope decomposition; D – decomposition of a scope into roles; R – refinement.

As a result, we have two orthogonal development processes with the same starting point – the original specification of a scope. Both developments arrive at role specifications describing agent functionality in the corresponding scopes. However, the hierarchy of scopes and subscopes should be reflected in the corresponding specifications of agent roles. Hence the roles in subscopes must be the extensions of the corresponding roles in the scopes. In other words, to guarantee the consistency of developed roles, we have to show that the subscope roles refine the corresponding scope roles.

In our Lecture scenario, we derived the specifications of agents in roles Student and Teacher. These specifications describe the functionality of the corresponding agents after joining scope Lecture. On the other hand, roles Student' and Teacher' describe the behaviour of the corresponding agents while they enter scope Group Work which is a subscope of Lecture. These roles have to satisfy the requirements specified in Student and Teacher. At the same time, they can provide additional functionality specific to Group Work. By proving formally that Student' is a refinement of Student, and Teacher' is a refinement of Teacher, we guarantee consistency of agent behaviour in nested scopes Lecture and Group work. In Fig.2, this is shown by the arrows connecting roles Student' and Student, and roles Teacher' and Teacher.

### 3.3 Agent Design

Agent design starts with the selection of roles that the agent must implement. It is permitted to implement any number of roles from different scopes. Initially roles inside of an agent are totally independent specifications that may well correspond to several independent processes running in an agent. Agent refinement specifies additional operations that control agent behaviour during migration, location selection, scope creation and joining, and other activities not covered by roles.

During agent refinement process, the agent roles can also be refined, possibly by adding some new functionality. Due to the nature of refinement, the refined roles are still compatible with the original abstract roles.

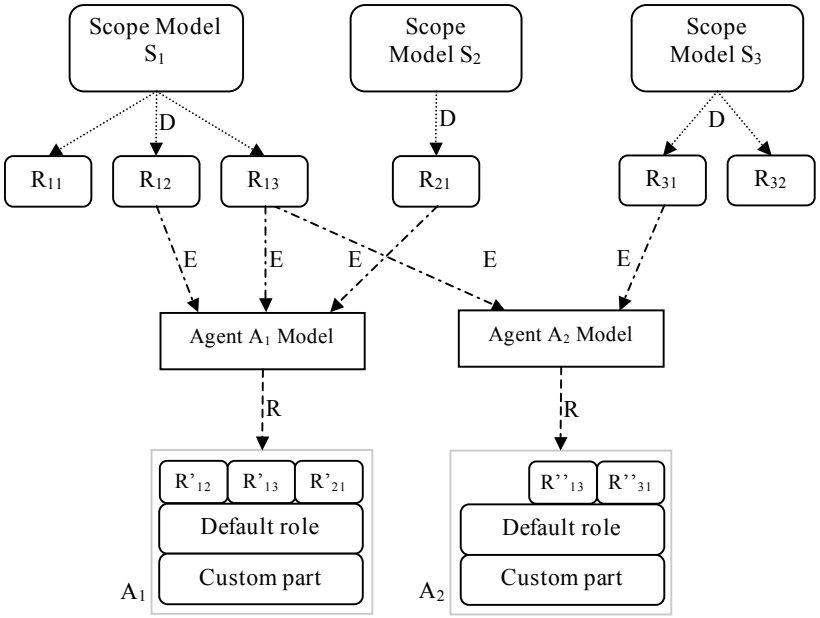


**Fig.3.** Relations between agents, scope models and roles. D – decomposition of a scope into roles; E – extension of role specification an agent model; R – refinement of an agent model.

We start building an agent specification by extending one or more roles obtained formally through the decomposition of abstract scope models (see Fig. 3). The refinement step introduces a specification of the minimal agent functionality called the default role. It allows an agent to talk to locations, create/join/leave scopes, and migrate. The agent may also need some logic that glues independent interfaces and allows them to talk to each other. This is done via the global agent variables and the special methods for accessing to them.

After the agent specification is ready, it is used to build the source code for the actual agent program. The source is linked against the middleware library to get an executable agent program. The generated agent source may run on PDAs, laptops,

desktop PCs and smart-phones using the platform-specific middleware implementation as the adaptation layer.

The standard work cycle of an agent looks like this: an agent detects the available locations and connects to at least one of them, then looks for current activities on the location(s) or creates its own new scope, and finally joins a scope and plays one of the implemented roles in it. Only when the agent decides to play a particular role in a scope, it really starts to cooperate with other agents. The agent is capable of understanding its peers since the role functionalities of all the scope participants are based on the same abstract model. As a result, the composition of agent functionalities in a scope corresponds to the initial abstract model.

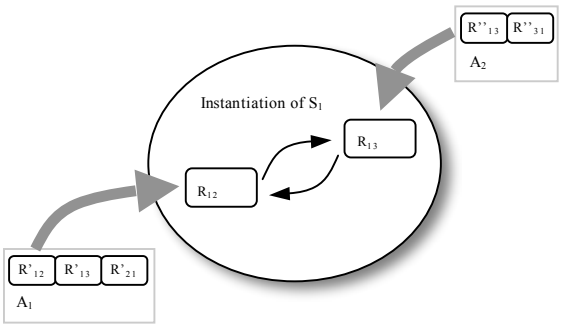


**Fig. 4.** An instantiation of an abstract model

The correctness of a model instantiation, or in other words, the fact that the scope instantiates the corresponding abstract scope model, can be demonstrated by analysing the agent design process and assuming that there is a correct transition from agent model to agent implementation. In Fig.4 we illustrate an instantiation of an abstract model which is formed when all the roles in the scope are taken by some agents.

## 3.4 Fault Tolerance

Ability to operate in a volatile, error prone environment will be an intrinsic feature of CAMA. Hence CAMA systems should be able to withstand various kinds of faults, i.e., guarantee fault tolerance. The most typical fault is a temporal connectivity loss which can cause failures of communication between cooperating agents or between an agent and the location.

Since in the CAMA approach the agent and location software are developed from the corresponding B specifications, the fault tolerance mechanisms should be already integrated into these specifications, so that development of fault tolerance means is becoming part of the system development. For example, while modelling collaboration between agents in the specification of a scope, we have to define the agent behaviour in the presence of message losses, hardware failures etc. Moreover, while developing agent roles (interfaces) from the corresponding scope specifications, fault tolerance mechanisms should be distributed between involved parties.

Representing fault tolerance in CAMA constitutes an important research topic which we will further investigate in our future work.

## 4 B Specification of the Middleware

To ensure correct behaviour of the location-based system, the middleware of the location should enforce a certain discipline on agents. For instance, the properties of the scopes defined upon scope creation are preserved in spite of volatile connectivity and dynamic nature of scopes. Moreover, it should guarantee the integrity of the information about agents in locations and scopes. These complex interdependencies should be stated explicitly and verified. We have developed a formal specification of the location middleware which is the core of the system. It corresponds to the most complex part of the system and not only defines the operations that the location provides to support communication between agents but also state the properties of the data structures in the location. The actual middleware implementation will be based upon this formal model. An abstract description of the location specification is presented below. The full B specification can be found in [9].

**MACHINE**
Location
**VARIABLES**
AgentNames, /* *Agents active in the location* */
Scopes, /* *Created scopes* */
ScopeRolesTaken, /* *A number of agents taken a particular role in a particular scope* */
AgentRoleData, /* *Public data disclosed by the agent while taking a certain role* */
AgentScopes, /* *For each active agent defines the scopes in which it is active* */
ScopeAttributes, /* *Scope descriptions provided by scope creators* */
ScopeAgentRoles /* *The roles taken by agents in active scopes* */
**INVARIANT**
*Types of variables & interdependencies between data*

**INITIALIZATION**
*Initially there are no agents and correspondingly no scopes in the location*
...
**OPERATIONS**

/* *Engagement request* */
a_id < --Engage =
**ANY** Role_and_Data **WHERE**
*Role_and_Data is the information about the supported roles supplied by the agent*
**THEN**
**CHOICE**
*successful engagement to the location by issuing valid ID to the agent via a_id and update of AgentNames and AgentRoles*
**OR**
*failed engagement to the location by issuing invalid ID to the agent*
**END**;
**END**;

/* *Disengagement request* */

```
rr <-- Disengage = …

/* Scope creation request from an agent */
scope_id <-- CreateScope =
  ANY a_id, scopeDescr, role WHERE
      a_id is ID of the agent requesting to create a scope
      scopeDescr defines the necessary conditions for joining a scope
      role: the role that the requesting agent a_id will play in the created scope
  THEN
    CHOICE
       successful scope creation by issuing valid scope ID via scope_id,
       updating list of active scopes Scopes and list of
       scope descriptions ScopeAttributes updating AgentScopes,
       ScopeRolesTaken and ScopeAgentRoles
    OR
      unsuccessful scope creation by issuing invalid scope ID via scope_id
    END
  END;

/* Scope remove request */
result <-- DeleteScope = …

/* Scope join request */
result <-- JoinScope =
  ANY a_id, scope_id, role WHERE
      a_id is ID of the agent requesting to join the scope
      scope_id is ID of the scope which the agent is attempting to join
      role is the role which a_id will play in the scope
  THEN
    IF
      the agent a_id is not already participating in scope_id &
      requested role is a valid role for the scope &
      conditions for participating in the scope are not violated
    THEN
      the agent a_id is successfully joined the scope
      the information about the agent is updated
      in AgentScopes, AgentRoles, and ScopeAgentRoles
      the information about the number of agents playing the role is updated for the scope
    ELSE
      the agent a_id is rejected to join the scope
    END
  END;

/* Scope leave request */
result <-- LeaveScope = …

/* Prompt information about the scopes in which an agent can participate */
scopes <-- GetScopes = …
END
```

## 5 Conclusions

The presented work is tightly linked to the Ambient Campus case study of the RODIN Project. One of the project goals is to develop the methodology (based on formal methods) that would allow us to fully model and build the mobile location-based systems. The requirements document (written for the Ambient Campus case study) is the first step towards creating the formal model of such systems.

At the same time, we are developing middleware that will support our mobile agent abstractions. This paper presents the formal B specification of the location, i.e., the core part of the middleware. The choice of the location-based architecture (discussed in [5]) has influenced all the parts of our work on the case study, including the methodology.

It is our plan to investigate more closely the agent design process. We are also planning to conduct several extensive experiments covering the full cycle of system development – starting from an abstract system model through all steps until we get running software.

**Acknowledgments**. This work is supported by IST FP6 RODIN Project.

## References

1. J.-R. Abrial. *The B-Book*. Cambridge Univ. Press, 1996.

2. R. De Nicola, G. Ferrari, R. Pugliese. *Klaim: a Kernel Language for Agents Interaction and Mobility*. IEEE Transactions on Software Engineering, 24(5):315-330, 1998.

3. E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

4. D. Gelernter. *Generative Communication in Linda*. ACM Computing Surveys. 7(1): 80-112, 1985.

5. A. Iliasov, A. Romanovsky. *Exception Handling in Coordination-based Mobile Environments*. Proc. of COMPSAC 2005. Edinburgh, (UK), July 2005. IEEE CS.

6. *MATISSE Handbook for Correct Systems Construction*. 2003.http://www.esil.univ-mrs.fr/~spc/matisse/Handbook/

7. G. P. Picco, A. L. Murphy, G.-C. Roman. *Lime: Linda Meets Mobility*. Proc of the 21st Int. Conference on Software Engineering (ICSE'99), Los Angeles (USA), May 1999.

8. Steria, Aix-en-Provence, France. *Atelier B, User and Reference Manuals*, 2001. Available at http://www.atelierb.societe.com/index.html

9. B Specification of Location. Available from http://www.abo.fi/~Linas.Laibinis/Location.mch