Università di Pisa
Dipartimento di Informatica

Technical Report:

The Mob$_{adtl}$ model and method to design network aware applications

G. Ferrari, C. Montangero, L. Semini, S. Semprini
Dipartimento di Informatica, Università di Pisa

ADDRESS: via F. Buonarroti 2, 56127 Pisa, Italy. TEL: +39 050 2212700 FAX: +39 050 2212726

The Mob$_{adtl}$ model and method to design network aware applications

G. Ferrari, C. Montangero, L. Semini, S. Semprini
Dipartimento di Informatica, Università di Pisa

Abstract

We introduce a high level declarative model for mobile and ad hoc computing. The model takes the form of a logical theory in a Unity-like linear time temporal logic and is specifically designed to deal with Network Awareness, Mobility, and Coordination. A companion design methodology based on formal refinements is also introduced in this paper. The model and its design methodology are validated through an illustrative case study, namely a system that coordinates the activities of mobile components over an ad hoc network.

1 Introduction

The model which underpins most of the wide area network (WAN) applications (e.g. Gnutella, Napster) typically consists of a large number of heterogeneous computational entities (sometimes referred to as nodes or sites of the network) that interact by exploiting network communication facilities. The various nodes are handled by different authorities having different administrative policies and security requirements. Components of WAN applications are characterized by a highly dynamic behavior: they have to deal with the unpredictable changes over time of the network environment, due to the availability of network connectivity, lack of services, node failures, network reconfiguration, and so on.

Current software technologies emphasize the notion of *web service* [17] as an evolutionary paradigm to control the design and the development of WAN applications. Conceptually, web services are stand-alone components that reside over the nodes of the network. Each web service is characterized by an interface which is network accessible through standard network protocols and describes the interaction capabilities of the service. Hence, WAN applications are developed by locating, combining and integrating together web services. In the next few years there will be standard networking infrastructures based on SOAP-UDDI-WSDL to develop applications that exploit web services. Commercial examples are SunOne and .NET proposed by Sun and Microsoft, respectively. However, independently from the technology, *service orchestration*, i.e. the exploitation in a highly dynamic environment (the web) of pre-existing components in such a way that their composition satisfies the requirements of the application, is still a challenging problem.

Research activities in coordination languages and models are improving the formal understanding of dynamically adaptable components. However, the relationships between the software design methodologies and the underlying technologies have not yet been precisely identified. A rigorous understanding of these relationships requires both a formal model of WAN computing and a companion design methodology. This paper introduces a formal model for WAN computing called Mob_{adtl}. The model takes the form of an axiomatic presentation in a Unity-like style and is specifically designed to deal with *Network Awareness*, *Mobility*, *Coordination* and *Refinement*.

Network Awareness WAN applications do not appear as single integrated computer facilities to their users as it is the case of traditional distributed applications. For instance, users of traditional distributed applications can invoke a service regardless of whether the service is local, remote or under the control of a different network authority. Instead, in the WAN setting the *awareness* of network information is crucial to choose the best services that match user's requirements. For instance, users can react to phenomena like network congestion by binding their devices to different available resources. Similarly, network awareness can be exploited to control resource usages and resource accesses in order to ensure and maintain certain security levels. Finally, network awareness can be exploited to provide the designer with as much information as possible about the sources of network exceptions, to specify robust exception handlers.

In Mob_{adtl}, a network application consists of a net of elaboration nodes: the *neighborhoods*, which are bounded environments where the application *components* live. Each neighborhood is equipped with a distinguished component, the *guardian*, which embodies the node authority: guardians monitor the components in the neighborhood and limit the resources they can use. The pair neighborhood/guardian fully identifies the network environment of a component and models network awareness explicitly.

Mobility WAN applications design and implementation also exploits the *mobility* paradigm. The main breakthrough is that WAN applications may exchange active units of behavior and not just raw data (*logical mobility*) [16, 9]. Mobility concepts emerge also when developing applications for nomadic devices with intermittent access to the network (*physical mobility*).

In Mob_{adtl} some components (called *agents*) have the computational ability of moving from neighborhood to neighborhood. Hence, the model supports logical mobility, and the units of mobility – the agents – are conceptually made up of code, data, and control. However, agents mobility greatly increases the possibility of bad behaviours and the feasibility of security attacks (e.g. denial of services). Our approach provides a fine grained mechanism to control logical mobility. In particular, guardians intercept agents' requests and decide which of them can enter or leave the neighborhood they control. Therefore, guardians also provide the machinery to define *security* policies [12]. The net of neighborhoods in Mob_{adtl} is described by the mutual knowledge of the guardians, i.e. each guardian knows which neighborhoods are reachable. This way we can model the changes in the topology of a net of devices that physically move.

Coordination WAN applications are highly decentralized and dynamically evolving. Hence, they should be easily scalable in order to manage addition/removal of services, subnetworks and users without requiring to be reconfigured. The Mob_{adtl} axiomatic presentation identifies a *coordination theory* which specifies how components are glued together. Hence, Mob_{adtl} permits to take coordination issues into account in the early design stages and not consider them later on top of existing infrastructures, and fosters the evaluation of the impact of specific policies on the overall architecture.

Refinement The Mob_{adtl} design methodology is based on the notion of refinement [3]. Each refinement step extends the Mob_{adtl} axiomatic presentation to specify new properties. In particular, the refinement steps of our basic model result in models with a more elaborated structure, with respect to the network topology, the security policies, etc. The refinements of the coordination section demonstrates the full potential of the design methodology. Indeed, these refinements lead to the complete specification of the properties of the underlying coordination middleware. Some experiment in this direction targeted to a specific technology, CORBA, are reported in [22].

The design methodology is supported by a *Proof Assistant*, called Mark (Mob_{adtl} Reasoning Kit) [13] that partially automates the verification process, making it feasible to avoid error prone "by hand" arguments.

The paper is organized as follows. In Section 2 we present the logic we use to specify mobile systems in Mob_{adtl}, and its proof system. Section 3 introduces the design methodology, based on a structured refinement strategy. We keep the specification of each class of homogeneous components separate from the specification of other classes, and refine them independently, in a fully compositional way. The interaction between different classes of components is described in a distinguished coordination theory. Sections 4 and 5 present Mob_{adtl} at two levels of refinement: we first state and formalize the requirements for the model, then we formally specify the Mob_{adtl} reference model. The properties stated at the requirements level can also be read as abstract and global properties of the reference model and help in understanding it. In Section 6 we validate pragmatically the model and the methodology by specifying a system which addresses the coordination of the activities of a conference program committee. The case study is of particular interest since it couples aspects of physical –the devices of the PC members are equipped with wireless technology, and build an ad hoc network during the meeting – and logical mobility –applications try to migrate towards another device in order to survive to the disconnection of their computational environment from the net.

2 The logic used in Mob$_{adtl}$

Mob$_{adtl}$ is specified using an asynchronous, distributed, temporal logic based on the Unity logic [21]. We build on the Unity logic since it is only a subset of linear time temporal logic, with a limited number of operators, which are rather easy to understand intuitively, and cannot be nested. However, the logic permits to express most of the important properties of a system. This suggests that Unity is simpler than linear time temporal logic and that proofs are more straightforward, making it a more appealing tool for the designer. The extensions we make permit to deal with:

- *Distributed asynchronous systems.* We name the *components* of the system and relate properties which might hold in distinguished components, although in an asynchronous setting.
- *Reactions to events.* Unity operators relate properties satisfied by the states of a computation. For instance $p \mapsto q$ says that each state satisfying p is eventually followed by a state satisfying q. Conversely, they are not suited to directly relate an event (the property capturing a state change, instead of generic state property) with conditions on the states of a computation. The ability to cope with events allows us to specify straightforwardly reactions to state changes (transitions).

The specification of a system of n components, $m^1, \ldots, m^n$, consists a set of formulae with the structure given in Table 1.

Table 1: Syntax.

$$
\begin{array}{lll}
F & ::= & M\colon S \;\textsc{unless}\; S \;\Big|\; M\colon A \;\textsc{when}\; S \;\textsc{needs}\; S \;\Big|\; \textsc{init}\; DS \;\Big|\; M\colon S \;\textsc{leads\_to}\; DS \;\Big| \\
 & & M\colon A \;\textsc{when}\; S \;\textsc{op}\; DS \quad (\;\textsc{op}\; \in \{\textsc{because\_c}, \textsc{because}, \textsc{causes\_c}, \textsc{causes}\}) \\
DS & ::= & M\colon S \;\Big|\; DS \wedge DS \;\Big|\; DS \vee DS \\
S & ::= & A \;\Big|\; \neg S \;\Big|\; S \wedge S \\
A & ::= & \text{atom} \\
M & ::= & \text{component name or component variable}
\end{array}
$$

Two local operators are derived: $M :\ \textsc{stable}\ S \stackrel{def}{=} M : S\ \textsc{unless}\ false$

$M :\ \textsc{inv}\ S \stackrel{def}{=} \textsc{init}\ M : S \wedge M :\ \textsc{stable}\ S$

We often omit $\textsc{when}\ true$.

We let variables M, N, M^1, M^2, ... range over the set of component names.

A term is one of: a variable, a constant symbol, or a compound structure $f(a_1, ..., a_n)$ where a_i are terms. If p is a predicate symbol and t_i are terms, then

$p(t_1, ..., t_n)$ is an atom; *true* and *false* are also atoms. The set of the atoms is called A. In some cases, e.g. with predicate $\neq$, we use the infix notation. A *state formula* S is built up from A using negation ($\neg$) and conjunction ($\wedge$). Quantification is implicit, as discussed in Sect. 2.1. We call *localized state formula*, a formula $M:S$ where M is a component name and S a state formula. A *distributed state formula* DS is built on localized state formulae using conjunction and disjunction. *Formulae* describe computations relating distributed state formulae by the temporal operators CAUSES_C, CAUSES, NEEDS, BECAUSE_C, BECAUSE, WHEN, and the Unity [8] operators LEADS_TO, UNLESS, STABLE, and INV.

2.1 A Note on Quantification

Formulae are written without explicit quantification. The intended meaning is that a formula F is universally quantified over all values of the variables appearing in the premises of F, and existentially quantified on the remaining variables. For example

$$M : p(x) \text{ WHEN } q(y) \text{ CAUSES } N : r(x, y, z)$$

should be understood as prefixed by $\forall\, M, x, y\ \exists\, N, z$.

The domain over which a variable is quantified (i.e. its sort) can be understood from the context or explicitly defined. We assume these sets to be invariant during time.

To capture variables binding, e.g. y and M^2 in the following example, rules are to be understood as universally quantified on all the variables in the premises. These are also all the variables in the rule, since we do not permit the introduction of fresh variables in rule consequences. For instance, the following rule is to be read as if preceded by $\forall M^1$, M^2, M^3, x, y, z.

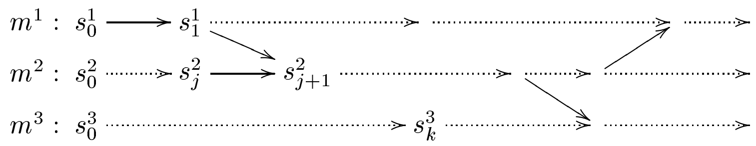
$$\frac{M^1 : p(x) \ \text{ CAUSES } \ M^2 : q(y) \quad M^2 : q(y) \ \text{ LEADS\_TO } \ M^3 : r(z)}{M^1 : p(x) \ \text{ CAUSES } M^3 : r(z)}$$

We do not want the variables existentially quantified in the formulae appearing in the rule premises to be universally quantified in the formulae in the rule consequences. Therefore, some additional constraints on the variables can be specified as side conditions of the rules. Their satisfaction might be obtained with a variable renaming. This holds in particular in the "cut" rules, like the transitivity above. Examples can be found in Section 2.3.

2.2 Semantics

Mob$_{adtl}$ formulae are interpreted on *computations*. A computation is a rooted graph of states like those in Figure 1. States are either the result of a local transition or of a communication (dotted arrows denote sequences of transitions, plain arrows single transitions or communications). We call s^i_j the j^{th} state of component m^i.

Figure 1: Computation of a system with three components



An event *occurs* when a given property is established, i.e. it *becomes* true in a state. Operators causes_c, causes, needs, because_c, and because permit to deal with events. We only consider atomic events, and the atoms preceding the temporal operators are to be read as events. The state formulae following the operators are normal conditions. Some examples of formulae follow.

$$m^1\!:\!p \ \textsc{when}\ c\ \textsc{causes}\ (m^2\!:\!q \wedge m^1\!:\!r) \quad (1) \qquad\qquad m^1\!:\!q\ \textsc{causes}\ m^3\!:\!r \quad (2)$$

$$m^2\!:\!q \ \textsc{when}\ c\ \textsc{because}\ (m^2\!:\!r \wedge m^1\!:\!p) \quad (3) \qquad\qquad m^2\!:\!q\ \textsc{needs}\ m^2\!:\!c \quad (4)$$

For instance, (2) specifies that the event q in m^1 is sufficient to let r hold (causes) in m^3 in the future. Note that causes is just a temporal operator, and that there can be other causes for r in m^3 than q in m^1. Necessary causes are specified using because: the condition must necessarily hold before the event occurs. needs requires a condition to hold at the time the event occurs. Finally, when is used to express more complex properties, by *subjecting* the effects of an event to additional conditions. Formula (1) states that event p occurring in a state of m^1 in which c holds, must be followed by states of m^2 and m^1 satisfying q and r, resp., as in the computation to the left in Figure 2. The computation to the right is a model of formulae (3) and (4).

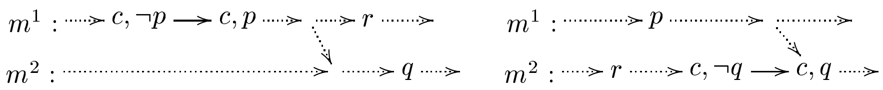


Figure 2: Examples of computations.

Suffix c stands for *closely*: causes_c requires the condition to hold in the same state in which the event occurs, or in the next one; because_c requires the condition enabling the event to hold in the *close past*, i.e. in the same or in the previous state.

A system satisfies a specification if and only if all its computations are models of all the formulae in the specification. Formally, a computation $\mathcal{C}$ of a system is a model of a formula F, if and only if: *(a)* all the components named in F are in $\mathcal{C}$, and *(b)* the states in $\mathcal{C}$ satisfy appropriate conditions given by cases on the structure of F in Table 2. Communications and local transitions define the

Table 2: Semantics. The semantics defines the conditions a computation has to verify to be a model for a formula, given by cases on the structure of formula.

$M^i: A$ WHEN B NEEDS C	:	$\forall j \geq 0.\ s^i_j \models A! \wedge B \ \rightarrow\ s^i_j \models C$
$M^i: A$ WHEN B BECAUSE/BECAUSE_C DS		$\forall j > 0.\ s^i_j \models A! \wedge B \ \rightarrow\ s^i_j \models$ P/P_C DS
$M^i: A$ WHEN B CAUSES/CAUSES_C DS	:	$\forall j \geq 0.\ s^i_j \models A! \wedge B \ \rightarrow\ s^i_j \models$ F/F_C DS
$M^i: A$ UNLESS B	:	$\forall j \geq 0.\ s^i_j \models A \wedge \neg B \ \rightarrow\ s^i_{j+1} \models A \vee B$
$M^i: A$ LEADS_TO DS	:	$\forall j \geq 0.\ s^i_j \models A \ \rightarrow\ s^i_j \models$ F DS
INIT DS	:	$s^i_0 \models$ BEGIN DS for all i

Where for V valuation function, OP $\in \{$ BEGIN , P_C , P , F_C , F $\}$:

$s^i_j \models p(\bar{x})$	iff	$p(\bar{x}) \in V(s^i_j)$ ($\bar{x}$ is a tuple of bounded terms)
$s^i_j \models \neg A$	iff	$s^i_j \not\models A$
$s^i_j \models A \wedge B$	iff	$s^i_j \models A$ and $s^i_j \models B$
$s^i_j \models A!$	iff	($s^i_j \models A$ and $s^i_{j-1} \not\models A$) or ($j = 0$ and $s^i_0 \models A$)
$s^i_0 \models$ BEGIN $M^k : A$	iff	$s^k_0 \models A$
$s^i_j \models$ P_C $M^k : A$	iff	($i = k$ and $s^i_j \models A$) or ($\exists s^k_l.\ (s^k_l,\ s^i_j) \in \mathcal{N}$ and $s^k_l \models A$)
$s^i_j \models$ P $M^k : A$	iff	$\exists s^k_l \leq s^i_j.\ s^k_l \models A$
$s^i_j \models$ F_C $M^k : A$	iff	($i = k$ and $s^i_j \models A$) or ($\exists s^k_l.\ (s^i_j,\ s^k_l) \in \mathcal{N}$ and $s^k_l \models A$)
$s^i_j \models$ F $M^k : A$	iff	$\exists s^k_l \geq s^i_j.\ s^k_l \models A$
$s^i_j \models$ OP $DS \wedge DS'$	iff	$s^i_j \models$ OP DS $\wedge$ OP DS'
$s^i_j \models$ OP $DS \vee DS'$	iff	$s^i_j \models$ OP DS $\vee$ OP DS'

next state relation $\mathcal{N}$: this relation contains all the pairs (s^i_j, s^i_{j+1}), plus some pairs (s^i_j, s^k_l), with $i \neq k$, that denote the communications. The *future state* relation $\geq$ is the reflexive and transitive closure of relation $\mathcal{N}$. Accordingly to Section 2.1, all variables in F are bound.

Note that most operators can be distributed: a notable exception is NEEDS, which can be used only locally. Indeed, for NEEDS to hold, the event and its enabling condition must be synchronous, and this notion is defined only locally in an asynchronous world.

There is an asymmetry between CAUSES (CAUSES_C) and BECAUSE (BECAUSE_C): in the case of CAUSES the event in the initial state causes a reaction, while the semantics of BECAUSE says that the event can freely occur in the initial state, without requiring the condition to hold. The choice made for CAUSES permits to specify also closed systems, where the execution can be triggered by the data in the initial state, even in the absence of external actions on the system. Also, we can say that a distributed formula DS will eventually hold, by requiring $true$ $\textsc{causes}$ DS. Conversely, we do not take into account the past conditions enabling an event occurring in the initial state.

Finally, even though the asynchronous model of communication forbids global assertions on the states of different components, we can assume that a global knowledge is possible of the initial state. The semantics of operator $\textsc{init}$ captures this fact: in the initial state, it is possible to "know" the properties which hold in each component.

2.3 Rules of the logic

We present here the most relevant theorems among those we used in the refinement proofs. More theorems can be found in [1]. Theorems have been proved exploiting the semantics in Table 2.

One of the simplest is the transitivity of $\textsc{because}$, while a chain of $\textsc{causes}$ needs some side condition to be transitively closed. The reason is that we need to be sure that *event* $N : B$ occurs after event $M : A$, while the first premise only requires that *condition* $N : B$ follows $M : A$.

$$\textit{because transitivity}\qquad \frac{M : A\ \textsc{because}\ N : B \qquad N : B\ \textsc{because}\ DS}{M : A\ \textsc{because}\ DS}$$

$$\textit{causes chain}\qquad \frac{\begin{array}{cc} M{:}A\ \textsc{causes}\ N{:}B & N{:}B\ \textsc{causes}\ DS \\ N{:}B\ \textsc{because}\ M{:}A & \textsc{init}\ N{:}\neg B \end{array}}{M{:}\ A\ \ \textsc{causes}\ DS}$$

In order to avoid undesired variable capturing, these two theorems come with the side condition: $vars(M{:}A) \cap vars(DS) \subseteq vars(N{:}B)$.

If we consider formulae including the $\textsc{when}$ operator, things become more complex: to apply chaining to $M{:}A\ \textsc{when}\ C\ \textsc{causes}\ N{:}B$ we need the following rule.

$$\textit{when causes chain}\qquad \frac{\begin{array}{c} M{:}A\ \textsc{when}\ C\ \textsc{causes}\ N{:}B \qquad N{:}B\ \textsc{causes}\ DS \\ M{:}A\ \textsc{needs}\ C \qquad N{:}B\ \textsc{because}\ M{:}A \qquad \textsc{init}\ N{:}\neg B \end{array}}{M{:}\ A\ \textsc{when}\ C\ \ \textsc{causes}\ DS}$$

Note that *causes chain* is a special case of *when causes chain*, where C is true.

The following result is particularly useful when $B1 \vee B2$ is $true$ and we can eliminate the $\textsc{when}$ condition.

$$\textit{split by cases}\qquad \frac{M{:}\ A\ \textsc{when}\ B1\ \textsc{causes}\ DS \qquad M{:}\ A\ \textsc{when}\ B2\ \textsc{causes}\ DS}{M{:}\ A\ \textsc{when}\ B1 \vee B2\ \textsc{causes}\ DS}$$

To avoid variables capture, we require that $(vars(M{:}A) \cup vars(B1)) \cap vars(DS) = (vars(M{:}A) \cup vars(B2)) \cap vars(DS)$.

3 Design methodology

The Mob$_{adtl}$ design methodology is based on the notion of refinement [3]. The refinement relation between Mob$_{adtl}$ theories corresponds to logical deduction: we say that a theory $\mathcal{T}'$ refines theory $\mathcal{T}$ if and only if any formula F holding in $\mathcal{T}$ holds also in $\mathcal{T}'$. We call *specification* the set of axioms defining a theory.

We structure theories so that the specification of each class of homogeneous components is kept separate from the specification of other classes, and can be refined independently, in a fully compositional way. However, complete independence would require to fix the interface of the components in each class already at the "zero" refinement step, where requirements are first specified. This means that the refinement steps can extend the interfaces, but not modify them.

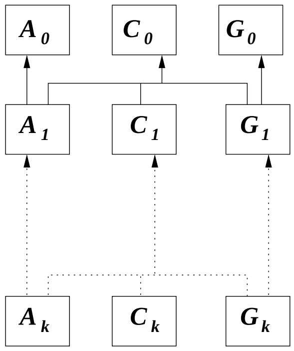
To avoid imposing this constraint to the designer, we adopt a different structuring and refinement strategy. Here, we discuss it in the simple case of two classes of components, $\mathcal{A}$ and $\mathcal{G}$ (i.e. the case of interest for the next sections). At each step we consider both the specification of the two classes, and the specification of the way they coordinate their behaviours. The specification of a class contains only local axioms or axioms relating events and conditions in different components of that class. The coordination axioms relate remote events and conditions of components in different classes. At each refinement step it is possible to extend the set of axioms to specify new properties, or to refine and *distribute* the formulae in the existing theories. For instance formula $A : p$ CAUSES $G : q$ in the coordination theory could be refined by introducing $A : p$ CAUSES $A : r$ in the theory of class $\mathcal{A}$, $A : r$ CAUSES $G : s$ in the coordination theory, and $G : s$ CAUSES $G : q$ in the theory of class $\mathcal{G}$.

The final goal of the refinements is to specify the classes fully and to leave in the coordination theory only the assumptions on the underlying middleware or the properties that will have to be guaranteed by the application context. For instance, a typical middleware property is that point–to–point communication is reliable; assumptions on the context can include human behaviour.

Figure 3 shows the overall refinement/extension structure of the Mob$_{adtl}$ theories. At the zero refinement step we define the global properties of the model: *agents* and *guardians* are the classes of interest, $\mathcal{A}_0$, $\mathcal{G}_0$ are their specification, and $\mathcal{C}_0$ specifies the coordination among them. We define the refinement relation between theories accordingly to the proposed refinement strategy. In the case of agents and guardians, for instance, we ask that at each step theory $\mathcal{A}_i$ refines $\mathcal{A}_{i-1}$, $\mathcal{G}_i$ refines $\mathcal{G}_{i-1}$, and $\mathcal{A}_i \cup \mathcal{C}_i \cup \mathcal{G}_i$ refines $\mathcal{C}_{i-1}$. Indeed, the first refinement step, corresponding to the axiomatization in Section 5, shows a first distribution of the coordination requirements to agents and guardians. The lower levels represent further refinements, like those of Section 6 involving physical mobility.

At each refinement step, we can derive the general properties that hold in the system: we consider the union of all the theories at the step, and reason on them using the theorems of the logic. The derivation of the general properties is needed to prove that the refinement is correct. Moreover, since the step can

Figure 3: Design methodology: a theory hierarchy.



extend the set of axioms in each theory, some new properties can hold in the system because of the new axioms.

3.1 The proof assistant Mark

We developed the proof assistant Mark (Mob$_{adtl}$ Reasoning Kit) to specify and verify properties thus avoiding "by hand" arguments, which usually lead to problems in the development process [13].

Mark consists of several libraries and a back-end theorem prover, Isabelle [25] and works as follows. The user supplies it with the system specification. Then, to prove a property, the user proceeds interactively selecting tactics or theorems among those listed by Mark. When the proof terminates, the user can add it to the Mob$_{adtl}$ proof libraries. If a new tactic has been singled out during the proof, it can be elaborated and inserted in the Mob$_{adtl}$ tactic library.

The advantages of using Mark include: the ability to keep track of the assumptions a property relies on; the ability to handle standard bureaucratic activities (boolean reasoning and standard verification techniques); the ability to support the proving pattern that emerges when composing subsystems.

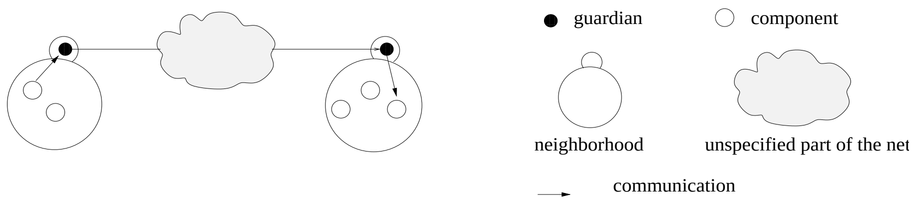
Our experience with Mark so far showed that reasoning about network aware applications is effectively enhanced. In particular, the ability of introducing tactics, i.e. proofs patterns, provides us with a powerful mechanism to manage the complexity of proofs.

4 Requirements of the model

We describe the requirements of the Mob$_{adtl}$ model, and formalize them in the logic. Figure 4 illustrates the entities of the model.

Location. In Mob$_{adtl}$, a network application consists of an immutable net of elaboration nodes: the *neighborhoods*. A neighborhood is a bounded environment where several *components* live: it corresponds to a logical locality and

Figure 4: The Mob$_{adtl}$ model



can represent a unit of physical mobility. Components model units of executing computation. Components can be stationary or mobile: we call *agent* a mobile component. The components can communicate, also remotely, and the agents can move, changing their neighborhood.

Net topology. Nothing is said in the general model about the logical structure of the net. The only constraint is that neighborhoods are not nested in a hierarchy. In the refinements, specific routing policies can implement more structured topologies, e.g. one where a neighborhood plays the role of firewall for a set of other neighborhoods. In this case, all the traffic is routed via the firewall.

Location awareness. Each neighborhood is associated with a particular stationary component, the *guardian*. The knowledge of their own guardian makes components location aware.

Authority. Guardians monitor the components in their neighborhood and limit the resources they can use, playing the role of authorities. Movement requests can be refused by guardians, e.g. for security reasons. More in general, the guardians intercept messages and agents and decide which of them can enter or leave the neighborhood they control. They also provide the routing facilities to forward messages and to handle moving agents: however, we require that the Mob$_{adtl}$ model does not constrain routing policies a priori. In Figure 4 we represent the set of traversed neighborhoods with a cloud. The figure only wants to make clear that the guardians at both ends of a communication are necessarily involved in the message delivery. The same hold for agent migration: movement request must be examined at least by two guardians: those of the origin and target neighborhoods. We do not require that these guardians explicitely accept the request. This is done on purpose, so that any more specific policy (e.g. "agree with silence") can be introduced in the refinements.

Mobility. Agents can move from neighborhood to neighborhood. The model supports strong mobility, since the units of mobility – the agents – are made up of code, data, and control.

Subjective mobility. Mobility is subjective, that is the agents can control their location and cannot be passively moved by other entities.

Communication. Communications are based on asynchronous message passing, and communications between components occur via the guardians.

Naming and profiles. Guardians and components have a unique name. The target of communication and movement requests can be described implicitly by a *profile*, i.e. a set of constraints. The constraints predicate on the attributes of the target, e.g. by explicitely naming the target itself.

As far as mobility is regarded, the use of a profile to specify a target neighborhood enables, for instance, guardians and agents to negotiate the resources needed before committing to a movement. As far as communication is regarded, a profile can describe the receiver component in terms of security constraints and exported services (in case of a message request to a server component). As a further example, profiles permit to smoothly integrate requirements on the quality of service in the early design stages.

4.1 Requirements specification

In this section we formalize the requirements of the model. The specification in Table 3 is structured in theories, according to the design methodology presented above:

- $\mathcal{A}_0$ contains the local axioms of the agents, that constrain mobility to be subjective (S_1–S_4), and agents not to be ubiquitous (U);
- $\mathcal{G}_0$ contains the axiom that fixes the net structure (N), and the general properties of profile satisfaction (Sc).
- $\mathcal{C}_0$ contains the coordination axioms (A_1, A_2, Id_1, Id_2, M–M$''$ and C–C$''$), and the global initial constraints (I_1–I_2).

Here and in the following, we label the safety properties with the label of the liveness property on the same predicates, primed. In the formulae, A, S, R are agents, G, O, T are guardians (we use S and R –sender and the receiver of a message– and O and T –origin and target of a movement– for readability), P is a profile, and D specifies the details of an exception.

Location and net topology. Location awareness is encoded in the component state via predicate *guardedby(G)*, that is, the current guardian identifies the current location. Stationary components are characterized by the invariance of this predicate. Agents are always in some place (a consequence of S_2), and are not ubiquitous (U). Guardians are not guarded, i.e. locations are not nested (N). Initially every guardian knows the agents in its neighborhood, and every agent knows the name of its guardian (I_1, I_2). The same property cannot be stated for the other states of the system, because of the asynchronous setting (we recall that the initial state is the only one where we can assume a global knowledge of the system, i.e. where it is possible to "know" the properties which hold in each component). In the Section 5.4 we will discuss how this mutual knowledge is maintained during system execution.

Table 3: Requirements Specification.

$\mathcal{A}_0$	**U:**	A: INV $\neg$ (guardedby(G) $\wedge$ guardedby(G′) $\wedge$ G$\neq$G′)
	$\mathbf{S}_1$**:**	A: moveTo(P,H) NEEDS guardedby(G) $\wedge$ history(H)
	$\mathbf{S}_2$**:**	A: guardedby(G) $\wedge$ history(H) UNLESS
		guardedby(G′) $\wedge$ history([G|H]) $\wedge$ G$\neq$G′
	$\mathbf{S}_3$**:**	A: history([G|H]) BECAUSE A: moveTo(P,H)
	$\mathbf{S}_4$**:**	INIT A: history([])
$\mathcal{G}_0$	**N:**	G: INV $\neg$ guardedby(G′)
	Sc:	G: INV ((satisfy(TS,P) $\wedge$ TS′ $\subseteq$TS $\wedge$ P′ $\subseteq$P $\rightarrow$ satisfy(TS′,P′))
$\mathcal{C}_0$	$\mathbf{I}_1$**:**	INIT G: guarding(A) $\vee$ A: $\neg$ guardedby(G)
	$\mathbf{I}_2$**:**	INIT A: guardedby(G) $\vee$ G: $\neg$ guarding(A)
	$\mathbf{A}_1$**:**	A: guardedby(T) BECAUSE
		T: mobReq(A,O,P,St) $\wedge$ O: mobReq(A,O,P,St′)
	$\mathbf{A}_2$**:**	R: in(M,S) WHEN guardedby(G) BECAUSE G: msgReq(M,S,P,St) $\wedge$
		G′: (msgReq(M,S,P,St′) $\wedge$ guarding(S))
	M:	A: moveTo(P,H) WHEN guardedby(O) CAUSES
		A: guardedby(T) $\vee$ exc(mob(A,O,P),D)
	M′:	A: guardedby(T) BECAUSE A: moveTo(P,H) $\wedge$
		G: (mobReq(A,O,P,St) $\wedge$ satisfy({T},P))
	M″:	A: exc(mob(A,O,P),D) BECAUSE A: moveTo(P,H) $\wedge$
		G: (mobReq(A,O,P,St) $\wedge$ toBeVetoed(mob(A,O,P),D))
	C:	S: out(M,P) CAUSES R: in(M,S) $\vee$ S: exc((msg(M,S,P)),D)
	C′:	R:in(M,S) BECAUSE
		S: out(M,P) $\wedge$ G: (msgReq(M,S,P,St) $\wedge$ satisfy({R},P))
	C″:	S: exc(msg(M,S,P),D) BECAUSE
		G: (msgReq(M,S,P,St) $\wedge$ toBeVetoed(msg(M,S,P),D)
	$\mathbf{Id}_1$**:**	G: mobReq(A,O,P,St) BECAUSE A: moveTo(P,H)
	$\mathbf{Id}_2$**:**	G: msgReq(M,S,P,St) BECAUSE S: out(M,P)

Authority. Each guardian holds a representation of the neighborhood it controls via predicate *guarding(A)*, that identifies the agents in the neighborhood. Guardians at both ends of a movement (A_1) and of a communication (A_2) necessarily check the corresponding requests. A movement request *mobReq(A,O,P,St)* records the information needed by the guardians involved in the movement process to know who is moving, its origin location, and the requested profile. Similarly, a message request *msgReq(M,S,P,St)* specifies the message body, the sender, and the profile of the receiver. Variable *St* specifies the state of the request. Predicate *in(M,S)* represents a message M received from S.

Predicate *toBeVetoed* represents the decision of a guardian to veto a component action, like a movement, or a communication (M″, C″).

Mobility. The movement of an agent is represented by a change of guardian. Predicate *moveTo(P,H)* represents the commitment of an agent to move to a neighborhood satisfying profile P. H represents the agent's history when the request is made. The history is introduced to formalize subjective mobility, as discussed below.

The commitment of an agent to move can result in an effective movement or in an exception (M). A movement can occur only to a target neighborhood that satisfies the requested profile. Satisfaction is decided by one of the guardians involved in the movement (M$'$); exceptions follow the decision of a guardian along the route of a movement (M$''$).

A movement request is attributed to agent A only if A actually committed to move (Id_1).

Subjective mobility. To express that mobility is subjective, we need to bind each movement to a corresponding request. This can be done by keeping track of the history of any agent, as a list of the locations it has gone through (predicate *history(H)*), and by labelling each movement request with the current history (S_1). We subject a movement to an update of the agent's movements history (S_2). Moreover, the history can be updated only if a corresponding movement request exists (S_3). Initially, the history of an agent is empty (S_4).

Communication. The properties of the communications are similar to those of mobility. Predicate *out(M,P)* represents the intention of a component to send a message M, and P specifies the profile of the receiver. Also in this case the request can result in a successful delivery or in an exception (C). Conversely, a message is received only if actually sent, and if the receiver satisfies the profile (C$'$). Finally, exceptions occur only because of a decision of one of the guardians involved in the delivery (C$''$).

There can be a communication request attributed to a sender S only if S actually committed to communicate (Id_2).

Naming and profiles. Predicate *satisfies* relates names and profiles. It is used to model the choice of a target neighborhood of a movement or the receiver of a communication based on the given profile (M$'$ and C$'$). The axioms do not constrain the choice to be unique. This freedom permits to accommodate, as a refinement, unicast, multicast, and broadcast communications. An example refinement is discussed in Section 5.4. On the other hand, in the case of mobility, the axioms for subjectivity and non ubiquity guarantee uniqueness of the target.

A profile is specified as a set of constraints, and distinct profiles can have overlapping constraints. Predicate *satisfies(TS,P)* says that the neighborhoods in set TS satisfy profile P. The set theoretic nature of P entails that *satisfies(TS,P$'$)* holds for all P' subsets of P. Similarly, *satisfies(TS$'$,P)* holds for all TS' subsets of TS (Sc).

To conclude the section, we briefly comment on the use of vetoes. In the WAN setting we cannot ignore events like failure or unreachability of hosts: we use vetoes to deal with them explicitely. In a uniform manner, we use vetoes also to model mobility and communication failures due to the access policies of the neighborhoods.

5 Reference Model

In this section, we refine the requirements specification to the Mob$_{adtl}$ reference model. In particular, we refine axioms in $\mathcal{C}_0$, since our interest is to make more explicit the responsibilities of agents and guardians. Moreover, one of the goals of the refinement steps is to reduce the coordination theory.

The behaviour of agents, guardians, and their coordination is specified by theories $\mathcal{A}_1$, $\mathcal{G}_1$, and $\mathcal{C}_1$, respectively (Table 4). These theories refine $\mathcal{A}_0$, $\mathcal{G}_0$, and $\mathcal{C}_0$, accordingly to the refinement strategy discussed in Section 3. Theories $\mathcal{A}_1$ and $\mathcal{G}_1$ include $\mathcal{A}_0$ and $\mathcal{G}_0$, respectively, while $\mathcal{C}_1$ does not include $\mathcal{C}_0$: axioms in $\mathcal{C}_1$ are those in Table 4, plus I_1 and I_2 that we do not repeat for simplicity.

5.1 Axioms for mobility

We use variable St in $mobReq(A, O, P, St)$ to model the state of a mobility request. St takes the following values: i, saying that the request is in an initial stage, u, saying that profile P is still unresolved, or $dest(T)$, where T is the name of the target, once selected.

When agent A decides to move, it notifies a mobility request *mobReq(A,O,P,i)* to its guardian O, specifying the profile P of the target neighborhood. This request can either trigger a movement process (Gm$_1$), in which case the agent is declared *moving* (Gm$_4$), or be immediately rejected, if the agent is already moving (Gm$_5$). In the case the request is accepted, it is routed till it is resolved (i.e. a destination neighborhood is selected) and reaches the destination neighborhood, which takes the agent under its own control. It can happen that no neighborhood satisfying P exists or that it exists but it is not reachable. Moreover, any of the involved guardians can veto the request because of some policy. In all these cases an exception is thrown to the agent that returns under the control of the origin guardian. Figure 5 shows some steps of the migration of agent A from O to T.

The coordination theory describes the interactions among agents and guardians, that is the transmission of an agent's request to its guardian (Cm$_1$), and the communication to the agent that the movement took place (Cm$_2$) or that

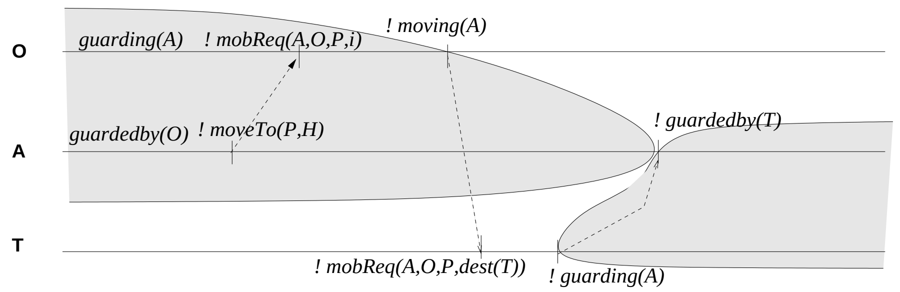


Figure 5: A computation where an agent moves ("!" denotes an event).

an exception occurred (Cm_3).

The system of the guardians is in charge of finding a destination neighborhood that satisfies the requested profile (Gm_2'), and to route the movement request to the destination guardian (Gm_1, Gm_2) that takes the agents under its control (Gm_3). Moreover, any guardian involved in the process can generate a veto (Gm_1, Gm_2, Gm_3). If the veto details are different from *dr*, the agent returns under the control of the origin guardian (Gm_6). Term *dr* stays for double request, and it is used to veto the movement requests of agents that are already moving. In this case, since the first request is still valid, the agent does not necessarily return under the control of the origin guardian.

To intercept double movement requests (Gm_5), predicate *moving(A)* is used: the agents that asked to leave the neighborhood and are not back yet are declared to be moving (Gm_4). We also need to avoid two movement requests from the same agent to be received at the same time (Gm_7). Once an agent entered a neighborhood, the guardian always either guards it or serve a mobility request of its (A_3, A_4), but not both at the same time (A_5), i.e. predicates *guarding* and *moving* are mutually exclusive. This way, guardians keep track of all the components that ever entered the neighborhood: either they are in and guarded, or out, and thus moving.

5.2 Axioms for communications

We let variable St in $msgReq(M, S, P, St)$ take values u saying that the profile P is still unresolved, or $rec(R)$, where R is the name of the selected recipient.

When an agent wants to send a message M, the event communication request *msg(M,S,P,u)* is raised in its current guardian. The request can either succeed, i.e. the message is delivered to an agent satisfying P, or fail, in which case an exception is raised to the sender. If the request is successful, delivery occurs via the guardian of the receiver.

The coordination theory describes the interactions between agents and guardians, that is the transmission of a sender's request to its guardian (Cc_1), and the delivery of the message to the receiver (Cc_2) or the communication of an exception to the sender (Cc_3).

Guardians find a recipient that satisfies the requested profile (Gc_1), and route the message to its destination guardian (Gc_2). Also in the case of a communication, each guardian involved in the process can generate a veto (Gc_2, Gc_3, Gc_4).

5.3 Initial state

The initial state of agents and guardians is constrained not to contain spurious data like exceptions, communication or movements commitments and message or movement requests (Ia, Ig).

Note that when instantiating Mob_{adtl} to specify an actual system, the designer has to fully describe the initial configuration: which are the components, which guardian controls them. Assume components c_1, c_2 and guardians g_1,

Table 4: Axioms of the Mob_{adtl} reference model.

$\mathcal{A}_1$	**Ia:**	INIT A: $\neg$ moveTo(P,H) $\wedge$ $\neg$ out(M,P) $\wedge\neg$ exc(Type,D)
$\mathcal{C}_1$	$\mathbf{Cm}_1$**:**	A: moveTo(P,H) WHEN guardedby(O) CAUSES O: mobReq(A,O,P,i)
	$\mathbf{Cm}_2$**:**	T: guarding(A) CAUSES A: guardedby(T)
	$\mathbf{Cm}_3$**:**	G: toBeVetoed(mob(A,O,P),D) CAUSES A: exc(mob(A,O,P),D)
	$\mathbf{Cm}'_3$**:**	A: exc(mob(A,O,P),D) BECAUSE G: toBeVetoed(mob(A,O,P),D))
	$\mathbf{Cm}'_2$**:**	A: guardedby(T) BECAUSE T: guarding(A)
	$\mathbf{Cm}'_1$**:**	O: mobReq(A,O,P,i) BECAUSE A: moveTo(P) $\wedge$ guardedby(O)
	$\mathbf{Cc}_1$**:**	S: out(M,P) WHEN guardedby(G) CAUSES G: msgReq(M,S,P,u)
	$\mathbf{Cc}_2$**:**	G: msgReq(M,S,P,rec(R)) WHEN guarding(R) CAUSES_C R: in(M,S) $\vee$ G: toBeVetoed(msg(M,S,P),D)
	$\mathbf{Cc}_3$**:**	G: toBeVetoed(msg(M,S,P),D) CAUSES S: exc(msg(M,S,P),D)
	$\mathbf{Cc}'_3$**:**	S: exc(msg(M,S,P),D) BECAUSE G: toBeVetoed(msg(M,S,P),D)
	$\mathbf{Cc}'_2$**:**	R: in(M,S) BECAUSE_C G: (msgReq(M,S,P,rec(R)) $\wedge$ guarding(R))
	$\mathbf{Cc}'_1$**:**	G: msgReq(M,S,P,u) BECAUSE S: (out(M,P) $\wedge$ guardedby(G)) $\vee$ G$'$: (msgReq(M,S,P,u) $\wedge$ G $\neq$ G$'$)
$\mathcal{G}_1$	$\mathbf{Gm}_1$**:**	O: mobReq(A,O,P,i) WHEN guarding(A) CAUSES G: mobReq(A,O,P,u) $\vee$ O: toBeVetoed(mob(A,O,P),D)
	$\mathbf{Gm}_2$**:**	G: mobReq(A,O,P,u) CAUSES T: mobReq(A,O,P,dest(T)) $\vee$ G: toBeVetoed(mob(A,O,P),D)
	$\mathbf{Gm}_3$**:**	T: mobReq(A,O,P,dest(T)) CAUSES T: guarding(A) $\vee$ T: toBeVetoed(mob(A,O,P),D)
	$\mathbf{Gm}_4$**:**	O: mobReq(A,O,P,i) WHEN guarding(A) CAUSES_C O: moving(A)
	$\mathbf{Gm}_5$**:**	O: mobReq(A,O,P,i) WHEN moving(A) CAUSES O: toBeVetoed(mob(A,O,P),dr)
	$\mathbf{Gm}_6$**:**	G: toBeVetoed(mob(A,O,P),D) WHEN D$\neq$dr CAUSES O: guarding(A)
	$\mathbf{Gm}'_1$**:**	G: mobReq(A,O,P,u) BECAUSE O: mobReq(A,O,P,i) $\wedge$ guarding(A)
	$\mathbf{Gm}'_2$**:**	G: mobReq(A,O,P,dest(T)) BECAUSE G$'$: mobReq(A,O,P,u) $\wedge$ G$'$: satisfy({T},P)
	$\mathbf{Gm}'_{3/6}$**:**	G: guarding(A) BECAUSE G: mobReq(A,O,P,dest(G)) $\vee$ G$'$: toBeVetoed(mob(A,G,P),D) $\wedge$ D$\neq$dr
	$\mathbf{Gm}'_{1/2/3}$**:**	G: toBeVetoed(mob(A,O,P),D) BECAUSE G: mobReq(A,O,P,St)
	$\mathbf{Gm}'_4$**:**	O: moving(A) BECAUSE_C O: mobReq(A,O,P,i)
	$\mathbf{Gm}'_5$**:**	G: toBeVetoed(mob(A,O,P),dr) BECAUSE G=O $\wedge$ G: mobReq(A,O,P,i) $\wedge$ moving(A)
	$\mathbf{Gm}_7$**:**	O: INV $\neg$ (mobReq(A,O,P,i) $\wedge$ mobReq(A,O,P$'$,i) $\wedge$ P$\neq$P$'$)
	$\mathbf{Gc}_1$**:**	G: msgReq(M,S,P,u) CAUSES G$'$: msgReq(M,S,P,rec(R)) $\vee$ G: toBeVetoed(msg(M,S,P),D)
	$\mathbf{Gc}_2$**:**	G: msgReq(M,S,P,rec(R)) WHEN $\neg$ guarding(R) CAUSES G$'$: (msgReq(M,S,P,rec(R)) $\wedge$ guarding(R)) $\vee$ G: toBeVetoed(msg(M,S,P),D)
	$\mathbf{Gc}'_{1/2}$**:**	G: msgReq(M,S,P,rec(R)) BECAUSE G$'$: msgReq(M,S,P,u) $\wedge$ satisfy({R},P)
	$\mathbf{Gc}''_{1/2}$**:**	G: toBeVetoed(msg(M,S,P),D) BECAUSE G: msgReq(M,S,P,u) $\vee$ msgReq(M,S,P,rec(R))
	$\mathbf{A}_3$**:**	G: guarding(A) UNLESS moving(A)
	$\mathbf{A}_4$**:**	G: moving(A) UNLESS guarding(A)
	$\mathbf{A}_5$**:**	G: INV $\neg$ (guarding(A) $\wedge$ moving(A))
	Ig:	INIT G: $\neg$ mobReq(A,O,P,St) $\wedge$ $\neg$ msgReq(M,S,P,St) $\wedge$ ...

g_2: the needed assertions are of the kind c_1: guardedby(g_1), c_2: guardedby(g_2), and g_1: guarding(c_1), g_2: guarding(c_2).

5.4 Discussion on the axiomatization

The reference model does not describe how messages or agents reach their target, or why and where an exception is thrown: the axioms only make clear that any of the guardians involved in a communication or a movement, can veto it. Also, the guardians at both ends are necessarily involved, as shown in Figure 4. Which other guardians may be involved, is fixed by the routing and security policies at lower refinement levels. The formulae introduced there detail the guardians, and take care of what happens in between the sender and target guardians, who raises exceptions, and why. In particular, the general axioms do not describe how to deliver a message to a moving target. However, they imply that any refinement either guarantees the *control stabilization property* (i.e. that any message actually reaches its target, if not vetoed) or has ways to stop tracing a target which is moving too fast, and eventually notify a veto (e.g. after a given number of hops between guardians).

We now discuss a point left open in Section 4.1, related with the mutual knowledge between agents and guardians. We already know that initially every guardian knows the agents in its neighborhood, and every agent knows the name of its guardian. S_2 guarantees that no agent can assert to be "lost", and Cm'_2 guarantees that an agent can assert to be guarded by a guardian T only after T has asserted to guard the agent. Dually, as soon as a guardian controls a new agent, it notifies the agent. The agent thus knows its new guardian (Cm_2).

A further point is when an agent has to stop computing after a movement request. The interplay of guarding, moving and guardedby is such that the model does not constrain implementation choices with respect to mobility. E.g. there is no need to stop the execution of an agent immediately after a request to move: the implementation can wait to do so until permission to leave is granted by the local guardian. The model has nothing to say on the local computations in the meantime, it only entails that communication and mobility requests are dealt with differently: communications are regularly served, while mobility attempts are captured as double requests.

Another issue which is left unspecified in the general model is how exceptions are routed: the model only requires that they are delivered. A simple refinement could impose that also exceptions are delivered via the destination guardian. Finally, the general model is insecure, and malicious agents could fake communication and mobility requests wrapping them in messages. Again, the specification of suitable safety properties to avoid these misbehaviours is left to further refinements.

To conclude, we refine the reference model to solve yet another open facet of its, namely the number of recipients of a message. In fact, there are not enough safety conditions to restrict R in axioms C and C$'$ in $\mathcal{C}_0$ to take exactly one value (unicast) or all the values satisfying the profile (multicast). Unicast can be modelled in a message request as a constraint in the specified profile (*uni*). In order to provide a one-to-one communication, we need to guarantee that only one guardian resolves the profile. In this example we let the guardian of the sender to make the choice (Uc_1). Besides, we need to guarantee that the profile resolution gives a unique result (Uc_2 and Uc_3). Multicast is used to deliver a message to all the receivers that satisfy a given profile. To obtain the multicast of a message, the sender adds to the profile for the message request the extra constraint *multi*. To model multicast delivery, we exploit the universal quantification of R in the premises of Mc_1, i.e. all the receivers R that satisfy the constraints in the profile will receive the message if a veto is not raised. Broadcast communication can easily be modelled as a special case of multicast, using the empty profile, so that any recipient will satisfy the profile.

Table 5: Specifying unicast and multicast.

$\mathcal{C}_u$	**Uc$_1$:**	G: msgReq(M,S,P∪{uni},u) BECAUSE S: (out(M,P∪{uni}) ∧ guardedby(G))
$\mathcal{G}_u$	**Uc$_2$:**	G: INV ¬ (msgReq(M,S,P∪{uni}, rec(R)) ∧ msgReq(M,S,P∪{uni}, rec(R′)) ∧ R≠R′)
	Uc$_3$:	G: STABLE msgReq(M,S,P∪{uni}, rec(R))
$\mathcal{G}_b$	**Mc$_1$:**	G: msgReq(M,S,P∪{multi},u) WHEN satisfy({R},P∪{multi}) CAUSES G: msgReq(M,S,P∪{multi}, rec(R)) ∨ G: toBeVetoed(msg(M,S,P∪{multi}),D)

6 Case Study

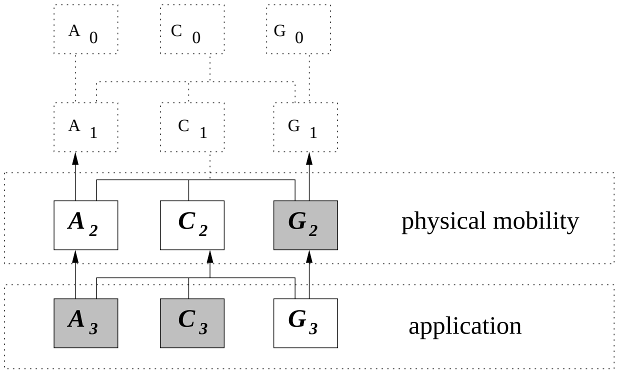
In this section, we validate pragmatically our model and our methodology by specifying a system which addresses the coordination of the activities of a conference program committee (PC). More specifically, we consider the fragment dealing with the paper acceptance phase, during the PC review meeting.

In our near future scenario, the PC members convene to discuss the submitted papers, carrying their own reviews on their portable device, like a PDA, a portable computer, etc. We assume that these devices are equipped with wireless technology, and that during the meeting, an ad hoc network is built.

The purpose of the application is to enable the PC members to share and update their reviews, as well as the paper assessments as they result from the discussions. When an agreement is reached on the acceptance/rejection of a paper, the shared information is updated: reviews and ratings can be adjusted, if needed, but only by the reviewer, who is then guaranteed of the fairness of the process.

In the simplest situation, each member's data are kept on the device. On connection, the local application publishes the reviews and gets access to the information already available on the network. However, we must assume that the PC members can join or leave the meeting dynamically, either because

Figure 6: Theories for the case study: the theories effectively extended are in gray.



they are late or have early planes, or their batteries go down and there are not enough power supplies available in the room. As a design requirement, we require that logical mobility is used to keep the distributed state of the discussion complete, even in this physical mobility scenario: applications migrate towards another device in order to survive to the disconnection of their computational environment from the network. At our level of abstraction we assume that the support finds a destination for the disconnecting applications. The precise definition of the constraints on these migrations is left to the detailed description of the application and of the infrastructure, where a profile can describe the needs of the application, in terms of computational power, operating system, identity of the owner of the target device, etc.

Some assumptions on the behaviour of the PC members are also needed, to orchestrate disconnections and migrations. The essential assumption is that no member will try to migrate his applications while another member is doing the same, in order to avoid cross–migrations and the consequent data losses. Practically, any disconnecting member will announce that he is going to disconnect, and the recipient of his applications will delay his own announcement till all applications have migrated. Besides, a leaving member will wait till his own applications have migrated.

These assumptions appear in the specification as safety axioms that will not be refined into the software; rather, exceptions will be raised when the assumptions are not honored. However, without them nothing interesting can be said of the system and we think it important to include them in the specification.

The next paragraphs sketch how we can work on the case study in Mob_{adtl}. Figure 6 shows the overall structure of this refinement of the Mob_{adtl} model, to deal with scenarios like that described in the case study. The first level of refinement deals with the properties that are of general interest for the kind of application we are considering. The second level of refinement attacks the

Table 6: Axioms for mobile device guardians

$\mathcal{G}_2$	**Gmd$_1$:**	G: mobReq(A,O,P,dest(T)) BECAUSE O: moving(A) $\wedge$ satisfy($\{$T$\}$,P) $\wedge$ reachable($\{$T$\}$)
	Gmd$_2$:	G: toBeVetoed(mob(A,O,P),D) NEEDS guarding(A)
	Gmd$_3$:	G: toBeVetoed(mob(A, O, P),D)) WHEN D$\neq$dr $\wedge$ reachable(RS) $\wedge$ satisfy(TS,P) NEEDS $RS \cap TS = \emptyset$
	R:	G: INV ((reachable(RS) $\wedge$ RS$'$ $\subseteq$RS $\rightarrow$ reachable(RS$'$))
	SR$_1$:	G: new(T,P) WHEN reachable(RS) $\wedge$ satisfy(TS,P) CAUSES_C G: reachable(RS$\cup\{$T$\}$) $\wedge$ satisfy(TS$\cup\{$T$\}$,P)
	SR$_2$:	G: exit(T,P) WHEN reachable(RS) $\wedge$ satisfy(TS,P) CAUSES_C G: reachable(RS$\setminus\{$T$\}$) $\wedge$ satisfy(TS$\setminus\{$T$\}$,P)
	SR$'_{1/2}$:	G: reachable(RS) UNLESS (exit(T,P) $\wedge$ reachable(RS$\setminus\{$T$\}$)) $\vee$ (new(T,P) $\wedge$ reachable(RS$\cup\{$T$\}$))
	SR$''_{1/2}$:	G: satisfy(TS,P) UNLESS (exit(T,P$\cup$P$'$) $\wedge$ satisfy(TS$\setminus\{$T$\}$,P)) $\vee$ (new(T,P$\cup$P$'$) $\wedge$ satisfy(TS$\cup\{$T$\}$,P))
	SR$_3$:	G: new(T,P) UNLESS exit(T,P)

specific application.

The refinement of mobile device guardians towards a particular support for physical mobility is not directly taken into account in the specification of the PC review meeting system. In fact, any refinement (for instance a Bluetooth guardian) of the mobile device guardian presents to the agents the same interface as the more general version. This interface is all we need to describe our system, since we do not want our application to be dependent on the particular infrastructure.

6.1 First level of refinement: physical mobility

We model a device with a neighborhood, and refine guardians to deal with physical mobility. The specification of mobile device guardians includes the ability to detect the relevant events, e.g. that another guardian is approaching, to model the awareness that a neighborhood is reachable. A neighborhood can be declared reachable or not, depending on further design decisions and technologies adopted. For instance, if Bluetooth technology is adopted, reachability would be a consequence of being in the same network or in a network connected by a bridge.

The interesting property of $\mathcal{A}_2 \cup \mathcal{C}_2 \cup \mathcal{G}_2$ is:

> *guardians keep track of reachable neighborhoods and refuse a movement request only if there is no reachable neighborhood satisfying the requested resource profile.*

The only theory effectively refined at this stage is $\mathcal{G}_2$ (see Table 6). Indeed, all the new features are provided by the guardians.

Due to the particular setting, each distributed form of agreement could be unfeasible or could lead to complex algorithmic solutions that are not in the aim of this example. Therefore, we decided to center the resolution process of a mobility request in the origin guardian. This means in particular that the choice of the destination neighborhood is made by the origin guardian on the basis of its knowledge of reachable neighborhoods (Gmd_1). Also, the origin guardian is the only one that can generate mobility vetoes (Gmd_2), which have to be motivated by the lack of a reachable neighborhood satisfying the requested profile (Gmd_3).

Guardians are also in charge of reacting to the entrance and leave of neighborhoods in the net and of keeping their knowledge on reachability and profile satisfaction up to date (R, SR_1, SR_2, $\mathrm{SR}'_{1/2}$, $\mathrm{SR}''_{1/2}$, SR_3). Event *new(T,P)* models the entrance of a neighborhood guarded by T, which satisfies profile P. Similarly, event *exit* models that a neighborhood left.

6.2 Second level of refinement: application facets

Agents are refined to reflect the fact that a migration request is caused by an explicit request of the PC member holding the device. Also, the specific resource profile for migration is fixed. On the guardians side, the main task of the guardian is to book–keep the ad hoc network for the (case study) application. The general property at this level is:

$$\begin{aligned}&\text{G: wantToLeave } \textsc{when} \text{ guarding(A) } \textsc{causes}\\ &\qquad\qquad \text{G: satisfy}(\{\text{T}\},\{\text{pcMeeting}\}) \wedge\\ &\qquad\qquad \text{A: guardedby(T)} \wedge \text{history}([\text{G}|\text{H}])\end{aligned}$$

that is, if the device wants to leave the network, the agent running the application migrates to another device still in the network.
The axioms, presented in Table 7, describe how the guardians manage the disconnection process. The user's decision to leave the PC meeting is reflected in the guardian by the event *wantToLeave*. Predicate *locked(T)* denotes that T is the selected guardian. Dually, predicate *leaving(G)* models, in the guardian T of the receiving device, the fact that it has been selected.

The assumptions on the behaviour of the PC member we made at the beginning of the section, are formalized by the axioms Ass_1–Ass_3. A PC member leaves the meeting if there is still a meeting, i.e. if there are still some device at the meeting (Ass_1). Moreover, we assume that the member in charge of hosting the applications of the leaving member will wait for them and will not leave the network until all the applications have completed the migration (Ass_2). Finally migration requests are not sent to leaving guardians (Ass_3). These are examples of safety properties that will not be led into the implementation by the refinement process since they must be guaranteed by the users and not by the application

The other axioms describe the behaviour of the support: when guardian G decides to leave, the support chooses the device that accepts the applications

Table 7: Axioms for the conference program committee application

$\mathcal{A}_3$	**Apc$_1$:** A: moveTo(P,H) NEEDS P={pcMeeting}
$\mathcal{C}_3$	**Cpc$_1$:** G: locked(T) WHEN guarding(A) CAUSES A: moveTo({pcMeeting},H)
	Cpc$'_1$: A: moveTo({pcMeeting},H) WHEN guardedby(G) BECAUSE G: locked(T)
$\mathcal{G}_3$	**Ass$_1$:** G: wantToLeave NEEDS new(T,{pcMeeting}) $\wedge$ $\neg$ exit(T,{pcMeeting})
	Ass$_2$: G: INV wantToLeave $\leftrightarrow$ $\neg$ leaving(G$'$)
	Ass$_3$: G: leaving(G$'$) NEEDS $\neg$ wantToLeave
	Gpc$_1$: G: wantToLeave CAUSES T: leaving(G) $\wedge$ G: locked(T)
	Gpc$'_1$: G: locked(T) BECAUSE T: leaving(G)
	Gpc$''_1$: T: leaving(G) BECAUSE G: wantToLeave
	Gpc$_2$: T: exit(G, P) WHEN leaving(G) CAUSES_C T: $\neg$ leaving(G)
	Gpc$'_2$: T: leaving(G) UNLESS exit(G, P)
	Gpc$_3$: INIT T: $\neg$ leaving(G)
	Gpc$_4$: INIT G: $\neg$ locked(T)
	Gpc$_5$: G: mobReq(A,O,P,i) NEEDS $\neg$ moving(A)
	Gpc$_6$: G: exit(G$'$,{pcMeeting}) BECAUSE G$'$: population({ })
	Gpc$_7$: G: INV (locked(T) $\rightarrow$ reachable({T}))
	Gpc$_8$: G: INV (locked(T) $\leftrightarrow$ satisfy({T},{pcMeeting}))

of G (Gpc$_1$); assumption Ass$_3$ guarantees that G is different from T. Then, G communicates to its applications where to migrate (Cpc$_1$); the lock in the receiving devices is removed after the leaving member actually left (Gpc$_2$, Gpc$'_2$); locked devices are reachable (Gpc$_7$) and belong to PC members (Gpc$_8$).

For the sake of simplicity, we restrict mobility in this example to be the consequence of a migration request, this means that applications move only towards another PC member device (Apc$_1$), and we assume that there will be not double requests (Gpc$_5$).

Finally, we subject the leave of a device to the fact that its population is empty (Gpc$_6$). This happens when all applications have completed the migration, as described in Table 8. Predicate *population* is used to model the set of the components which are guarded or on leave but not yet *moved*.

To complete the refinement step, theory $\mathcal{G}_3$ must include also the refinement of axiom Gm$'_{1/2/3}$ of theory $\mathcal{G}_1$ into the following one that prevents the raise of any veto:

G: toBeVetoed(mob(A,O,pcMeeting(G$'$),D) NEEDS false

Consequently, we can derive a stronger version Gm$_1$–Gm$_3$, where the vetoes disappear.

Table 8: Axioms for the conference program committee application: population

$\mathcal{C}_3$	**Cpo$_1$:** A: guardedby(T) WHEN history([G|H]) CAUSES G: moved(A)
$\mathcal{G}_3$	**Gpo$_1$:** G: moved(A) WHEN population(Po) CAUSES_C G: population(Po$\setminus\{$A$\}$)
	Gpo$_2$: G: guarding(A) WHEN population(Po) CAUSES_C G: population(Po$\cup\{$A$\}$)
	Gpo$'_{1/2}$: G: population(Po) BECAUSE_C G: (population(Po$\cup\{$A$\}$) $\wedge$ moved(A)) $\vee$ G: (population(Po$\setminus\{$A$\}$) $\wedge$ guarding(A))
	Gpo$_3$: G: INV (moved(A) $\rightarrow$ moving(A))
	Gpo$_4$: INIT G: population({ })

7 Related Works

The problem of supporting WAN computing from specification and architectural design to implementation and certification is at the edge of most of the current research in the field of Software Engineering, since traditional approaches (e.g. client-server architecture) are no any longer sufficient to meet the new demands. This section compares the Mob$_{adtl}$ approach with some of the most relevant approaches that have been proposed to address the problem of providing effective models for WAN computing. We will focus our attention on models that support explicitly network awareness, mobility, and coordination.

Mobile Unity [20] is an extension of the Unity logic and proof techniques to deal with mobility. Mobile Unity is based on the *transient* shared memory model: a variable shared by a pair of processes can be updated in an independent and non–coherent way when the processes are in distinguished localities. Network awareness is modelled explicitly as a distinguished variable that belongs to the state of a mobile component. Modifying the value of this variable corresponds to change the location of the component. Specifications include a *coordination* section, which defines how coherence is achieved among the values of a shared variable when the processes move to the same or to *close* locations. Mobile Unity has been refined and exploited in several works. In particular, it provided the foundations of a *middleware*, LIME, supporting mobility and coordination abstractions that simplify the development of WAN applications and ad hoc networks [26]. Applications of Mobile Unity have also faced the issues of algorithm development with mobility in mind [24].

Differently from Mob$_{adtl}$, the communication model of Mobile Unity is based on shared variables and imposes strong assumptions on the behaviour of the mobile units. In particular, the update of shared variables requires complex interactions among the involved units, which must remain colocated for all the time needed for the value propagation. The assumption of colocation is not always easy to guarantee. An asynchronous communication model better fits the setting of wide area networks.

The research activities in the field of coordination languages and models have improved the formal understanding of dynamically adaptable mobile components. In particular, the model introduced in [2] provides a discipline for orchestrating WEB services [17]. Their approach is based on the idea of separating WEB service providers from *contracts – connectors* mechanisms. WEB service coordination is regulated by the behaviour of contracts which act as guardians of WEB services.

Work on the theories of process calculi has developed models and verification techniques for mobility, distributions and security. We mention the Distributed Join-calculus [14], Klaim [10], the Distributed π-calculus [18], the Ambient calculus [6], the Seal calculus [30]. All these foundational calculi encompass a notion of location to reflect the idea of network awareness and a notion of migrating components. The Ambient calculus provides a rather rich model. An ambient is a bounded environment where computational components and ambients live. Hence, ambients provide the context of components including resources as well as other (sub-)contexts. Moreover, ambients are also the *units* of coordination as the boundary of an ambient can be reconfigured dynamically in order to manage addition/removal of services and subambients. However this generality is a source of interferences and unpredictable behaviours: imagine, for instance, the situation of an ambient wanting to move to a another ambient and somebody else wanting to move it elsewhere (ambients' mobility is not strictly subjective).

Several dialects of Ambients have been proposed to avoid these interference problems and to provide a finer grained control over ambients coordination. The *safe ambients* calculus [19] introduces *coactions*, imposing all parties involved in a coordination activity to agree on it. Safe Ambients permits defining powerful and accurate type systems for controlling mobility [19, 4].

The *seal calculus* [30] is a variant of the Ambients calculus where ambient boundaries cannot disappear (the *open* capability is dropped), ambient communication can take also place across ambient boundaries (between parent and children) via explicit communication channels. In [7], *interface types* are introduced to describe the coordination requests that an ambient may accept from its surrounding environment, e.g. one can declare the type of the ambients that can enter or exit it. The fine grain control of ambient coordination provides a better support for the specification of security policies. For instance, in the Boxed-Ambient Calculus [5], a type system ensures definition of policies for *mandatory access control*. The type system of Controlled Ambients (an extension of safe Ambients) [29] allows one to analyze resource usage.

There are several similarities between Mob_{adtl} and these ambient-like process calculi. The action–coaction mechanism or Seal communication patterns roughly corresponds to the interaction protocol between an agent (making a request to move) and the guardians involved in the movement. Indeed, guardians have been also explicitly introduced in Ambients: in [11] ambients are equipped with guardians which monitor the activities of the components inside an ambient and their interactions with the external operational environment. Hence, each guardian defines and enforces a local security policy and guardian coordination fixes the permissions for ambient movements.

Other approaches control coordination and reconfiguration of components by exploiting a notion similar to the idea of guardian. *Execution monitoring* [27] is a general class of enforcement mechanisms that work by monitoring execution steps of a target application and by terminating execution that is about to violate the security policy being enforced. Execution monitoring relies on the idea of security automata: the monitored application is executed in tandem with a simulation of the automaton defining the wanted security policy. *Wrappers*, namely code that encapsulates possibly untrusted components and can intercept the communications between each such component and the rest of the system, are significant examples (see, e.g., [15, 28]) of execution monitoring mechanisms. In [15], a *wrapper definition language* for composing COTS software components is defined along with a framework for wrappers management. In [28], box-π, a process calculus obtained as a minimal extension of the π-calculus with encapsulation, is defined and used for specifying wrappers that rigorously implement security policies.

Our proposal is closely related to the idea of execution monitoring, and guardians can be seen as high level security automata, that permit to specify and prove properties of the allowed interactions.

To conclude this discussion, we observe that the Mob_{adtl} approach is rather flexible. Indeed, the refinement steps of our model can provide more elaborated and specific models than the reference model given by the axioms. For instance, we can encode hierarchical models, including nested neighborhoods, by defining appropriate routing policies that refine the axiomatization of communications and migrations.

8 Conclusions

In this paper we presented Mob_{adtl}, a high level declarative model for the design of mobile and ad hoc computing. Mob_{adtl} offers full support to specify and reason about network awareness, mobility and coordination. Moreover, Mob_{adtl} provides a formal methodology to drive the design of applications based on the notion of structured refinements. Hence, Mob_{adtl} has to be properly understood as a basic model on top of which application specific concepts can be inserted via suitable refinements steps.

Future work include the use of the new logic YALL [23] in Mob_{adtl}, which offers more powerful mechanisms to reason on the specifications.

We are also planning to extend the methodology to include data refinement. It is frequent that in an initial refinement step we use a predicate p with arity n, and that in subsequent refinements we need some more parameters. For instance, parameter St in *mobReq* and *msgReq* should not appear in the requirements specification, while its need is evident in the axiomatization of the reference model. We are currently working at the formal results that will validate this kind of refinements. The idea is to use abstract interpretation techniques to relate predicates with different arities.

References

[1] Mark: Mob$_{adtl}$ reasoning kit. at www.di.unipi.it/~monta/SALADIN.

[2] L.F. Andrade, J.L. Fiadeiro, J. Gouveia, G. Koutsoukos, and M. Wermelinger. Coordination for orchestration. In F. Arbab and C.L. Talcott, editors, *5th Int. Conference on Coordination Models and Languages (COORDINATION 2002)*, volume 2315 of *Lecture Notes in Computer Science*, pages 5–13. Springer-Verlag, 2002.

[3] R.J.R. Back and J. von Wright. *Refinement Calculus. A Systematic Introduction.* Graduate texts in computer science. Springer-Verlag, 1998.

[4] M. Bugliesi and G. Castagna. Secure safe ambients. In *Proceedings of POPL '01*, pages 222–235. ACM Press, 2001.

[5] M. Bugliesi, G. Castagna, and S. Crafa. Reasoning about security in mobile ambients. In *Concur 2001*, volume 2154 of *Lecture Notes in Computer Science*, pages 102–120. Springer-Verlag, 2001.

[6] L. Cardelli and A. Gordon. Mobile ambients. *Theoretical Computer Science*, 240(1), 2000.

[7] G. Castagna, G. Ghelli, and F. Zappa Nardelli. Typing mobility in the seal calculus. In *Concur 2001*, volume 2154 of *Lecture Notes in Computer Science*, pages 82–101. Springer-Verlag, 2001.

[8] K.M. Chandy and J. Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading Mass., 1988.

[9] G. Cugola, C. Ghezzi, G.P. Picco, and G. Vigna. Analyzing Mobile Code Languages. In *Mobile Object Systems: Towards the Programmable Internet*, volume 1222 of *Lecture Notes in Computer Science*, pages 93–110. Springer-Verlag, Apr 1997.

[10] R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: a Kernel Language for Agents Interaction and Mobility. *IEEE Transactions on Software Engineering*, 24(5):315–330, 1998.

[11] G. Ferrari, E. Moggi, and R. Pugliese. Guardians for ambients-based monitoring. In *FWAN: Foundations of Wide Area Network Computing*, ENTCS. Elsevier Science B. V., 2002.

[12] G. Ferrari, C. Montangero, L. Semini, and S. Semprini. Multiple Security Policies in Mob$_{adtl}$. In P. Degano, editor, *Proc. Workshop on Issues in the Theory of Security (WITS'00)*, Geneva, 7,8 July 2000.

[13] G. Ferrari, C. Montangero, L. Semini, and S. Semprini. Mark, a reasoning kit for mobility. *Automated Software Engineering*, 9(2):137–150, Apr 2002.

[14] C. Fournet, G. Gonthier, J.-J. Lévy, L. Maranget, and D. Rémy. A Calculus of Mobile Agents. In U. Montanari and V. Sassone, editors, *Proc. CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 406–421. Springer-Verlag, Aug. 1996.

[15] T. Fraser, L. Badger, and M. Feldman. Hardening COTS software with generic software wrappers. In *IEEE Symposium on Security and Privacy*, pages 2–16, 1999.

[16] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Transactions on Software Engineering*, 24(5):342–361, 1998.

[17] IBM Software Group. Web services conceptual architecture. In *IBM White Papers*, 2000.

[18] M. Hennessy and J. Riely. Distributed Processes and Location Failures. *Theoretical Computer Science*. To appear.

[19] F. Levi and D. Sangiorgi. Controlling Interference in Ambients. In *Proc. 27^{th} Annual ACM Symp. on Principles of Programming Languages*, pages 352–264, Boston, Jan 00. ACM Press.

[20] P.J. McCann and G.-C.Roman. Compositional Programming Abstractions for Mobile Computing. *IEEE Transactions on Software Engineering*, 24(2):97–110, Feb. 1998.

[21] C. Montangero and L. Semini. Composing Specifications for Coordination. In P. Ciancarini and A. Wolf, editors, *Proc. 3nd Int. Conf. on Coordination Models and Languages*, volume 1594 of *Lecture Notes in Computer Science*, pages 118–133, Amsterdam, April 1999. Springer-Verlag.

[22] C. Montangero and L. Semini. Software specification and design: from formal methods to standard middleware. In *Proc. 6th ERCIM Inter. Workshop on Formal Methods for Industrial Critical Systems (FMICS'01)*, 2001.

[23] C. Montangero and L. Semini. Distributed states logic. In 9^{th} *International Symposium on Temporal Representation and Reasoning*, Manchester, UK, July 2002. IEEE CS Press.

[24] A.L. Murphy. *Enabling the Rapid Development of Dependable Applications in the Mobile Environment*. PhD thesis, Washington University, 2000.

[25] L. Paulson and T. Nipkow. Isabelle. www.cl.cam.ac.uk/Research/HVG/ Isabelle/.

[26] G.P. Picco, A.L. Murphy, and G.-C. Roman. LIME: Linda meets mobility. In D. Garlan and J. Kramer, editors, *Proceedings of the the* 21^{st} *International Conference on Software Engineering (ICSE'99)*, pages 368–377. ACM Press, May 1999.

[27] F. B. Schneider. Enforceable security policies. *ACM Transation on Information and System Security*, 2(4), 2000. Also appeared as Technical Report TR99-1759, Department of Computer Science, Cornell University, Jul 1999.

[28] P. Sewell and J. Vitek. Secure composition of untrusted code: Box pi, wrappers, and causality types. *To appear in Journal of Computer Security*, 2002. Full version as Technical Report 478, Computer Laboratory, University of Cambridge, November 1999.

[29] D. Teller, P. Zimmer, and D. Hirschkoff. Using ambients to control resources. In *Concur 2002 (to appear)*, Lecture Notes in Computer Science. Springer-Verlag, 2002.

[30] J. Vitek and G. Castagna. Towards a Calculus of Secure Mobile Computations. In *Proc. Workshop on Internet Programming Languages*, volume 1686 of *Lecture Notes in Computer Science*, Chicago, Illinois, May 1998. Springer-Verlag.