A Language for Modeling Network Availability

Petre, Luigia; Sere, Kaisa; Walden, Marina

Published: 01/01/2006

Document Version
Final published version

Luigia Petre | Kaisa Sere | Marina Waldén

A Language for Modeling Network Availability

Turku Centre *for* Computer Science

TUCS Technical Report
No 752, March 2006

A Language for Modeling Network Availability

Luigia Petre
Department of Information Technologies
Åbo Akademi University, FIN-20520 Turku, Finland.
luigia.petre@abo.fi
Kaisa Sere
Department of Information Technologies
Åbo Akademi University, FIN-20520 Turku, Finland.
kaisa.sere@abo.fi
Marina Waldén
Department of Information Technologies
Åbo Akademi University, FIN-20520 Turku, Finland.
marina.walden@abo.fi

TUCS Technical Report
No 752, March 2006

Abstract

Computer networks have become ubiquitous in our society and thus, the various types of resources hosted by them are becoming increasingly important. In this paper we study the resource availability in networks by defining a dedicated middleware language. This language is a conservative extension of the action system formalism, a general state-based approach to modeling and analyzing distributed systems. Our language formally treats aspects such as replicated and homonym resources, their mobility, as well as node failure and maintenance in networks. The middleware approach motivates the separation of the views and formalisms used by the various user types such as the network user, the application developer, and the network manager.

Keywords: Formal modeling, middleware, network availability, resource management

TUCS Laboratory
Distributed System Design Laboratory

1 Introduction

The networks have become an ubiquitous component of our society. Network applications range from e-commerce and Internet banking to digital TV, video-on-demand, and network games. Envisioned applications link the existing networks with more mundane appliances already containing embedded software, such as microwave ovens, refrigerators, VCRs, or the house clocks. With these applications it would be possible to check online the contents of the fridge, schedule the recording of a TV programme from work, or have the clocks automatically switched to the daylight saving time [17]. As users of such a network-oriented world, we would like to access all the applications we need at any time, without having to know about the network functioning mechanisms.

Meeting such a goal requests a high degree of reliance on the correct functioning and availability of the networks. Numerous formal frameworks have been developed to address the former issue, namely to specify a network-oriented application and then analyze its properties. Examples of such frameworks are CSP [7, 8] and CCS [10], UNITY [6], Object-Z [16], and action systems [1]. To partially address the latter issue of network availability, some formalisms define concepts such as *locations* and *mobility*: π-calculus [12, 11], Ambient calculus [5, 4], Mobile UNITY [14, 9], and topological action systems [13]. These formalisms are mostly targeted to application developers who need the concepts of location and mobility in their specifications. The proper network availability is to be treated at a more specialized lower level. Hence, we need to separately analyze the user requirements, using for this any formal framework dedicated to network-aware applications. These requirements may need to be expressed more precisely in terms of locations and mobility using a properly equipped formal framework. Then, at an even lower layer, we can analyze the capabilities of the network in handling the functioning of various applications. This layer is commonly referred to as the *middleware* layer.

In this paper we present a middleware language for supporting resource-centric computing. Our proposed language is based on topological action systems introduced in [13] as a framework extending the action system formalism conservatively towards location-aware computing. Here we build on this work by providing an approach dedicated to the network resources and nodes. A resource can be a *data-oriented repository*, a *piece of code* or a complete application, the latter referred to as a *computation unit*. Our language handles replicated and homonym resources, their mobility, as well as node failure and maintenance:

- The *replication* mechanism provides and maintains copies of resources at various locations in the network. If some nodes are temporarily down, due to failures or maintenance procedures, the resources stored at these nodes may still be accessible if they have replicas on other functional nodes.

- *Homonym resources* have the same name but possibly different meanings.

It is quite probable that such resources are defined in large networks, hence we need to have the mechanisms to handle them.

- *Resource mobility* refers to the change of location in the network and, thus, provides for resource availability in various parts of the network. This feature can be used by both the network manager and the application developer.

- Due to the increasing network traffic, the nodes need to be verified or *maintained* periodically: before shutting down a node for maintenance, the status of its resources is saved and when the node is verified, the status of its resources is restored. Nevertheless, the nodes can unexpectedly *fail*, losing all their resources; in this case there is no previous saving of the resource status.

Due to its applicability, our proposed language promotes the separation of concerns in network-oriented applications. Thus, the user requirements can be specified and analyzed in the context of action systems. Topological action systems as defined in [13] can be used to specify and develop applications that are location- and mobility-dependent. The management of the network with all its mechanisms for increased resource availability can then be specified and analyzed in the context of topological action systems as developed in this paper.

The paper is structured as follows. In Section 2 we present the language of topological action systems that can be used by a typical application developer; in Section 3 we show some properties of this language that make the location-transparent and the location-aware specifications compatible. The following sections are dedicated to the middleware language: Section 4 introduces the replicated resources, Section 5 the homonym resources, and Section 6 the resource mobility. In Section 7 we discuss the failure and maintenance of network nodes. Conclusions and related work are presented in Section 8.

2 Topological action systems

A topological action system is defined based on a connected graph. The nodes of this graph model the nodes of a network where computation can take place or where data can reside, i.e., the set of possible *locations* for resources. Let therefore $G = (V, E)$ be a *connected* graph, where V is the non-empty set of nodes in the graph and E the set of edges.

A topological action system consists of a finite set of *actions* that can evaluate and modify a finite set of *variables*. The values of the variables form the *state* of the system. In the following, we first concentrate on the variables and actions and then define formally the topological action system.

Variables Let Var be a finite set of *variable names*. We define a *variable* of a topological action system to be a quadruple (v, loc, Val, val) where $v \in Var$, $loc \subseteq V$, Val is a nonempty set of *values*, and $val \in Val$. The first field v denotes the name of the variable, the second (loc) its location in the network (V, E), the third (Val) the variable type, and the fourth (val) the current value of the variable. To avoid working with this quadruple in specifications, a few shorthand notations are introduced. We express the location of a variable called v by the expression $v.loc$ and the names of the variables located at a location α by the expression $\alpha.var$. The value of a variable called v is given by the expression $v.val$ and the type of a variable called v by the expression $v.type$. When we are interested only in the location of a variable, we can express this by $v@\alpha$, where $\alpha \in v.loc$ ($\alpha \in V$) or $v@\Gamma$, where $\Gamma \subseteq v.loc$ ($\Gamma \subseteq V$). When $|v.loc| > 1$ we say that the variable is *replicated*. We assume that the type of a variable is unchangeable.

Actions Let Act be a finite set of *action names*, distinct from Var. We define an *action* of a topological action system to be a triple (a, loc, A) where $a \in Act$ is an informal, optional name for the action, $loc \subseteq V$ is its location in the network (V, E), and A is its body, i.e., a statement that can model evaluation and updates of the variables. A set of shorthand notations is introduced to avoid working with such triples in specifications. We express the location of an action (a, loc, A) by the expression $A.loc$ and the bodies of the actions located at a location α by the expression $\alpha.action$. When we are interested only in the location of an action, this is given by $A@\alpha$, where $\alpha \in A.loc$ ($\alpha \in V$) or $A@\Gamma$, where $\Gamma \subseteq A.loc$ ($\Gamma \subseteq V$). When $|A.loc| > 1$ we say that the action is *replicated*. The name of an action is unchangeable. The body A of an action a is described by the following grammar:

$$A \quad ::= \quad abort \mid skip \mid v.val :=\!\!= e \mid b \rightarrow A \mid A \,;\, A \mid [\!]_{i \in I} A \mid \mathsf{if}\ b\ \mathsf{then}\ A\ \mathsf{else}\ A\ \mathsf{fi} \tag{1}$$

Here (v, loc, Val, val) is a variable so that $v \in Var$, $e \in Val$, b is a predicate, and I an index set. Intuitively, $abort$ is the action which always deadlocks, $skip$ is the stuttering action, $v.val :=\!\!= e$ is an assignment, $b \rightarrow A$ is a guarded action, which can be executed only when b evaluates to true, $A_1 \,;\, A_2$ is the sequential composition of two actions A_1 and A_2, $[\!]_I A_i$ is the non-deterministic choice among actions A_i, $i \in I$, and $\mathsf{if}\ b\ \mathsf{then}\ A_1\ \mathsf{else}\ A_2\ \mathsf{fi}$ is the conditional composition of two actions A_1 and A_2.

Action semantics The semantics of the action bodies in a topological action system is expressed with the *weakest precondition predicate transformer* (wp) [3, 2]. Assume an action body A and a postcondition q for A, i.e., a predicate describing the state after the execution of A. A and q are then mapped into the weakest predicate $wp(A, q)$ describing the state before A was executed, so that A establishes q. For the action bodies described by the grammar (1) we give below their

corresponding wp expressions:

$$\begin{array}{lcl}
wp(abort, q) & \widehat{=} & false \\
wp(skip, q) & \widehat{=} & q \\
wp(v.val := e, q) & \widehat{=} & q[e/v.val] \\
wp(b \rightarrow A, q) & \widehat{=} & (b \Rightarrow wp(A, q)) \\
wp(A_1 \ ; \ A_2, q) & \widehat{=} & wp(A_1, wp(A_2, q)) \\
wp([\!]_I A_i, q) & \widehat{=} & \forall i \in I \cdot wp(A_i, q) \\
wp(\textsf{if } b \textsf{ then } A_1 \textsf{ else } A_2 \textsf{ fi}, q) & \widehat{=} & (b \Rightarrow wp(A_1, q) \wedge \neg b \Rightarrow wp(A_2, q)).
\end{array}$$

We say that an action behaves miraculously when it establishes the postcondition $false$, which models an aborting state. The *guard condition* $g(A)$ defined as $g(A) \widehat{=} \neg wp(A, false)$ gives those states in which an action behaves non-miraculously. We assume here that wp is *strict*, i.e., $wp(A, false) = false$, hence $g(A) = true$, for every non-guarded action (see (1)). Strictness is not a restriction for wp, since any action A' such that $g(A') = b, b \neq true$, can be rewritten as $A' = b \rightarrow A$, where $g(A) = true$. The actions A that respect the condition $g(A) = true$ are called *always enabled*.

Topological action systems The computation unit in the network (V, E) is modeled by a *topological action system*, defined in the following form:

$$\begin{array}{lcll}
\mathcal{A} & \widehat{=} & |[& \mathbf{exp} \quad y; \\
 & & & \mathbf{var} \quad x; \\
 & & & \mathbf{imp} \quad z; \\
 & & & \mathbf{do} \ [\!]_{i \in I} \ A_i \ \mathbf{od} \\
 & &]| &
\end{array} \tag{2}$$

The first three sections are for variable declaration and usage, while the last describes the computation involved in $\mathcal{A}$, when I is finite. We assume that x, y and z are lists of variables whose names are pairwise disjoint, i.e., the name of a variable is unique in a topological action system.

The **exp** section describes the *exported* variables y declared by $\mathcal{A}$, $y = \{(y_l, y_l.loc, y_l.type, y_l^0)\}_{l \in L}$, where L is a finite set of indices. They can be used within $\mathcal{A}$, as well as within other topological action systems that import them. Initially, they are assigned the values y_l^0 and are located at $y_l.loc$. If the initialization is missing, arbitrary values from the type sets $y_l.type$ are assigned as initial values, while a default location $\{\lambda\}, \lambda \notin V$ is assigned as initial location. As the exported variables can be imported by other systems, their names are unchangeable.

The **var** section describes the *local* variables x declared by $\mathcal{A}$, $x = \{(x_j, x_j.loc, x_j.type, x_j^0)\}_{j \in J}$, where J is a finite set of indices. They can be used only within $\mathcal{A}$. Initially they are assigned the values x_j^0 and locations $x_j.loc$, or, if the initialization is missing, some arbitrary values from their type sets and $\{\lambda\}$ for location, respectively. As these variables are local to $\mathcal{A}$, their names can be changed. This

change has to respect the requirement of unique names for variables in a topological action system and has to be propagated in all the action bodies that access the respective local variables.

The **imp** section describes the *imported* variables $z, z = \{(z_k, \alpha_k, T_k)\}_{k \in K}$, where K is a finite set of indices. These variables are specified by name (z_k), desired location of import (α_k), and desired import type (T_k). They are used in $\mathcal{A}$ and are declared as exported in other topological action systems, thus, modeling the communication between topological action systems. The imported and the exported variables form the *global* variables of $\mathcal{A}$. The desired location of import α_k is denoted by $z_k.iloc$ and the desired type of import T_k by $z_k.itype$. The locations α_k can also be left unspecified, denoting the need of $\mathcal{A}$ to use variables with predefined names and types, independently of their location. In this case we write $z_k.iloc = \emptyset$. As the imported variables refer to exported variables of other topological action system, also their names are unchangeable.

The **do**...**od** section describes the computation involved in $\mathcal{A}$, modeled by a non-deterministic choice between actions with bodies A_i described by the grammar (1). If some of these actions are replicated, $|A_i.loc| > 1$, then A_i in the **do**...**od** section stands for $A_i@\rho_{i_1} \, [\!] \, A_i@\rho_{i_2} \, [\!] \, ... \, [\!] \, A_i@\rho_{i_{h_i}}$, where $|A_i.loc| = h_i$ and $A_i.loc = \{\rho_{i_1}, \rho_{i_2}, ..., \rho_{i_{h_i}}\}$. Hence, in order to describe the computation of an action in a topological action system we refer to actions $(a, \rho, A), \rho \in V$.

Assume that the names of all the variables accessed by an action (a, ρ, A) are in the set vA and the names of the accessed imported variables are in the set $iA, iA \subseteq vA$. It can often be the case that the variables vA and the action a are located in different nodes of the network and hence, the accessibility of the variables from the action is not necessarily guaranteed. To model the network accessibility of an action $(a, \rho, A), \rho \in V$, we define a function of the action and its location, called *cell*: $cell(A, \rho) \subseteq V$. The cell comprises the set of accessible locations for each action at a certain location.

To model that the action $(a, \rho, A), \rho \in V$ and its accessed variables are appropriately located, we define a predicate called *location guard*, denoted $lg(A@\rho)$:

$$\begin{aligned} lg(A@\rho) \;\; \widehat{=} \;\; & \forall v \in vA \cdot (\exists \alpha \in cell(A, \rho) \cdot v \in \alpha.var) \wedge \\ & (v \in iA \wedge v.iloc \neq \emptyset \Rightarrow \alpha = v.iloc) \end{aligned} \tag{3}$$

Before executing the action, the location guard verifies that, for each variable called $v, v \in vA$, there is a location α in the cell of the action that contains a variable with this name. If an imported variable $v \in iA$ is specified together with its desired location of import ($v.iloc \neq \emptyset$), then the location α coincides with the desired location of import $v.iloc$.

Enabledness The $guard$ of the action (a, ρ, A) is defined as

$$gd(A@\rho) \widehat{=} lg(A@\rho) \wedge g(A),$$

where $g(A)$ is the guard condition. An action (a, ρ, A) of a topological action system is said to be *enabled*, if its guard $gd(A@\rho)$ evaluates to true. An action can be chosen for execution only if it is enabled.

The topological action system $\mathcal{A}$ in formula (2) thus models computation via the action $[\!]_{i\in I}(A_i@\rho_{i_1} \,[\!]\, A_i@\rho_{i_2} \,[\!]\, ... \,[\!]\, A_i@\rho_{i_{h_i}})$. Hence, $\mathcal{A}$ is a set of actions with bodies A_i and locations ρ_{i_j}, $i \in I, j \in \{1, 2, ..., h_i\}$, operating on local and global variables. First, the local and exported variables whose values form the state of $\mathcal{A}$ are initialized. Then, repeatedly, enabled actions at various locations in V are non-deterministically chosen and executed, typically updating the state of $\mathcal{A}$. Actions that do not access each other's variables and are enabled at the same time can be executed in parallel. This is possible because their sequential execution in any order has the same result and the actions are taken to be atomic. Atomicity means that, if an enabled action with body A is chosen for execution, then it is executed to completion without any interference from the other actions of the system. The computation terminates if no action is enabled, otherwise it continues infinitely.

3 Scalability, granularity, and compatibility

Parallel composition The topological action system is defined as the basic computation unit. Yet, in order to model complex systems we need to be able to compose such units. This operation is described using the *parallel composition* operator.

Consider the topological action systems $\mathcal{A}$ and $\mathcal{B}$ below. We *assume* that the local variables of these systems have distinct names: $\{x_{j_1}\}_{j_1\in J_1} \cap \{w_{j_2}\}_{j_2\in J_2} = \emptyset$. If this is not the case, we can always change the name of a local variable to meet this requirement. The exported variables declared in $\mathcal{A}$ and $\mathcal{B}$ are *required* to have distinct names, $\{y_{l_1}\}_{l_1\in L_1} \cap \{v_{l_2}\}_{l_2\in L_2} = \emptyset$:

$$
\begin{array}{rcl}
\mathcal{A} & = & |[\ \mathbf{exp} \quad y; \\
& & \ \ \mathbf{var} \quad x; \\
& & \ \ \mathbf{imp} \quad z; \\
& & \ \ \mathbf{do} \ [\!]_{i_1\in I_1} A_{i_1} \ \mathbf{od} \\
& &]|
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{B} & = & |[\ \mathbf{exp} \quad v; \\
& & \ \ \mathbf{var} \quad w; \\
& & \ \ \mathbf{imp} \quad t; \\
& & \ \ \mathbf{do} \ [\!]_{i_2\in I_2} B_{i_2} \ \mathbf{od} \\
& &]|
\end{array}
$$

The *parallel composition* $\mathcal{A} \,||\, \mathcal{B}$ of $\mathcal{A}$ and $\mathcal{B}$ has the following form:

$$
\begin{array}{rcl}
\mathcal{A} \,||\, \mathcal{B} & \widehat{=} & |[\ \mathbf{exp} \quad u; \\
& & \ \ \mathbf{var} \quad s; \\
& & \ \ \mathbf{imp} \quad r; \\
& & \ \ \mathbf{do} \ A \,[\!]\, B \ \mathbf{od} \\
& &]|
\end{array}
\tag{4}
$$

where $u = y \cup v$, $s = x \cup w$ and r is the list obtained by concatenating the lists of imported variables z, t and removing those variables that are in the list u:

$r = (z \cup t) \setminus u$. Also, $A = \|_{i_1 \in I_1} A_{i_1}$ and $B = \|_{i_2 \in I_2} B_{i_2}$. The initial values and locations of the variables, as well as the actions in $\mathcal{A} \,||\, \mathcal{B}$ consist of the initial values, locations, and the actions of the original systems, respectively. The well-definedness of $\mathcal{A}||\mathcal{B}$ is ensured by the fact that all its variables have unique names: the exported variables of $\mathcal{A}$ and $\mathcal{B}$ are required to be distinct, the local variables of $\mathcal{A}$ and $\mathcal{B}$ are assumed to be distinct, and moreover, these local variables can always be renamed in order not to collide with the exported variables. The binary parallel composition operator '||' is associative and commutative and thus extends naturally to the parallel composition of a finite set of topological action systems.

Scalability Based on the parallel composition operator, our topological approach is enabled to *scale up*, i.e., to model larger systems. We can thus specify the computation of entire networks, including the location of their various resources. Based on the same operator, more flexibility of this approach has been demonstrated [13]. Thus we can decompose a topological action system into parallel 'smaller' units, so small that they encompass only a variable or an action. If the topological action system $\mathcal{A}$ in (2) has only two exported variables, two local variables, one imported variable and two actions ($I = \{1, 2\}$), then we rewrite it as follows:

$$\begin{array}{lcl} \mathcal{A} & = & |[\mathbf{exp}\ y_1]|\ ||\ |[\mathbf{exp}\ y_2]|\ ||\ |[\mathbf{var}\ x_1]|\ ||\ |[\mathbf{var}\ x_2]|\ ||\ \mathcal{C}_1\ \ ||\ \ \mathcal{C}_2 \\ \mathcal{C}_1 & = & |[\mathbf{imp}\ z, y_1, x_1\ ;\ \mathbf{do}\ A_1\ \mathbf{od}]| \\ \mathcal{C}_2 & = & |[\mathbf{imp}\ z, y_2, x_2\ ;\ \mathbf{do}\ A_2\ \mathbf{od}]| \end{array} \tag{5}$$

where z, y_1, x_1 are the names of the variables imported by $\mathcal{C}_1$, and z, y_2, x_2 are the names of the variables imported by $\mathcal{C}_2$. Hence, we have a collection of topological action systems, each describing only one resource and running in parallel with each other.

Such flexibility provided by the topological action systems approach allows the computation unit to *scale down* to fine grains of data and code resources. This is important because it provides a unique notation for modeling different kinds of resources in a network. Thus, a topological action system denotes not only an entire computation unit that acts via actions over a set of variables, but also a data repository (a variable or a set of variables) or mere code resources (an action or a set of actions).

Location of systems Topological action systems of the form $\mathcal{A}_1 = |[\mathbf{exp}\ y@\Gamma_1]|$, $\mathcal{A}_2 = |[\mathbf{var}\ y@\Gamma_2]|$, and $\mathcal{A}_3 = |[\mathbf{imp}\ z\ ;\mathbf{do}\ A@\Gamma_3\ \mathbf{od}]|$ can be seen as taking the location of their declared entities: $\mathcal{A}_1@\Gamma_1, \mathcal{A}_2@\Gamma_2, \mathcal{A}_3@\Gamma_3$. This rises the question of defining locations also for the entire computation unit, i.e., the topological action system, not only for base resources such as variables and actions. The concept of location for topological action systems is applicable when mobility and other features of various resources are modeled, as shown in the following sections.

If all the components of a topological action system have the same location, then this location is propagated to the topological action system. In case the locations differ, the topological action system gets the default location $\{\lambda\}$. Thus, we define the location of a topological action system $\mathcal{A}$,

$$\begin{array}{rcll} \mathcal{A} & = & [\![& \mathbf{exp} \quad y_1@\Phi_1, \cdots, y_n@\Phi_n; \\ & & & \mathbf{var} \quad x_1@\Psi_1, \cdots, x_m@\Psi_m; \\ & & & \mathbf{imp} \quad z; \\ & & & \mathbf{do} \ [\!]_{i \in I} \, A_i@\Delta_i \ \mathbf{od} \\ & &]\!] & \end{array}$$

as:

$$\mathcal{A}.loc \mathrel{\widehat{=}} \begin{cases} \Phi_i, & \Phi_i = \Psi_j = \Delta_k, \ \forall i, j, k \\ \{\lambda\}, & otherwise. \end{cases} \tag{6}$$

We also use the notation $\mathcal{A}@\alpha$ or $\mathcal{A}@\Gamma$ for expressing that $\alpha \in \mathcal{A}.loc$ or $\Gamma \subseteq \mathcal{A}.loc$, respectively. The reverse relation, of a topological action system propagating its location to its components holds in the following form. If $\mathcal{A}.loc = \{\alpha\}, \alpha \in V$ or $\mathcal{A}.loc = \Gamma, |\Gamma| > 1$ then the components of $\mathcal{A}$ all have the same location $\mathcal{A}.loc$. Yet, if $\mathcal{A}.loc = \{\lambda\}$ then we cannot say anything about the locations of the topological action system components.

The default location models the location of a server from which the entire network is accessible and vice versa, any resource located at λ is accessible to every other node in the network V. More precisely, an action with body A so that $A.loc = \{\lambda\}$ has $cell(A, \lambda) = V$, hence $lg(A@\lambda) = true$. This action is therefore enabled whenever the guard condition $g(A)$ holds. We also assume that $\lambda \in cell(A, \rho),$ for every action (a, ρ, A). Thus, if an action requires some variable located at $\{\lambda\}$, then the variable is accessible. This mechanism of default location is intended as a compatibility means with the more traditional local area networks where there is no replication and no significant mobility, and hence no location information is necessary. Such a local area network can be seen as located at $\{\lambda\}$, and so, is location-transparent.

4 Replicated Resources

One aspect of our middleware language is the definition and maintenance of replicated resources. The replication mechanism is intended for optimal availability of resources and, thus, we do not replicate resources located at $\{\lambda\}$. Such resources are anyway accessible to the whole network as explained above. In the following, we study the replication of each resource type in turn: first we consider variables, then actions, and then topological action systems.

4.1 Variables

In the previous sections we have seen the impact of variable names, in the definition of action enabledness as well as in the well-definedness of parallel composition. We now study variables having the same name.

A *replicated variable* is by definition a variable whose location has more than one element. More precisely, the replicas of a variable called v and located at $\Gamma = \{\alpha_1, \alpha_2, \alpha_3, ...\}$ have the same name, type, and value, but different locations excluding $\{\lambda\}$ $(v@\alpha_1, v@\alpha_2, v@\alpha_3, ...)$. Moreover, it makes no sense to have different replicas of the same resource at the same location.

Accessing replicated variables depends on whether the access is (only) for reading or for (reading and) updating the variable. If a variable is only accessed for reading, then no special rule is needed, hence the location guard in (3) is used. We can reinterpret this rule as follows: For any variable called $v, v \in vA$, we choose a location of one of its replicas so that this location is accessible to the action. If a variable is accessed also for updating, then we need a special rule: if a replicated variable is updated, then all its replicas have to be updated simultaneously to the same value. We model this case in two steps. First, we enforce a more restrictive form of the location guard than the one in (3):

$$\begin{array}{rcl} lg(A@\rho) & \widehat{=} & \forall v \in vA \cdot (\exists \alpha \in cell(A, \rho) \cdot v \in \alpha.var) \wedge \\ & & (|\mathbf{v.loc}| > \mathbf{1} \Rightarrow \mathbf{v.loc} \subseteq \mathbf{cell(A}, \rho)) \wedge \\ & & (v \in iA \wedge v.iloc \neq \emptyset \Rightarrow \alpha = v.iloc) \end{array} \quad (7)$$

This restricted form ensures that we can access all the replicas of a variable called v that needs to be updated. Second, if the action with body A is enabled and chosen for execution, then every assignment to such a variable is replaced by the sequential composition of assignments to all its replicas. As an example, the action in the topological action system

$$|[\mathbf{exp}\ y@\{\alpha, \beta\}\ ;\ \mathbf{do}\ (y.val \neq 5 \rightarrow y.val :=_{\cdot} 5)@\rho\ \mathbf{od}]|$$

first checks the guard condition $y.val \neq 5$, then the location guard $(\exists \delta \in cell(y.val \neq 5 \rightarrow y.val :=_{\cdot} 5, \rho) \cdot y \in \delta.var) \wedge \{\alpha, \beta\} \subseteq cell(y.val \neq 5 \rightarrow y.val :=_{\cdot} 5, \rho)$. If both conditions evaluate to $true$, then both $y@\alpha$ and $y@\beta$ are updated to the value 5. The atomicity property for actions ensures that other computations will not access y until all its copies are updated.

Creating replicas There are two ways to create replicas for a variable. We can either declare the variable as replicated or we can update its location via actions during the execution of the topological action system. In the latter case, consider that we have a variable called v. We can create another replica of this variable at the location $\alpha \neq \lambda$ using a special $copy$ action:

$$\begin{array}{lcl} A & ::= & ... \mid copy(v, \alpha), \\ copy(v, \alpha) & \widehat{=} & v.loc \neq \{\lambda\} \rightarrow v.loc :=_{\cdot} v.loc \cup \{\alpha\} \end{array} \quad (8)$$

This action is semantically sound, its wp expression having the following form:

$$wp(copy(v,\alpha),q) \;=\; (v.loc \neq \{\lambda\} \Rightarrow q[(v.loc \cup \{\alpha\})/v.loc])$$

Its guard condition is $v.loc \neq \{\lambda\}$. In order to create replicas at α, the action $copy$ needs to have this location accessible to its cell. Hence, the location guard is

$$\begin{array}{rcl} lg(copy(v,\alpha)@\rho) & = & \exists \beta \in cell(copy(v,\alpha),\rho) \cdot v \in \beta.var \\ & & \wedge \alpha \in \mathbf{cell}(\mathbf{copy}(\mathbf{v},\alpha),\rho) \end{array}$$

Removing replicas The reverse of the $copy$ operation is that of removing replicas of a variable called v:

$$\begin{array}{lcl} A & ::= & ... \mid remove(v,\alpha), \\ remove(v,\alpha) & \widehat{=} & \mathsf{if}\, v.loc \setminus \{\alpha\} \neq \emptyset\ \mathsf{then}\ v.loc := v.loc \setminus \{\alpha\} \\ & & \mathsf{else}\ v.loc := \{\lambda\}\ \mathsf{fi} \end{array} \qquad (9)$$

This action is semantically sound having the following wp expression:

$$\begin{array}{rcl} wp(remove(v,\alpha),q) & = & (v.loc \setminus \{\alpha\} \neq \emptyset \Rightarrow q[(v.loc \setminus \{\alpha\})/v.loc]) \wedge \\ & & (v.loc \setminus \{\alpha\} = \emptyset \Rightarrow q[\{\lambda\}/v.loc]), \end{array}$$

while its guard condition is $true$ and its location guard is similar to the location guard of $copy(v,\alpha)$. If $v.loc = \{\alpha\}$ and we still want to remove this 'replica', then the copy from α is indeed removed, but the variable is saved at the default location $\{\lambda\}$.

4.2 Actions

Actions have a different replication pattern compared to variables. They model *active* code resources, i.e., code which executes itself following its own semantic rules. We observe that executing $A \parallel A$ is equivalent to executing A, except that some of the nodes containing A may be unavailable. Hence, by having replicas of an action executed in parallel at different locations, we increase the enabledness of the code modeled by this action, i.e. we ensure a better availability.

A replicated action is by definition an action whose location has more than one element. More precisely, the replicas of an action (a, loc, A) so that $A.loc = \{\rho_1,\rho_2,\rho_3,...\}$ have the same name and body, but different locations excluding $\{\lambda\}$: ($A@\rho_1$, $A@\rho_2$, $A@\rho_3$, ...). It makes no sense to have more than one replica of an action at the same location.

Creating replicas There are two ways to create replicas for an action. We can either declare the action as replicated or we can update its location via actions during the execution of the topological action system. In the latter case, consider

that we have an action with body A. We can create another replica of this action at the location $\alpha \neq \lambda$ using a special $copy$ action:

$$\begin{array}{lcl} A & ::= & ... \mid copy(A, \alpha), \\ copy(A, \alpha) & \widehat{=} & A.loc \neq \{\lambda\} \rightarrow A.loc :=_{=} A.loc \cup \{\alpha\} \end{array} \tag{10}$$

This action is semantically sound, its wp expression having the following form:

$$wp(copy(A, \alpha), q) \;=\; (A.loc \neq \{\lambda\} \Rightarrow q[(A.loc \cup \{\alpha\})/A.loc])$$

Its guard condition is $A.loc \neq \{\lambda\}$ and its location guard is

$$\begin{array}{rcl} lg(copy(A, \alpha)@\rho) & = & \exists \beta \in cell(copy(A, \alpha), \rho) \cdot A \in \beta.action \\ & & \wedge \alpha \in \mathbf{cell}(\mathbf{copy}(\mathbf{A}, \alpha), \rho) \end{array}$$

We note that the action body needs to be appropriately located in order for the $copy$ operation to succeed. In this case, the action to be copied behaves more like a data resource.

Removing replicas The reverse of the $copy$ operation is that of removing replicas of an action with body A:

$$\begin{array}{lcl} A & ::= & ... \mid remove(A, \alpha), \\ remove(A, \alpha) & \widehat{=} & \mathsf{if}\, A.loc \setminus \{\alpha\} \neq \emptyset\, \mathsf{then}\, A.loc :=_{=} A.loc \setminus \{\alpha\} \\ & & \mathsf{else}\, A.loc :=_{=} \{\lambda\}\, \mathsf{fi} \end{array} \tag{11}$$

This action is semantically sound having the following wp expression:

$$\begin{array}{rcl} wp(remove(A, \alpha), q) & = & (A.loc \setminus \{\alpha\} \neq \emptyset \Rightarrow q[(A.loc \setminus \{\alpha\})/A.loc]) \wedge \\ & & (A.loc \setminus \{\alpha\} = \emptyset \Rightarrow q[\{\lambda\}/A.loc]), \end{array}$$

while its guard condition is $true$ and its location guard is similar to the location guard of $copy(A, \alpha)$. If $A.loc = \{\alpha\}$ and we still want to remove this 'replica', then the copy from α is indeed removed, but the action is saved at the default location $\{\lambda\}$.

4.3 Topological action systems

Consider a topological action system $\mathcal{A}$ in (2) so that $\mathcal{A}.loc \neq \{\lambda\}$. $\mathcal{A}$ is called *replicated* if $\mathcal{A}.loc > 1$. This means that all the variables and actions of $\mathcal{A}$ are replicated at $\mathcal{A}.loc$. Creating and removing replicas of $\mathcal{A}$ is based on creating and removing replicas for all the variables and actions of $\mathcal{A}$. We formally extend the

action grammar (11) with two more actions:

$$
\begin{array}{lcl}
A & ::= & \ldots \mid copy(\alpha) \mid remove(\alpha), \\
copy(\alpha) & \widehat{=} & \mathcal{A}.loc \neq \{\lambda\} \rightarrow \\
 & & \quad \forall l \in L \cdot copy(y_l, \alpha); \\
 & & \quad \forall j \in J \cdot copy(x_j, \alpha); \\
 & & \quad \forall i \in I \cdot copy(A_i, \alpha) \\
remove(\alpha) & \widehat{=} & \mathsf{if}\, \mathcal{A}.loc \setminus \{\alpha\} \neq \emptyset\ \mathsf{then} \\
 & & \quad \forall l \in L \cdot remove(y_l, \alpha); \\
 & & \quad \forall j \in J \cdot remove(x_j, \alpha); \\
 & & \quad \forall i \in I \cdot remove(A_i, \alpha) \\
 & & \mathsf{else}\ \mathcal{A}.loc := \{\lambda\}\ \mathsf{fi}
\end{array}
\tag{12}
$$

These actions are semantically sound, having tedious but obvious wp expressions. The actions $copy(\alpha)$ and $remove(\alpha)$ refer to the topological action system they are specified in, hence, we cannot manipulate other systems based on these actions. Namely, computation units can only duplicate themselves and similarly for reducing their number of replicas.

5 Homonym variables

Another capability of our middleware language is the treatment of homonym variables. The more general case of such variables having the same name, but possibly different types and values, is slightly different with respect to replication. These resources can be declared in different topological action systems: $\mathcal{T}_1 = |[\mathbf{exp}\ (z_k, \{\alpha\}, T_1, a)]|$ and $\mathcal{T}_2 = |[\mathbf{exp}\ (z_k, \{\beta\}, T_2, b)]|$, where z_k is their common name, $\{\alpha\}$ and $\{\beta\}$ their distinct locations, T_1 and T_2 their types, and a and b their values. The types T_1 and T_2 can be identical or not. The difference with respect to a replicated variable located at $\{\alpha, \beta\}$ is that each homonym variable has its own update history, i.e., their updates are independent of each other.

Clearly, the systems where homonym variables are declared cannot compose in parallel with each other. However, another topological action system importing the variable $(z_k, \alpha_k, z_k.type)$ can compose with either of $\mathcal{T}_1$ or $\mathcal{T}_2$. In this case, for importing a variable both its name and its type have to match with its specification $(z_k, \alpha_k, z_k.type)$. Hence, the last conjunction in the location guard formula (3) $(v \in iA \wedge v.iloc \neq \emptyset \Rightarrow \alpha = v.iloc)$ becomes $(v \in iA \Rightarrow v.type = v.itype \wedge (v.iloc \neq \emptyset \Rightarrow \alpha = v.iloc))$. In this way the rightly typed variable is imported.

Besides the above modification, we need to ensure that at a certain location there is only one variable with a certain name $v, \forall v \in V$. This integrity condition is modeled by a function that records, for every location in V and every variable name in Var, the number of variables having that name and located there: $\forall v \in Var, \forall \alpha \in V \cdot \alpha.no(v) \in \{0, 1\}$. Thus, $\alpha.no(v) = 1$ means that a variable called v is located or has a replica at α and $\alpha.no(v) = 0$ means that there is no variable

called v located or with a replica at α. When we specify our systems we need to ensure the well-definedness of this function: a replicated variable v has only one copy at every $\alpha \in v.loc$ and there are no homonym variables called v with common locations, $\forall v \in Var$.

Hence, the guards of the actions that could modify the values of the function during execution have to prohibit this. The action $copy(v, \alpha)$ thus becomes: $copy(v, \alpha) \widehat{=} v.loc \neq \{\lambda\} \wedge \alpha.\mathbf{no}(\mathbf{v}) = \mathbf{0} \rightarrow v.loc :\!= v.loc \cup \{\alpha\}$.

6 Mobility of resources

In addition to replication, mobility is a central feature in network computations and in our middleware language. We can model data, code, as well as computation unit mobility using the topological action system framework. Hence, we obtain a model for *resource mobility*.

We start by extending the grammar (12) with the following actions:

$$
\begin{array}{lcl}
A & ::= & ... \mid move(v, \alpha_0, \alpha) \mid move(A, \alpha_0, \alpha) \mid move(\alpha_0, \alpha), \\
 & & \alpha_0, \alpha \in V \\
move(v, \alpha_0, \alpha) & \widehat{=} & \alpha_0 \in v.loc \rightarrow v.loc :\!= v.loc \setminus \{\alpha_0\} \cup \{\alpha\} \\
move(A, \alpha_0, \alpha) & \widehat{=} & \alpha_0 \in A.loc \rightarrow A.loc :\!= A.loc \setminus \{\alpha_0\} \cup \{\alpha\} \\
move(\alpha_0, \alpha) & \widehat{=} & \alpha_0 \in \mathcal{A}.loc \rightarrow \\
 & & \forall l \in L \cdot move(y_l, \alpha_0, \alpha); \\
 & & \forall j \in J \cdot move(x_j, \alpha_0, \alpha); \\
 & & \forall i \in I \cdot move(A_i, \alpha_0, \alpha)
\end{array}
\tag{13}
$$

Here v is the name of a variable, A is the body of an action, and $\mathcal{A}$ is a topological action system. The $move$ actions model the movement of resources (variable, action, topological action system) from the initial location α_0 to a location α in the network. These actions are guarded by the condition that the initial location of the resource contains the location α_0. The $move$ actions are semantically sound; the first two have the following wp expressions and guard conditions:

$$
\begin{array}{lcl}
wp(move(v, \alpha_0, \alpha), q) & = & (\alpha_0 \in v.loc \Rightarrow q[(v.loc \setminus \{\alpha_0\} \cup \{\alpha\})/v.loc]) \\
wp(move(A, \alpha_0, \alpha), q) & = & (\alpha_0 \in A.loc \Rightarrow q[(A.loc \setminus \{\alpha_0\} \cup \{\alpha\})/A.loc]) \\
g(move(v, \alpha_0, \alpha)) & = & \alpha_0 \in v.loc \\
g(move(A, \alpha_0, \alpha)) & = & \alpha_0 \in A.loc
\end{array}
$$

Moving $\mathcal{A}$ from α_0 to α is based on moving all the variables and actions of $\mathcal{A}$. The corresponding action is also semantically sound, its wp-expression being based on the above given expressions. The action $move(\alpha_0, \alpha)$ refers to the topological action system it is specified in. Hence, computation units can only move themselves; we cannot manipulate other systems based on this action.

We can note the role of the guard condition $\alpha_0 \in r.loc$ (see (13)) of a $move$ action for the replicated resource r, $|r.loc| > 1$. This condition ensures that only

the copy located at α_0 is moved to α while the rest of the copies of r do not change their location.

Guards In order for the $move$ actions to be executable, the target location α needs to be accessible. Furthermore, the variable called v and the action with body A have to be available in the cell of the $move$ actions. Since these resources are located at α_0 (ensured by the guard condition) the location α_0 also needs to be available to the $move$ actions. We have the following location guards:

$$\begin{array}{lcl} lg(move(v,\alpha_0,\alpha),\rho) & = & \{\alpha_0,\alpha\} \subseteq cell(move(v,\alpha_0,\alpha),\rho) \\ lg(move(A,\alpha_0,\alpha),\rho) & = & \{\alpha_0,\alpha\} \subseteq cell(move(A,\alpha_0,\alpha),\rho) \\ lg(move(\alpha_0,\alpha),\rho) & = & \{\alpha_0,\alpha\} \subseteq cell(move(\alpha_0,\alpha),\rho) \wedge \\ & & \bigwedge_{l\in L} lg(move(y_l,\alpha_0,\alpha),\rho_l) \wedge \\ & & \bigwedge_{j\in J} lg(move(x_j,\alpha_0,\alpha),\rho_j) \wedge \\ & & \bigwedge_{i\in I} lg(move(A_i,\alpha_0,\alpha),\rho_i) \end{array} \quad (14)$$

The location guard of the action $move(\alpha_0,\alpha)$ also contains the location guards of the $move$ actions for the variables and actions of $\mathcal{A}$.

Similarly as for the $copy$ action, we strengthen the guard condition of the $move$ action so that $\forall v \in Var, \forall \alpha \in V$ the function $\alpha.no(v)$ remains well-defined. The action $move(v,\alpha_0,\alpha)$ thus becomes:

$$move(v,\alpha_0,\alpha) \widehat{=} \alpha_0 \in v.loc \wedge \alpha.\mathbf{no}(\mathbf{v}) = \mathbf{0} \rightarrow v.loc := v.loc \setminus \{\alpha_0\} \cup \{\alpha\}$$

7 Node failure and maintenance

The last step in the definition of our middleware language is to handle the failure and maintenance of nodes. We do this in two steps. First, consider a partition of the network nodes V into active and inactive nodes, $V = V_{act} \cup V_{inact}$ so that $V_{act} \cap V_{inact} = \emptyset$. We extend the grammar (9) with the following actions:

$$\begin{array}{lcl} A & ::= & ... \mid fail(\alpha) \mid begin\_maint(\alpha) \mid end\_maint(\alpha), \\ fail(\alpha) & \widehat{=} & \alpha \in V_{act} \rightarrow V_{act} := V_{act} \setminus \{\alpha\};\ V_{inact} := V_{inact} \cup \{\alpha\} \\ begin\_maint(\alpha) & \widehat{=} & \alpha \in V_{act} \rightarrow V_{act} := V_{act} \setminus \{\alpha\};\ V_{inact} := V_{inact} \cup \{\alpha\} \\ end\_maint(\alpha) & \widehat{=} & \alpha \in V_{inact} \rightarrow V_{act} := V_{act} \cup \{\alpha\};\ V_{inact} := V_{inact} \setminus \{\alpha\} \end{array} \quad (15)$$

We assume here that $\alpha \in V, \alpha \neq \lambda$, i.e., the server does not fail and does not need to be maintained. Intuitively, we want to interpret the above actions as follows. The action $fail(\alpha)$ models the unexpected failure of node α, hence the node changes its status from active to inactive. The action $begin\_maint(\alpha)$ models that the node α is marked as inactive since some maintenance procedures will be performed for it. The action $end\_maint(\alpha)$ models that the node α has returned to normal operation after certain maintenance procedures have been performed for it.

The form of these actions is rather incomplete in (15), hence we proceed to the second step. We have to model what happens with the resources located at nodes α that fail or need maintenance, so we slightly refine the actions above. For maintenance, we would like to 'save' the resource information so that we can restore it when the node functions normally again. For this we further partition V_{inact} into nodes under maintenance and failed nodes, $V_{inact} = V_{maint} \cup V_{fail}$ so that $V_{maint} \cap V_{fail} = \emptyset$. The new forms for the maintenance actions are then:

$$\begin{array}{lcl} begin\_maint(\alpha) & \widehat{=} & \alpha \in V_{act} \rightarrow V_{act} :\!= V_{act} \setminus \{\alpha\};\ V_{maint} :\!= V_{maint} \cup \{\alpha\} \\ end\_maint(\alpha) & \widehat{=} & \alpha \in V_{maint} \rightarrow V_{act} :\!= V_{act} \cup \{\alpha\};\ V_{maint} :\!= V_{maint} \setminus \{\alpha\} \end{array}$$

If α becomes inactive due to failure, then all the variables and actions located there disappear; hence, the action $fail(\alpha)$ becomes:

$$\begin{array}{lcll} fail(\alpha) & \widehat{=} & \alpha \in V_{act} \rightarrow & V_{act} :\!= V_{act} \setminus \{\alpha\};\ V_{fail} :\!= V_{fail} \cup \{\alpha\}; \\ & & & \forall v \in \alpha.var \cdot remove(v, \alpha); \\ & & & \forall A \in \alpha.action \cdot remove(A, \alpha) \end{array}$$

We observe that such a fail operation is relatively safe: in the worst case – when a resource is only located at the failing node α – it saves a copy of each such resource at the default location $\{\lambda\}$.

All the above actions are simple guarded assignments with obvious wp expressions and guards. Based on this model of failure and maintenance of nodes, we need to ensure that ordinary actions execute only when their accessed variables are located at active locations. This is solved by enforcing a small modification to the location guard. Thus, instead of $(\forall v \in vA \cdot (\exists \alpha \in cell(A, \rho) \cdot v \in \alpha.var) \wedge ...)$ we require $(\forall v \in vA \cdot (\exists \alpha \in cell(A, \rho) \cap \mathbf{V_{act}} \cdot v \in \alpha.var) \wedge ...)$.

Due to the possibilities of node failure and maintenance, the location guard $lg(A@\rho)$ of an action (a, ρ, A) also needs to verify the extra condition $\rho \in \mathbf{V_{act}}$, and so $lg(A@\rho) = \rho \in \mathbf{V_{act}} \wedge (\forall v \in vA \cdot ...)$.

Moreover, the actions that handle locations need to ensure their active status. It is meaningless to create replicas or move resources to locations that are inactive or to remove replicas from inactive locations, since the resources there might already have been removed by the safe failure operation. Hence, the condition $\alpha \in V_{act}$ has to be conjuncted with the location guards of the actions $copy$, $move$, and $remove$.

8 Conclusions

Topological action systems build conservatively on the earlier action system formalism; this is an advantage, since we can use their properties and mechanisms without having to reinvent them. Independently of this inherent link between the base and the inheriting formalism, we need to use each for what they are better equipped. Action systems have been used before to demonstrate their value in

modeling and analyzing location-transparent applications [3, 2, 15]. Topological action systems as shown here concentrate on the management of the network in applications that are location-aware. Hence, we have presented a precise language that provides all the needed tools for a middleware network-programmer.

The same principle of inheritance is the basis of π-calculus and Mobile UNITY. CCS systems are represented by parallel compositions of processes that may synchronize and communicate via preestablished links. π-calculus is an extension of CCS based on the idea that not only values are exchanged over the links, but links themselves can be communicated. As a result, π-calculus is one of the first modeling approaches for mobility: the fact that a process varies its links is seen as if the process moves, in order to attain the respective links.

UNITY [6] is a state-based formalism for modeling distributed systems. Modularity is modeled by allowing several programs per UNITY system. In this case, homonym variables declared in distinct programs are shared and the statements can be executed synchronously or asynchronously. Mobility and computations based on it are modeled in Mobile UNITY. To coordinate the programs, the conditions of sharing certain variables or of synchronizing certain statements are given. As programs can move and these conditions can contain location-based predicates, a dynamic style of computing as well as transient variable sharing or synchronization are modeled.

As a different approach defined from scratch, Ambient Calculus is dedicated to modeling computations over the Internet. An ambient is an administrative domain where computation happens. It can also have sub-ambients or be contained in a parent ambient. Sibling ambients can (locally) communicate if they possess certain complementary capabilities, similarly to CCS and π-calculus communication primitives. Mobility is defined as the need to (stepwise) cross barriers and is the most important feature of an ambient.

We end up by noticing that the main contribution of our middleware language consists in enforcing conditions (guards) that influence the enabledness of otherwise always enabled actions. The fact that our non-guarded actions are always enabled simplifies the understanding of the overall action enabledness. Since our setting is that of networks, the enforced conditions model the availability of the network. By carefully defining the status of each network node, we motivate all the guards defined in this paper.

References

[1] R. J. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In *Proceedings of the 2nd ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing*, pp. 131-142, 1983.

[2] R. J. Back and K. Sere. From Action Systems to Modular Systems. In *Software - Concepts and Tools*, Vol. 17, pp. 26-39, Springer-Verlag, 1996.

[3] R. J. Back and K. Sere. Superposition Refinement of Reactive Systems. In *Formal Aspects of Computing*, Vol. 8, No. 3, pp. 324-346, Springer-Verlag, 1996.

[4] L. Cardelli. Mobility and Security. In *F. L. Bauer and R. Steinbrüggen (eds), Proceedings of the NATO Advanced Study Institute on Foundations of Secure Computation*, NATO Science Series, IOS Press, pp. 3-37, 2000.

[5] L. Cardelli. Abstractions for Mobile Computation. In *J. Vitek and C. Jensen (eds). Secure Internet Programming: Security Issues for Mobile and Distributed Objects*. Lecture Notes in Computer Science, Vol. 1603, pp. 51-94, Springer-Verlag, 1999.

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

[7] C.A.R. Hoare. Communicating Sequential Processes. In *Communications of the ACM*, Vol. 21, No. 8, pp. 666-677, 1978.

[8] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, 1985.

[9] C. Mascolo, G. P. Picco, and G.-C. Roman. A Fine-Grained Model for Code Mobility. In *O. Nierstrasz and M. Lemoine (eds), Proceedings of ESEC99 – The 7th European Software Engineering Conference*, Lecture Notes in Computer Science, Vol. 1687, pp. 39-56, Springer-Verlag, 1999.

[10] R. Milner. *A Calculus of Communicating Systems*. Lecture Notes in Computer Science, Vol. 92, Springer-Verlag, 1980.

[11] R. Milner. *Communicating and Mobile Systems: the π-calculus*. Cambridge University Press, 1999.

[12] R. Milner, J. Parrow, and D. Walker. A Calculus of Mobile Processes I and II. In *Information and Computation*, Vol. 100, No. 1, pp. 1-77, 1992.

[13] L. Petre, K. Sere, and M. Waldén. A Topological Approach to Distributed Computing. In *Proceedings of WDS 99 – Workshop on Distributed Systems*, Electronic Notes in Theoretical Computer Science, Vol. 28, pp. 97-118, Elsevier Science, 1999.

[14] G.-C. Roman and P. J. McCann. A Notation and Logic for Mobile Computing. In *Formal Methods in System Design*, Vol. 20, No. 1, pp. 47-68, 2002.

[15] K. Sere and M. Waldén. Data Refinement of Remote Procedures. In *M. Abadi and T. Ito (eds.), Proceedings of TACS'97 – International Symposium on Theoretical Aspects of Computer Software*, Lecture Notes in Computer Science, Vol. 1281, pp. 267-294, Springer-Verlag, 1997.

[16] G. Smith. *The Object-Z Specification Language*. Kluwer Academic Publishers, 2000.

[17] A. S. Tanenbaum. *Computer Networks*, fourth edition. Pearson Education, Inc., Prentice Hall PTR, 2003.

Turku
Centre *for*
Computer
Science

Lemminkäisenkatu 14 A, 20520 Turku, Finland | www.tucs.fi

University of Turku
- Department of Information Technology
- Department of Mathematics

Åbo Akademi University
- Department of Computer Science
- Institute for Advanced Management Systems Research

Turku School of Economics and Business Administration
- Institute of Information Systems Sciences

ISBN 952-12-1696-4
ISSN 1239-1891