Report Number: WUCS-99-07

1999-01-01

A Fine-Grained Model for Code Mobility

Cecilia Mascolo, Gian Pietro Picco, and Gruia-Catalin Roman

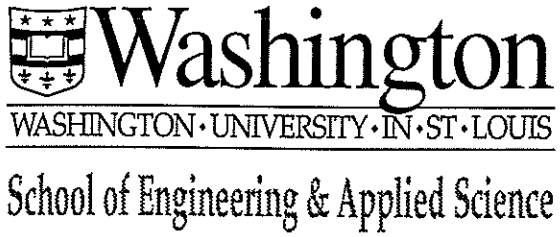
In this paper, we take the extreme view that every line of code is potentially mobile, i.e., may be duplicated and/or moved from one program context to another on the same host or across the network. Our motivation is to gain a better understanding of the range of constructs and issues facing the designer of a mobile code system, in a setting that is abstract and unconstrained by compilation and performance considerations traditionally associated with programming language design. Incidental to our study is an evaluatoin of the expressive power of Mobile UNITY, a notation and proof logic for mobile... Read complete abstract on page 2.

A Fine-Grained Model for Code Mobility

Cecilia Mascolo, Gian Pietro Picco, and Gruia-Catalin Roman

Complete Abstract:

In this paper, we take the extreme view that every line of code is potentially mobile, i.e., may be duplicated and/or moved from one program context to another on the same host or across the network. Our motivation is to gain a better understanding of the range of constructs and issues facing the designer of a mobile code system, in a setting that is abstract and unconstrained by compilation and performance considerations traditionally associated with programming language design. Incidental to our study is an evaluatoin of the expressive power of Mobile UNITY, a notation and proof logic for mobile computing.



A Fine-Grained Model for Code Mobility

Cecilia Mascolo
Gian Pietro Picco
Gruia-Catalin Roman

WUCS-99-07

March 1, 1999

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

A Fine-Grained Model for Code Mobility

Cecilia Mascolo[1,2], Gian Pietro Picco[2], and Gruia-Catalin Roman[2]

[1] Dip. di Scienze dell'Informazione. University of Bologna.
Mura Anteo Zamboni, 7. 40127 Bologna, Italy.
mascolo@cs.unibo.it
[2] Dept. of Computer Science, Washington University, Campus Box 1045,
One Brookings Drive, Saint Louis, MO 63130-4899, USA.
{picco|roman}@cs.wustl.edu

Abstract. In this paper we take the extreme view that every line of code is potentially mobile, i.e., may be duplicated and/or moved from one program context to another on the same host or across the network. Our motivation is to gain a better understanding of the range of constructs and issues facing the designer of a mobile code system, in a setting that is abstract and unconstrained by compilation and performance considerations traditionally associated with programming language design. Incidental to our study is an evaluation of the expressive power of Mobile UNITY, a notation and proof logic for mobile computing.

1 Introduction

The advent of world-wide networks, the emergence of wireless communication, and the growing popularity of the Java language are contributing to a growing interest in dynamic and reconfigurable systems. Code mobility is viewed by many as a key element of a class of novel design strategies which no longer assume that all the resources needed to accomplish a task are known in advance and available at the start of the program execution. Know-how and resources are searched for across the networks and brought together to bear on a problem as needed. Often the program itself (or portions thereof) travels across the network in search of resources. While research has been done in the past on operating systems that provide support for process migration, mobile code languages offer a variety of constructs supporting the movement of code across networks. Java, Tcl, and derivatives support the movement of architecture-independent code that can be shipped across the network and interpreted at execution time [9, 8]. Obliq [3] permits the movement of code along with the reference to resources it needs to carry out its functions. Telescript [16] is representative of a class of languages in which fully encapsulated program units called agents migrate from site to site. Location, movement, unit of mobility, and resource access are concepts present in all mobile code languages. Differentiating factors have to do with the precise definitions assigned to these concepts and the operations available in the language.

Language design efforts are complemented by the development of formal models. Their main purpose is to gain a better understanding of fundamental issues facing mobile computations. Of course, such models are expected to play an

important role in the formulation of precise semantics for mobile code languages and constructs, to serve as a source of inspiration for novel language constructs, and to uncover likely theoretical limitations. Basic differences in mathematical foundations, underlying philosophy, and technical objectives led to models very diverse in flavor. The π-calculus [12] is based on algebra and treats mobility as the ability to dynamically change structure through the passing of names of entities including communication channels. The ambient calculus [4] is also algebraic in style but emphasizes the manipulation of and access to administrative domains captured by a notion of scoping. Mobile UNITY [11] is a state transition system in which the notion of location is made explicit and component interactions are defined by coordination constructs external to the components' code.

The work reported in this paper is closely aligned with the investigative style of the formal models community but directed towards identifying opportunities for novel mobility constructs to be used in language design. We are particularly interested in examining the issue of granularity of movement and in studying the consequences of adopting a fine-grained perspective. Simply put, we asked ourselves the question: What is the smallest unit of mobility and to what extent can the constructs commonly encountered in mobile code languages be built from a given set of fine-grained elements? Proper choice of mobility operations, elegant and uniform semantic specification, formal verification capabilities, and expressive power are several issues closely tied into the answer to the basic question we posed.

In the model we explore here the units of mobility are single statements and variable declarations. Location is defined to be a site address and units can move among sites, can be created dynamically, and can be cloned. Complex structures can be constructed by associating multiple units with a process. The process is the unit of execution in our model. In the simplest terms, a process is merely a common name that binds the units together and controls their execution status—more complex structures can be built but they are outside the scope of this paper. All the mobility operations available for units are also applicable to processes. In addition, processes have the means to share code and resources via a referencing mechanism limited strictly to the confines of a single site. A reference can be thought of as a name that allows one process to access some code or data in some other process. References across sites are not permitted but they survive movement, e.g., access is restored when the two processes meet again. As such, unit reference and unit containment have distinct semantics with respect to both scoping rules and mobility.

Mobile UNITY provides the notational and formal foundations for this study. The new model can be viewed to a large extent as a specialization of Mobile UNITY. This enables us to continue to employ the coordination constructs of Mobile UNITY and its proof logic. The result is a small set of macro definitions that map the fine-grained model proposed here to the standard Mobile UNITY notation, and a simple semantic specification of the mobility constructs in terms of the coordination language that is at the core of Mobile UNITY. This application of Mobile UNITY is novel. Mobile UNITY has been used previously in the

definition of high level transient interactions (e.g., transiently and transitively shared variables) in both a physical and logical mobile setting [11], in formal specification and verification of Mobile IP [10], and in the specification and verification of mobile code paradigms (e.g., code on demand, remote evaluation, and mobile agents) [15].

The structure of the paper is the following. Section 2 contains an informal overview of the model. In Section 3 we introduce the overall structure of the model, in Section 4 we give a description of the mobility primitives of our model, and Section 5 defines their formal semantics. Finally, in Section 6 we compare our approach with the existing ones, and in Section 7 draw some conclusions and discuss future work.

2 Model Overview

We now give an informal overview of our model. We consider a network composed of sites, that are the physical locations on which computations take place. Sites may represent physical hosts, or a separate logical address within a host, e.g., an interpreter. On the sites we have *units* that represent code or data. We do not constrain a unit containing code to contain a complete specification of a code fragment, but we allow it to contain even a single line of code. The variables used in the code units are considered "placeholders" and they do not carry a value (i.e., they carry an undefined value). Units representing data contain a single variable declaration and they carry the actual value of the variable. The model provides a sharing mechanism between values of variables with the same name in code and data units, so that code can execute and change values of variables in data units.

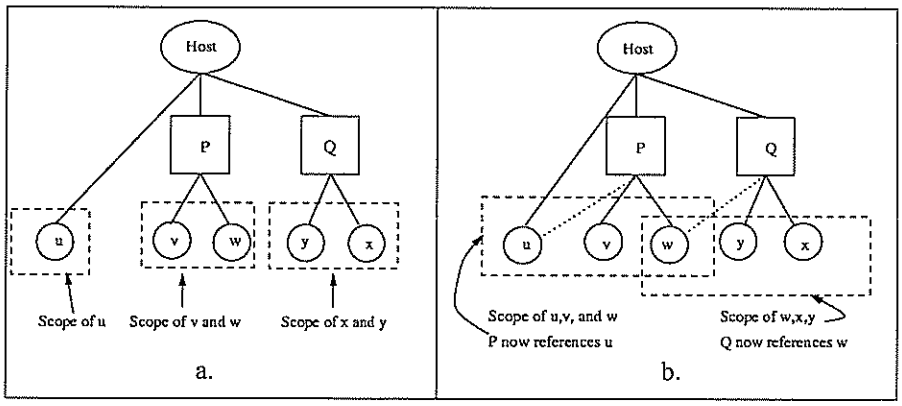


Fig. 1. Processes, units, and scoping rules. Solid lines represent the containment relation among sites, processes, and units, while dotted lines represent referencing of units. Dashed rectangles represent a common scope for units.

Because code and data can be split across units, we need to introduce some notion of composition and scoping. For this purpose we introduce the concept of *process*. Processes are unit containers that reside on the sites. Unlike units they carry an activation status—they can be active, inactive, or terminated. Processes define restricted scopes for the units on the locations. Units can then be located "inside" a process (i.e., in its "private space"), in which case the units are said to be contained in the process.

The model presented in this paper is kept simple by not allowing processes to contain other processes. However, we are investigating this enhancement in order to study its advantages. The scope of a unit contained in a process is the private space of the process (i.e., the space on which the unit is located). The binding mechanisms defined by the model allow sharing among variables with the same name *in the same scope*. The scope of a unit that is not contained in any process (i.e., located directly on the site space) is restricted to the unit itself. In Figure 1.a we show an example. The scope of unit v contains also unit w (and vice versa) as they are both contained in process P, while unit u is not contained in any process and its content is not shared with anyone else.

However, it is natural and necessary to have some notion of sharing of units among processes at the same location in order, for instance, to specify the sharing of a common resource. For this purpose we allow a process to *reference* a unit contained in another process on the same location. In such a case, the unit is considered to be in the scope of both processes. Processes can also reference units not contained in any process (i.e., located directly in the site). These units can be thought of as library classes or resources provided by the site to all processes located there. Figure 1.b shows an evolution of the system from Figure 1.a: here the unit u is referenced by process P, and units u, v, and w are in the same scope. Unit w is referenced by process Q: units x, y, and w are then considered in the same scope, and the sharing mechanism can apply. Notice that units x and y are not in the scope of unit v yet.

A process is a unit of execution in the sense that its status constrains the execution of the code belonging to units inside its scope. Processes are mobile as well: the movement of a process implies the movement of all the units contained. Referenced units however, are not moved along with the process that refers to them as they are not part of its private space. Furthermore, the binding mechanism inhibits the access to referenced units whenever the referencing process

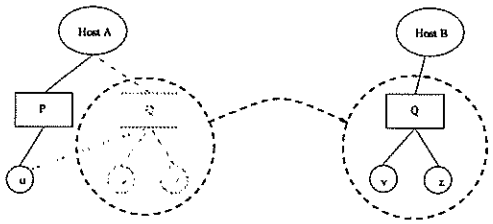


Fig. 2. Movement of a process.

```
System Swapping
  Program P(i) at λ
    declare
      x: integer [] y: integer
    initially
      x < 10 [] y < 10
    assign
        s: x, y := y, x  if x ≥ y
    [] m1: λ := λ + 1
    [] m2: λ := λ − 1
  end
  Components
      ( [] i : 0 ≤ i < N :: P(i).λ = location(i))
  Interactions
      P(i).x ≈ P(j).x      when P(i).λ = P(j).λ
                           engage max{P(i).x, P(j).x}
end
```

Fig. 3. A simple Mobile UNITY system exhibiting random movement.

and the referenced unit are not on the same site. Figure 2 shows a pictorial representation of this concept. It is important to notice, however, that all the references to units are not discarded at the time of the move, and when a referenced unit and the corresponding process become colocated on any site the binding is re-established.

The model also provides mechanisms to generate and duplicate components, to explicitly activate, deactivate, or terminate processes, and to establish or sever a reference between a process and a unit. In the next section we show the details of the structure of the model.

3 Overall Model Structure

We now provide a more formal treatment of the manner in which the model is built. Along the way, we also describe the Mobile UNITY notation. A Mobile UNITY specification is composed of many *programs*, a **Components** section and an **Interactions** section. The program is the basic unit of definition and mobility of the Mobile UNITY document. Figure 3 shows a Mobile UNITY system for reordering values of variables. Distribution of components is taken into account through the distinguished attribute λ associated to each program.

The **declare** section contains the declaration of program variables. The symbol [] acts as a separator. The **initially** section constrains the initial values of the variables. In the example in Figure 3, x and y are only constrained to assume a value less than 10. In the **assign** section the statement named s is a guarded assignment guarded by the clause following the **if**. The two values of the variables are swapped if the value of the first one is greater than the other.

The Mobile UNITY **Components** section defines the components existing during the life of the system. Mobile UNITY does not allow dynamic creation of new components. Mobile UNITY program definition contains an index (i.e., i) after the name of the program (i.e., P). This allows multiple instances of the same program in the **Components** section. In Figure 3, for instance, N different

instances of program P are instantiated and placed at various initial locations based on their index value[1], initialized using the function location.

All the variables of a Mobile UNITY component are considered local to the component. No communication takes place among components without the presence of interaction statements spanning the scope of the components. The **Interactions** section contains these statements, and provides communication and coordination among components. In the example, the **Interaction** section allows the *sharing* of values between the two variables x of different programs when the programs containing them are at the same location. Only some of the program instances end up sharing the values of variables x, depending upon their initial location (see function location and subsequent moves). The Mobile UNITY construct $\approx$ defines transient sharing of values far as long as the **when** condition holds. The **engage** statement defines a common value to be assigned (atomically) to both the variables as the **when** condition transitions from false to true. In this example the value assumed by the two variables is the maximum over their individual values. It is possible to specify also a **disengage** statement that defines the values the two variables would respectively be assigned to whenever the **when** predicate is no longer true. If no **disengage** is specified the variables retain the values they had before the **when** condition became false: in our example no disengagement is specified.

The Mobile UNITY execution consists of a fair interleaving of programs statement executions, including the statements present in the **Interactions** section. The sharing construct has a higher priority and is executed any time a change in the values of the variables involved in the sharing happens.

Mobile UNITY considers a program to be the smallest unit of mobility. In this paper we want to allow mobility of a variable declaration or of a line of code. For this purpose we set out to reinterpret the syntax of a standard Mobile UNITY program such that every variable declaration and every labeled statement is interpreted as a stand-alone program, henceforth called a *unit*. A program now becomes only a static unit of definition. Statements and declarations as well as processes become the units of mobility. Of course, processes also serve as units of execution. With this interpretation, the declaration of x in Figure 3 corresponds to the unit:

Program x_i
 declare x: integer
 initially $x < 10$
 assign skip
end

The name of the unit (i.e., x) is the name of the variable declared in the program; the index i enables us to consider multiple instances of the same unit.

[1] The three-part notation ⟨**op** *quantified_variables* : *range* :: *expression*⟩ will be used throughout the paper. It is defined as follows: The variables from *quantified_variables* take on all possible values permitted by *range*. If *range* is missing, the first colon is omitted and the domain of the variables is restricted by context. Each such instantiation of the variables is substituted in *expression* producing a multiset of values in which **op** is applied.

Notice that a unit capturing a declaration also contains the corresponding initialization statement for the declared variable. This is the definition of what we call a *data* unit. As will be shown in the next section, the annotation **var** is used to distinguish between variable present in pure data units and those appearing in code units. For code units (i.e., units containing statements), the name of the unit is the label of the statement defined in the program (with an index i). The following is a code unit derived from the statement labeled with s in Figure 3:

Program s_i
declare x: integer [] y: integer
initially $x = \perp$ [] $y = \perp$
assign $x, y := y, x$ **if** $x \geq y$
end

The statement is copied in the **assign** section of the unit. All the variables used in the statement are declared (in the **declare** section) and initialized as unbound, i.e., $\perp$. This initialization underlines the fact that this unit only contains code, and that the variables are only placeholders (i.e., do not contain real values).

Having reduced the granularity of the components, we need to introduce some form of containment in order to make it possible to assemble units together. Because the resulting assemblies will form units of execution we call the container *process*. Processes have an index like units in order to allow multiple instances of the same process. Processes can be instantiated and placed on an initial location from within the **Components** section. Since processes are dynamic components we attach to them a status attribute ω that can assume values ACTIVE, INACTIVE, and TERMINATED.

In order to overcome the difficulty of dynamically creating components in Mobile UNITY we assume to have a sufficiently large number of instances of components initially located in a sort of "ether". We formalize this by saying that they reside at the location $\lambda = \perp$, i.e., undefined. In this manner, whenever we need to duplicate, or instantiate a new component we simply can change the location of some component in the ether from undefined to an actual location.

The sharing defined in the **Interactions** section of Figure 3 is given by the programmer. We introduce in our model an automatic sharing mechanism allowing variable sharing inside the scope of a single process. As we indicated in Section 2, variables with the same name in the same scope share the same value. Thus, if we have two units that declare x in the same process we have the sharing of their values without adding an explicit command for that as the automatic sharing mechanism will allow it.

In the following two sections we introduce the mobility constructs we have built. Their semantics is captured as part of the **Interactions** section. As shown later, the specification is simple and very compact. Next we give a general introduction of the mobility operations we provide leaving the formal semantics definition for Section 5.

4 Mobility Constructs

The previous sections illustrated the overall structure of our model, and how it differs from Mobile UNITY, both in terms of syntactic differences in the way a specification is textually laid out, and of semantic differences related to the units of execution, mobility, and definition. Central to our model is the interplay among the notions of execution, scoping, containment, and location. Mobility not only determines the set of resources that are available at a given location, but also allows the dynamic reconfiguration of the code and data associated with a given process. In this section we describe in more detail the set of constructs defined in our model. In the next section, we will use Mobile UNITY to give formal semantics to these constructs.

In order to keep the presentation grounded in a practical example, we consider here a mobile code version of the well-known *leader election* problem among a set of nodes networked in a ring configuration. For the sake of simplicity, our solution will employ a single token, whose value is updated at each node by comparing it with the value of the identifier of the node the token is at. The algorithm is trivial, because it is guaranteed to find the leader in exactly one round. However, the interesting aspect of our solution is not the algorithm, rather the way the distributed computation is deployed into the network.

We assume that no nodes are initially able to take part in a leader election. The distributed algorithm is started by injecting into the ring a process that contains the necessary knowledge about the distributed computation—a *voter*. This process clones itself repeatedly until the whole ring is populated with voters. Interestingly, voters do not contain the logic associated with the token, i.e., they do not know how to compare the node's value with the token's value—the *poll* strategy. The knowledge about this key aspect of the algorithm is injected into the ring in a separate step of the computation under the form of a code unit which is placed on an arbitrary node of the ring. Each voter is able to detect the presence of the poll code unit on its node and move it into its own scope, thus effectively enabling the execution of the unit. The poll code unit has access to a node-level data unit that contains the node value. This enables the comparison needed to vote. Again, a self replicating scheme is employed, where the voter passes on a copy of the unit to the next node in the ring. This structure of the system, where the poll strategy is kept separate and is loaded dynamically into the voter, enables the dynamic reconfiguration of the ring. This happens when a new code unit that contains a different poll strategy is injected in the ring. Again, voters will detect its presence on their sites and replace the old strategy with the new one. Finally, the token is injected in the ring, which starts the actual leader election.

It should be apparent at this point how our example, despite its simplicity, highlights many of the leitmotiv of mobile code: simultaneous migration of the code and state associated with a unit of execution, dynamic linking (and upgrade) of code, location-dependent resource sharing. Thus, for instance, our example can be easily adapted to an active network scenario where a new service, (in our case the ability to perform leader election), is deployed in the network,

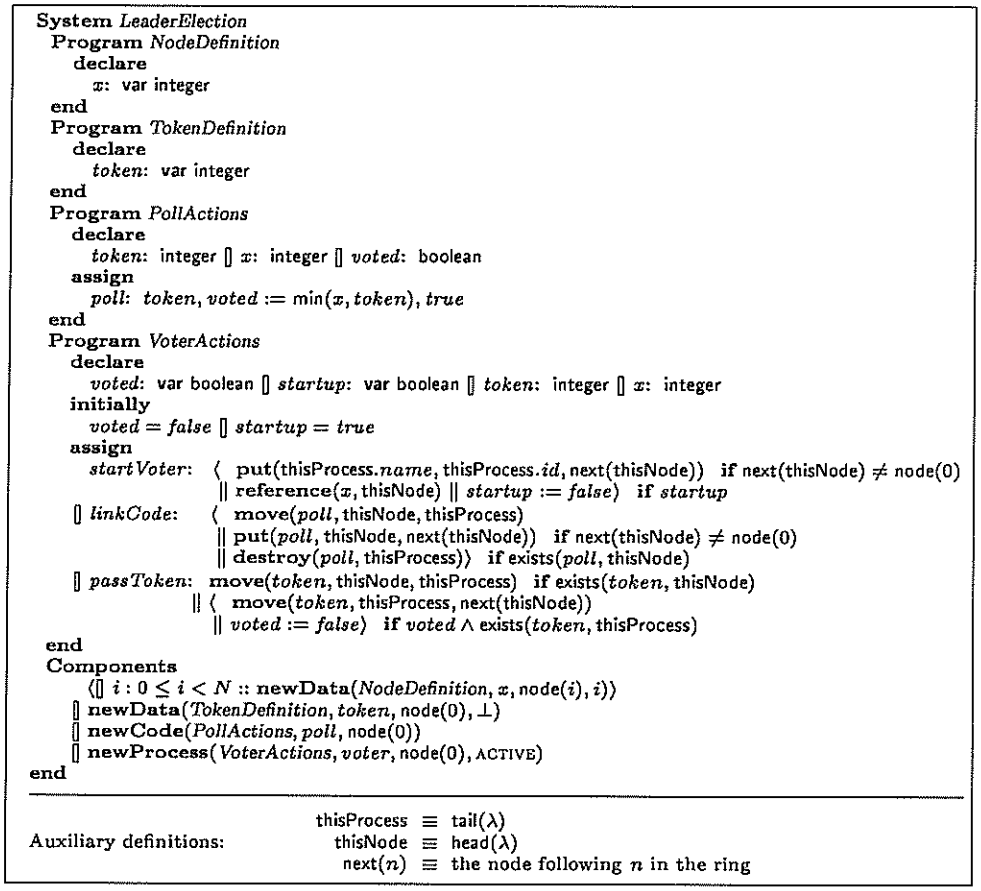


Fig. 4. Specifying leader election in Mobile UNITY extended with fine-grained mobile code constructs.

and some of its constituents (in our case the poll strategy) are dynamically upgraded over time.

A formal specification of our leader election algorithm is shown in Figure 4, while Figure 5 shows its graphical representation. The specification uses the fine-grained mobile code constructs of our model. The upper part of the specification contains three program definitions.

NodeDefinition specifies a single data unit x associated with a node. Note how the type declaration for this integer variable is prepended by the keyword **var**, which characterizes the variable as a data unit, i.e., storing its own value. Similarly, *TokenDefinition* specifies a single data unit associated with the variable *token*. The value of these two variables is accessed through sharing by code units specified by the program *PollActions*. This contains a single statement *poll*, which describes the behavior of the poll strategy. As discussed in the next section, the formal semantics of the model prescribes that execution of this statement

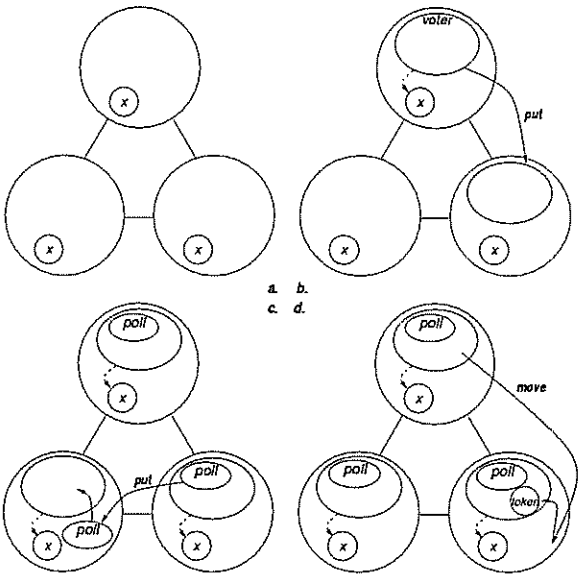


Fig. 5. Leader election with mobile code.

is prevented when the corresponding code unit is not within the scope of any process. Thus, the comparison in *poll* is performed only when the corresponding code unit is co-located in a voter process that also contains the data unit corresponding to *token*. In this case, the binding rules of the model, expressed using the transient variable sharing abstraction provided by Mobile UNITY, effectively force the same value in both *token* variables, hence enabling the comparison specified by *poll*. Simultaneously, an additional auxiliary boolean variable *voted* is set to signal to the enclosing voter, again by means of sharing of the variable *voted*, that the token needs to be passed along the ring.

Voters are specified by the program *VoterActions*, that declares the variables mentioned so far and an additional boolean *startup* that is used to determine whether it is necessary to perform some initialization tasks, i.e., cloning the voter itself on the next node to perform the initial deployment of processes in the ring, and acquiring a reference to the node's value. These tasks are performed simultaneously by the statement *startVoter*, which also resets *startup* to prevent the creation of multiple clones of the voter.

In *startVoter*, cloning is performed by the **put** operation, that executes only if the voter that is invoking the operation does not immediately precede in the ring node(0) where the whole computation started, thus guaranteeing that each node hosts a single voter. The statement uses some of the auxiliary definitions shown at the bottom of the figure. In particular, thisProcess and thisNode are just a renaming of the tail and head functions that operate on the location λ of the voter, and serve the only purpose of improving readability. It is interesting to note, however, that thisProcess contains a fully qualified name for the process,

hence its name and identifier can be accessed using the notation described in the previous section. This is useful in invoking the **put** operation which, in its most general form **put**($name, id, location_{dest}$) expects the *name* and *id* of the component that must be copied and a *location* that represents the destination of the copy. An overloaded form **put**($name, location_{cur}, location_{dest}$) is also provided, that enables the specifier to "query" the scope defined by $location_{cur}$ and gets implicitly the identifier for the component with the given *name*.

As will become more clear in the next section, copying takes place behind the scenes by picking a fresh component from the ether and setting its location to the one passed as a parameter. However, note that the **put** operations, like most of the operations provided in our model, is defined on components, i.e., both on processes and units. Hence, in the case of processes the copying is performed recursively on the process and on all its constituent units. In the case of **put**, the bindings that the process may have established are not preserved as a consequence of this copy operation, i.e., all the variables are restored to their initial values, which effectively represents a "weak" form of copying. Our model provides also a stronger notion with the **clone** operation, that preserves all the bindings owned by the process.

The statement *startVoter* establishes also a reference to the variable x, whose value is contained in a data unit instantiated on each site. To understand in more detail this latter aspect, let us take a brief detour and jump temporarily to the **Components** section, to look at the initial configuration of the system. The first statement uses the operation **newData** to create a data unit named x using the definition provided in the program *NodeDefinition*, assigns to it the value i, and places it on the i^{th} node. Since the statement is quantified over the number N of nodes in the system, each node hosts an instance of the data unit as a result of the operation. Also, it is interesting to note that, after the operation is performed, the name of the data unit can be obtained as $x.name$, its identifier as $x.id$, and its value as $x.value$. However, the explicit reference to the value attribute is usually left implicit when the data unit appears on the left hand side of assignments. Similarly, the other three statements in the **Components** section create on the first node respectively the data unit for the token, the code unit for the poll strategy, and the voter process. Given the nature of our model, which enables movement to the level of a single Mobile UNITY variable or statement, it is interesting to note how *VoterActions* actually represents the unit of definition for a number of units, namely, the data units corresponding to *voted* and *startup*, and the code units corresponding to *startVoter*, *linkCode*, and *passToken*. In principle, each of these could be moved or copied independently. Since this is not the case in this example, they have been grouped together under *VoterActions*. This simplifies the text of the specification by minimizing the number of **Program** declarations, and also enables the creation of a single process that automatically contains instances for all the aforementioned units by using **newProcess**. Finally, note how the value of a process is its activation status, i.e., either ACTIVE or INACTIVE.

Now, let us go back to the **reference** operation in *startVoter*. Thanks to the

binding rules, this operation establishes a transient sharing between the variable in the data unit x defined in *NodeDefinition* and the declaration in the voter. Note how, similarly to what was described for **put**, only the name of the data unit x is specified, while its identifier is determined by implicitly querying the node. The model provides also the inverse operation **unreference**.

The statement *linkCode* takes care of replicating the poll strategy and, possibly, of substituting the new poll code for the old one. It executes only when the exists function in the guard evaluates to true. The function exists, also shown at the bottom of the specification, effectively models the aforementioned query mechanism, and enables *linkCode* to execute only when a code unit with name *poll* is found on the node. If the unit is found, the **move** operation brings it within the process, thus enabling its execution. Simultaneously, a copy of the unit is sent to the next node in the ring via a **put**, provided that the next node is not node(0). At the same time, if a pre-existing *poll* unit is found in the process the **destroy** operation removes it from the system.

Finally, *passToken* handles the movement of the token. Again, the query mechanism is used to get implicitly the identifier of any *token* data unit present on the node and **move** it within the process to establish the proper bindings. After the poll is performed, i.e., *voted* is set to true, the token is moved from the scope of the voter to the next node in the ring.

5 Formal Semantics

Our general strategy is to reduce the new model for code mobility to a specialization of the standard Mobile UNITY notation and proof logic. The first step, explained in the previous sections, shows how we reinterpret a notation which looks very close, if not identical, to that of Mobile UNITY by simply treating each variable declaration and statement as a separate, independent program. Multiple instantiations of each such fine-grained program, called a unit, are defined in the **Components** section. Once this transformation from concrete to an abstract syntax is completed, the still missing parts of the model are the mechanics of data sharing within the confines of each process, the control over the scheduling of statements for execution, and the definition of the various mobility constructs. Our strategy is to capture all these semantic elements as statements present in the **Interactions** section of the Mobile UNITY system and to disallow the programmer from adding anything else to the **Interactions** section. The result is a specialization of Mobile UNITY to the problem of fine-grained mobility. The fact that the entire semantic specification can be reduced to a small set of coordination statements attests to the flexibility of Mobile UNITY. In the remainder of the section we consider in turn the topics of variable sharing, statement scheduling, and mobility operations. Throughout this section we assume that:

1. Each component, i.e., data unit, code unit, or process c_i is characterized by its location ($c_i.\lambda$), name ($c_i.name = c$), identifier ($c_i.id = i$), request field

($s.\rho$) designed to hold mobility commands the system is expected to execute on its behalf, and type ($c_i.\tau \in \{\text{DATAUNIT}, \text{CODEUNIT}, \text{PROCESS}\}$).
2. Each data unit d has also an associated value ($d_i.value$).
3. Each process p is also characterized by a list of contained units called children encoded in the location of the unit, a list of referenced units ($p.\gamma$), and its activity status ($p.\omega \in \{\text{ACTIVE}, \text{INACTIVE}, \text{TERMINATED}\}$).

By and large, the programmer does not need to refer to any of these attributes even though they are essential to the formal semantic definition. When writing code, the programmer will typically refer to a component's name (e.g., c) rather than its fully qualified name (e.g., c_i) consisting of the component name extended by the identifier associated to that specific instantiation of the component. Given the name, the identifier can be extracted easily by employing the functions find and exists defined as follows:

$$\mathsf{find}(u, l) = \begin{cases} i & \text{when } u_i.\lambda = l \text{ for some index } i, \\ \bot & \text{otherwise} \end{cases}$$

$$\mathsf{exists}(u, l) = \mathsf{find}(u, l) \neq \bot$$

5.1 Scoping Rules

Since a code unit can only access its own variables, the mechanism by which we establish scoping and access rules is that of forcing variables with the same name and present in the same scope (i.e., contained in the same process) to be shared. This can be readily captured by employing one of the high level constructs of Mobile UNITY, transient variable sharing across programs ($A.a \approx B.b$ **when** p). The predicate p controlling the sharing simply needs to capture the scoping rules. Figure 6 shows how these rules can be stated as two Mobile UNITY coordination statements. Statement 1 handles sharing between a variable in a data unit and a variable in a code unit while statement 2 defines the sharing between two variables in data units. Statement 1 states that variables $u_i.x$ and $w_j.y$ share the same value when:

- they have the same name;
- u_i is a data unit and w_j is a code unit;
- the two units are at the same location or either the data unit or the code unit is referenced by the process owning the other unit and the two units are on the same site.

The **engage** value is the value of the variable in the data unit. The two **disengage** values are the actual value shared for the data unit variable, and the undefined value for the code unit variable (as variables in code units are not supposed to carry a value unless they are sharing it with a data unit), respectively.

The function sharing tells if two units have a common "parent" (a parent can be the process whom they are linked to or the one form whom they are referenced), i.e. the units are in the same scope. In turn, sharing uses the functions $\mathsf{childOf}(v_k, u_i)$, that indicates the fact that v_k is child of u_i (i.e., v_k is a

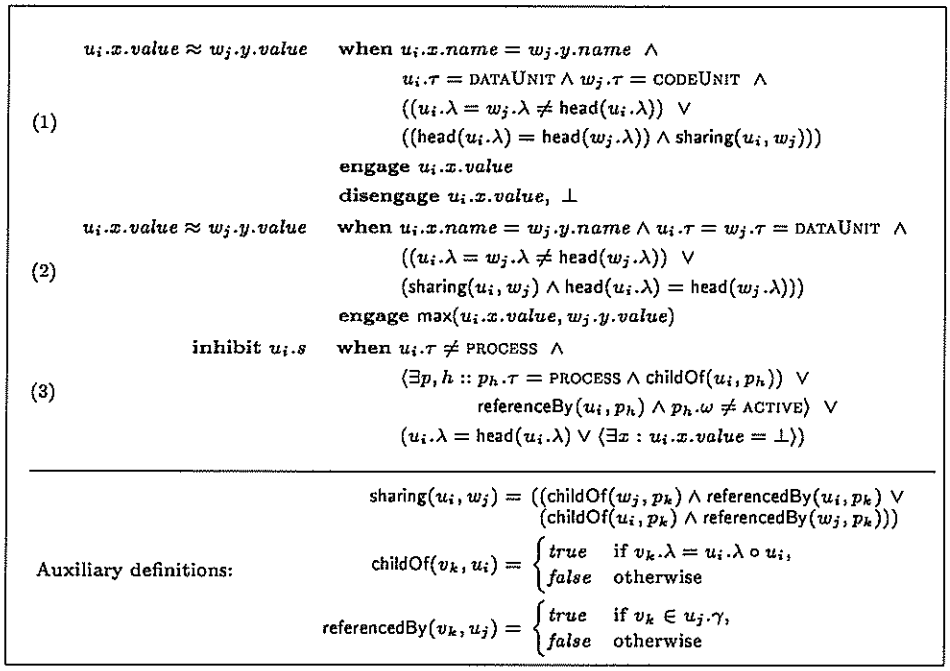


Fig. 6. Establishing bindings among units using transient variable sharing and statement inhibition.

unit linked to u_i), and $\text{referencedBy}(v_k, u_j)$, that indicates the fact that v_k is referenced by u_j.

Statement 2 allows sharing between variables in two data units. The variables must have the same name in the same scope. The sharing happens under the same conditions as in statement 1 except for the fact that both variables are in data units. The **engage** clause forces the two variables to share the maximum value. Different policies can implement a different semantics for reconciliation of values. As no **disengage** is specified the variables retain the values they had before the **when** condition became false.

The update of all shared variables must happen in the same atomic step as the assignment to any of them. However, sharing is specified separately from the (possibly many) assignments that may change the value of a variable. To accomplish this, Mobile UNITY has a two-phased operational model where the first phase involves an ordinary assignment statement execution and the second is responsible for propagating changes to shared variables. We call the statements that execute in the second phase *reactive statements*. Logically, the set of reactive statements are executed to fixed point right after each non reactive statement and one reactive statement may trigger the execution of other reactive statements. Transient sharing is ultimately defined using reactive statements but this is outside the scope of the paper.

5.2 Statement Scheduling

In Mobile UNITY, each statement is assumed to be executed infinitely often in an infinite execution, i.e., weakly fair selection of the statements is the basis for the scheduling process. The coordination constructs of Mobile UNITY include a mechanism for guard strengthening called an **inhibit** construct. In **inhibit** s **when** p, for instance, the statement s continues to be selected as before, but its effect is that of a **skip** whenever the condition p is not met. We take advantage of this construct in statement 3 of Figure 6 to inhibit:

- statements not in the scope of an active process;
- statements that have unbound variables (i.e., undefined values).

A variable appearing in a statement is always *unbound* if it is not shared with a variable present in a data unit.

5.3 Mobility Constructs

The programmer views the **move** construct as a mechanism by which a component at one location is relocated on another known site or is placed within the scope of some other known process. This form of the **move** construct:

$$\textbf{move}(unitName, currentLocation, newLocation)$$

is actually a special instance of the more general form in which the identity of the unit is already known. One can simply determine the identity by employing the function find as in

$$\textbf{move}(unitName, \mathsf{find}(unitName, currentLocation), newLocation)$$

Of course, if multiple instances of the same unit exist one is selected nondeterministically. In order to explore the manner in which we assigned semantics to the mobility constructs associated with our model we will focus our presentation on the general form of the construct. Moreover, we will assume that the unit in question is a process named p with identifier i destined for location l:

$$\textbf{move}(p, i, l).$$

Our general strategy is to treat the operation as a macro reducible to a simple local assignment statement to the distinguished variable ρ (see Figure 7):

$$\rho := (\text{REQ}, \text{MOVE}, p, i, l)$$

where the first two fields of the record stored in ρ indicate the propagation status (i.e., an initial request) and the nature of the request (i.e., a **move**).

We delegate the actual execution of the operation to a series of coordination statements built into the **Interactions** section. The coordination statements propagate the request to the contained units and ultimately carry out the migration of the individual components to the new location. All these actions are executed atomically because they are encoded as reactive statements that execute to fixed point before the system is allowed to take any other action. The

$$
\begin{aligned}
\mathbf{move}(u,i,l) &\equiv \rho := (\text{REQ}, \text{MOVE}, u, i, l)\\
\mathbf{put}(u,i,k,l) &\equiv \rho := (\text{REQ}, \text{PUT}, u, i, (\mathsf{getid}(u), l)) \parallel k := \mathsf{getid}(u)\\
\mathbf{clone}(u,i,k,l) &\equiv \rho := (\text{REQ}, \text{CLONE}, u, i, (\mathsf{getid}(u), l)) \parallel k := \mathsf{getid}(u)\\
\mathbf{destroy}(u,i) &\equiv \rho := (\text{REQ}, \text{DESTROY}, u, i, ())\\
\mathbf{activate}(u,i) &\equiv \rho := (\text{REQ}, \text{ACTIVATE}, u, i, ())\\
\mathbf{deactivate}(u,i) &\equiv \rho := (\text{REQ}, \text{DEACTIVATE}, u, i, ())\\
\mathbf{terminate}(u,i) &\equiv \rho := (\text{REQ}, \text{TERMINATE}, u, i, ())\\
\mathbf{new}(u,k,l) &\equiv \rho := (\text{REQ}, \text{NEW}, u, \mathsf{getid}(u), l) \parallel k := \mathsf{getid}(u)\\
\mathbf{reference}(u,i,v,k) &\equiv \rho := (\text{REQ}, \text{REFERENCE}, u, i, (v,k))\\
\mathbf{unreference}(u,i,v,k) &\equiv \rho := (\text{REQ}, \text{UNREFERENCE}, u, i, (v,k))
\end{aligned}
$$

Auxiliary definitions: $\mathsf{getid}(name) \equiv \langle \min i : name_i.\lambda = \bot :: i \rangle$

Fig. 7. Mapping mobility constructs to Mobile UNITY statements.

(4) $w_j.\rho = \bot$ **if** $w_j \neq u_i \parallel u_i.\rho = (\text{EXEC}, command, u, i, args)$
reacts-to $w_j.\rho = (\text{REQ}, command, u, i, args)$

(5) $u_i.\rho = (command, args) \parallel \langle \parallel v, k : \mathsf{childOf}(v_k, u_i) \wedge \mathsf{toPropagate}(command) ::$
$v_k.\rho = (exec, command, \mathcal{F}(command, u, i, v, args))$
reacts-to $u_i.\rho = (\text{EXEC}, command, u, i, args)\rangle$

Fig. 8. Modeling the actions of the run-time support.

(6) $u_i.\lambda := l$ **if** $(u_i.\omega \neq \text{TERMINATED} \wedge (u(i).\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l)) \wedge u_i.\lambda \neq \bot) \parallel$
$u_i.\rho := \bot$ **reacts-to** $u_i.\rho = (\text{MOVE}, (l))$

(7) $u_j.\lambda, u_j.\omega := l, u_i.\omega$ **if** $(u_i.\lambda \neq \bot \wedge (u_i.\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l))) \parallel$
$u_i.\rho := \bot$ **reacts-to** $u_i.\rho = (\text{PUT}, (j, l))$

(8) $u_j.\lambda, u_j.\omega := l, u_i.\omega$ **if** $(u_i.\lambda \neq \bot \wedge (u_i.\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l))) \parallel$
$u_i.\rho := \bot \parallel \langle \forall x, y : u_i.x.name = u_j.y.name :: u_j.x.value := u_i.x.value \rangle$
reacts-to $u_i.\rho = (\text{CLONE}, (j, l))$

(9) $u_i.\lambda := \bot$ **if** $((u(i).\tau = \text{PROCESS} \Rightarrow u_i.\omega \neq \text{TERMINATED}) \wedge u_i.\lambda \neq \bot) \parallel u_i.\rho := \bot$
reacts-to $u_i.\rho = (\text{DESTROY}, ())$

(10) $u_i.\omega := \text{ACTIVE}$ **if** $(u_i.\omega = \text{INACTIVE} \wedge u_i.\tau = \text{PROCESS} \wedge u_i.\lambda \neq \bot) \parallel u_i.\rho = \bot$
reacts-to $u_i.\rho = (\text{ACTIVATE}, ())$

(11) $u_i.\omega := \text{INACTIVE}$ **if** $(u_i.\omega = \text{ACTIVE} \wedge u_i.\tau = \text{PROCESS}) \parallel u_i.\rho := \bot$
reacts-to $u_i.\rho = (\text{DEACTIVATE}, ())$

(12) $u_i.\omega := \text{TERMINATED}$ **if** $(u_i.\omega \neq \text{TERMINATED} \wedge u_i.\tau = \text{PROCESS} \wedge u_i.\lambda \neq \bot) \parallel u_i.\rho := \bot$
reacts-to $u_i.\rho = (\text{TERMINATE}, ())$

(13) $u_i.\lambda := l$ **if** $(u(i).\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l)) \parallel u_i.\rho := \bot$ **reacts-to** $u_i.\rho = (\text{NEW}, (l))$

(14) $u_i.\gamma := u_i.\gamma \cup \{v_j\}$ **if** $(v_j.\tau \neq \text{PROCESS} \wedge u_i.\tau = \text{PROCESS} \wedge u_i.\lambda \neq \bot \wedge v_j.\lambda \neq \bot) \parallel$
$u_i.\rho = \bot$ **reacts-to** $u_i.\rho = (\text{REFERENCE}, (v_j))$

(15) $u_i.\gamma := u_i.\gamma \setminus \{v_j\} \parallel u_i.\rho := \bot$ **reacts-to** $u_i.\rho = (\text{UNREFERENCE}, v_j)$

Fig. 9. Migrating components.

first thing that happens is to have the request transferred in the form of a command to the process p. The result is that $p_i.\rho$ is assigned the request with a propagation status of EXEC:

$$p_i.\rho := (\text{EXEC}, \text{MOVE}, p, i, l)$$

while the attribute ρ of the unit issuing the request is cleared. Of course, in general it might be the case that a unit requests its own movement and one needs to distinguish between the two cases as made evident in Figure 8.

If, for the sake of simplicity, we assume that the only units contained by p are d_j and s_k, the next reaction being triggered leads to having the process ready to start the move, a fact indicated by dropping the propagation status

$$p_i.\rho := (\text{MOVE}, l)$$

while simultaneously propagating the command to the contained units (see Figure 8), e.g.,

$$d_j.\rho := (\text{EXEC}, \text{MOVE}, d, j, l \circ p_i)$$
$$s_k.\rho := (\text{EXEC}, \text{MOVE}, s, k, l \circ p_i)$$

Figure 10 defines the function $\mathcal{F}$ that computes, in a command-specific manner, the arguments needed by the contained units. In this case, the location to where they need to move is the relocated process. Since further propagation is no longer possible the commands drop the propagation status in the next step

$$d_j.\rho := (\text{MOVE}, d, j, l \circ p_i)$$
$$s_k.\rho := (\text{MOVE}, s, k, l \circ p_i)$$

The last step is the change in location of each of the units (see Figure 9). Given the semantics of Mobile UNITY, this may happen in any order but the reactive statements will be executed again and again until fixpoint is reached, i.e.,

$$p_i.\lambda = l \wedge d_j.\lambda = l \circ p_i \wedge s_k.\lambda = l \circ p_i$$

If an attempt is made to move a unit before the containing process, an apparently inconsistent state is reached in which the unit is located inside of a nonexistent process but this is corrected as soon as the process move is complete. Thus the command completes always in a consistent state.

All other constructs function in a similar manner except that not all the commands are propagated to the contained units. For instance, **terminate** affects only the status of the process. The function toPropagate used in Figure 8 is designed to control the propagation process. The complete list of commands and the corresponding formalization appear in Figure 7 and 9.

6 Discussion

The model presented in this paper is the result of the combination of different research questions. On one hand, it can be regarded as a follow-up of the work described in [15], where mobile code design paradigms were modeled using standard

$$
\begin{aligned}
\mathcal{F}(\text{MOVE}, u, i, v, (l)) &= (l \circ u_i) \\
\mathcal{F}(\text{PUT}, u, i, v, (j, l)) &= (\text{getid}(v), l \circ u_j) \\
\mathcal{F}(\text{CLONE}, u, i, v, (j, l)) &= (\text{getid}(v), l \circ u_j) \\
\mathcal{F}(\text{DESTROY}, u, i) &= ()
\end{aligned}
$$

Fig. 10. Return values for $\mathcal{F}$.

Mobile UNITY. In Mobile UNITY, the unit of mobility coincides with the unit of execution. This is a reasonable choice for mobile computing, the original target of Mobile UNITY, but it is also a common choice for mobile code formalisms. Nevertheless, the aforementioned work highlighted how this choice actually limits the expressiveness of the resulting model in the context of mobile code, where migration of the unit of execution (often called a mobile agent) is only one of the possible relocation strategies. Other paradigms, like remote evaluation, or code on demand [6] require the ability to express the distinction between the unit of execution and a—smaller—unit of mobility that can be linked dynamically to extend and reconfigure the behavior of the unit of execution. This entails also the ability to create dynamically new components, while standard Mobile UNITY provides only static creation of components. Thus, a first motivation for this work was to push further the experience described in the aforementioned paper, to see whether Mobile UNITY could be somehow adapted to the realm of mobile code or a new notation and logic was needed.

Surprisingly enough, the coincidence of unit of mobility and execution is also a common choice for systems supporting mobile code. Although relocation paradigms are in principle independent from the implementation technology, it has been pointed out [7] that the abstractions provided by the latter play a key role as far as ease of programming and overall performance are concerned. In other words, the mobile agent programming abstraction is not always the best, and other relocation strategies concerned with the movement of a portion of the code or state of a mobile agent are often of practical importance when building applications. One of the authors has been involved in the implementation of mobile code applications where this fact has been observed experimentally [2]. This experience eventually led to the implementation of a system, called μCODE [14], whose goals are to provide a minimal set of programming constructs that handle the relocation of arbitrary code and state, and still exploit composability to create higher level abstractions like mobile agents.

The fundamental question of this latter work, despite its slant towards technological aspects, is along the lines of the fundamental question that underlies the work described in this paper. What is the minimal unit of mobility, and what are the fundamental constructs for code mobility? Here, we proposed an answer to these questions by reducing the unit of mobility to a single UNITY variable or statement. These constituents of the unit of execution may be represented in real programming language constructs in a manner that is quite different from variables and statements, typically objects and classes in the case of Java programs. However, the use of an abstract notation enabled us to choose a minimal granule for mobility without being distracted by the details of the

representation chosen in a particular programming language. We coped with the aforementioned problems of dynamic instantiation and finer unit of mobility by providing a specialization of Mobile UNITY where the semantics of the higher level constructs provided by the new model are expressed using standard Mobile UNITY. No modification to the Mobile UNITY notation and logic was required, a fact that makes a strong case for its flexibility.

Other languages for the specification of mobile code systems have been devised, and many of them (e.g., [5]) are based on a process algebra that extends π-calculus [12]. However, in π-calculus there is no notion of location, and movement is modeled by passing a name on a channel. This limitation has been recognized and an explicit notion of location has been modeled, e.g., in [1] and also in [13], where a tuplespace-based language is presented. In the ambient calculus [4], the notion of location is central. *Ambients* are self-contained nested environments that contain data and processes. Ambients can be moved although movement is constrained to be within the boundaries of adjacent locations—i.e., there is no global notion of localities. Finally, in languages based on process algebra processes are first class elements, and there is no explicit formalization for its constituents, i.e., code and the data.

7 Conclusions and Future Work

Code mobility is generally perceived to take place at the level of agents and classes. The model presented in this paper adopts an unusually fine level of granularity by considering the mobility of code fragments as small as single variables and statements. While our primary goal was that of demonstrating the technical feasibility of fine-grained mobility, the study has been instrumental in helping us develop a better understanding of the basic mobility constructs and composition mechanisms needed to support such a paradigm. Composition and scoping emerged as key elements to the construction of complex units out of bits and pieces of code. The need for both containment and reference mechanisms was not in the least surprising given current experience with object-oriented programming languages but it was refreshing to rediscover it coming from a totally new perspective. The distinction between the units of definition, mobility, and execution proved to be very helpful in structuring our thinking about the design of highly dynamic systems. The necessity to provide some form of name service capability in the form of the find function appears to align very well with the current trend in distributed object processing. Finally, the resulting model is unique in its emphasis on verifiability and novel in its usage of cascading reactive statements, a construct akin to event processing but much more general. These features are, to a very large extent, the direct result of our attempt to reduce the programming notation we offer to the semantics of Mobile UNITY. We see verifiability of paramount importance in the logical analysis of systems that exhibit such extraordinary levels of dynamic behavior and restructuring. The examples we presented are indicative of both the expressive power of the proposed model and its potential for practical uses to the development of novel

mobility constructs and applications in which code changes are frequent, e.g., active networks.

References

1. R. Amadio. An Asynchronous Model of Locality, Failure, and Process Mobility. In *Proc. of the 2^{nd} Int. Conf. on Coordination Models and Languages (COORDINATION '97)*, LNCS 1282. Springer, 1997.
2. M. Baldi and G.P. Picco. Evaluating the Tradeoffs of Mobile Code Design Paradigms in Network Management Applications. In *Proc. of the 20^{th} Int. Conf. on Software Engineering*, 1998.
3. L. Cardelli. A language with distributed scope. In *Proc. 22^{nd} ACM Symp. on Principles of Programming Languages (POPL)*, 1995.
4. L. Cardelli and A. Gordon. Mobile Ambients. In *Proc. of Foundations of Software Science and Computation Structures (FoSSaCS), European Joint Conferences on Theory and Practice of Software (ETAPS'98)*, LNCS 1378, Lisbon, Portugal, 1998. Springer.
5. C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In *Proc. 7^{th} Int. Conf. on Concurrency Theory (CONCUR)*, LNCS 1119. Springer, 1996.
6. A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, 24(5), 1998.
7. C. Ghezzi and G. Vigna. Mobile Code Paradigms and Technologies: A Case Study. In *Proc. 1^{st} Int. Workshop on Mobile Agents*, LNCS 1219. Springer, 1997.
8. R. Gray. Agent Tcl: A transportable agent system. In *Proc. of the CIKM Workshop on Intelligent Information Agents*, 1995.
9. J. Kiniry and D. Zimmerman. A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing*, 1(4), 1997.
10. P.J. McCann and G-.C. Roman. Modeling Mobile IP in Mobile UNITY. *ACM Trans. on Software Engineering and Methodology*. To appear.
11. P.J. McCann and G.-C. Roman. Compositional Programming Abstractions for Mobile Computing. *IEEE Trans. on Software Engineering*, 24(2), 1998.
12. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes I. *Information and Computation*, 100(1), 1992.
13. R. De Nicola, G. Ferrari, and R. Pugliese. KLAIM: A Kernel Language for Agents Interaction and Mobility. *IEEE Trans. on Software Engineering*, 24(5), 1998.
14. G.P. Picco. μCODE: A Lightweight and Flexible Mobile Code Toolkit. In K. Rothermel and F. Hohl, editors, *Proc. 2nd Int. Workshop on Mobile Agents*, LNCS 1477. Springer, 1998.
15. G.P. Picco, G.-C. Roman, and P. McCann. Expressing Code Mobility in Mobile UNITY. In *Proc. 6^{th} European Software Eng. Conf. (ESEC/FSE'97)*, LNCS 1301. Springer, 1997.
16. J. White. Telescript Technology: Mobile Agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1996.