A Fine-Grained Model for Code Mobility

Cecilia Mascolo[1,3], Gian Pietro Picco[2,3], and Gruia-Catalin Roman[3]

[1] Dip. di Scienze dell'Informazione, Università di Bologna
Mura Anteo Zamboni 7, 40127 Bologna, Italy
mascolo@cs.unibo.it

[2] Dip. di Elettronica e Informazione, Politecnico di Milano
P.za Leonardo da Vinci 32, 20133 Milano, Italy
picco@elet.polimi.it

[3] Dept. of Computer Science, Washington University
Campus Box 1045, One Brookings Drive, Saint Louis, MO 63130-4899, USA
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 [5] and Tcl [4] derivatives support the movement of architecture-independent code that can be shipped across the network and interpreted at execution time. Obliq [2] permits the movement of code along with the reference to resources it needs to carry out its functions. Telescript [11] 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 constructs available.

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 foundation, underlying philosophy, and technical objectives led to models very diverse in flavor. The π-calculus [8] is based on algebra and treats mobility as the ability to dynamically change structure through the passing of names of entities including communication channels. Several extensions have been proposed, many of which provide an explicit notion of location [1, 9]. In particular, the ambient calculus [3] emphasizes the manipulation of and access to administrative domains captured by a notion of scoping. Mobile Unity [6] 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. 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 specification of the semantics of 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 [6], in the formal specification and verification of Mobile IP [7], and in the specification and verification of mobile code paradigms (e.g., code on demand, remote evaluation, and mobile agents) [10].

The structure of the paper is the following. Section 2 contains an informal overview of the model, Section 3 introduces the overall structure of the model, Section 4 gives a description of the mobility primitives of our model, and Section 5 defines their formal semantics. Finally, in Section 6 we draw some conclusions.

2 Model Overview

We now give an informal overview of our model. We consider a network composed of sites. They are the physical locations on which computations take place. Sites may represent physical hosts or separate logical address spaces within a host, e.g., an interpreter. Sites may contain *units* that represent code or data. A code unit need not contain a complete specification of a code fragment, it may even be a single line of code. The variables used in the code units are considered "placeholders" and they do not carry a value (i.e., their value is undefined). 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, thus code can change values of variables in data units during execution.

Because code and data can be split across units, we need to include 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 sites. Units can be placed inside a process, i.e., in its "private space". Such units are said to be *contained* by the process[1]. The scope of a unit contained by a process is the private space of that 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.

Because it is often necessary to have sharing of units among processes at

[1] The model presented in this paper is kept simple by not allowing processes to contain other processes. We are investigating this enhancement at the present.

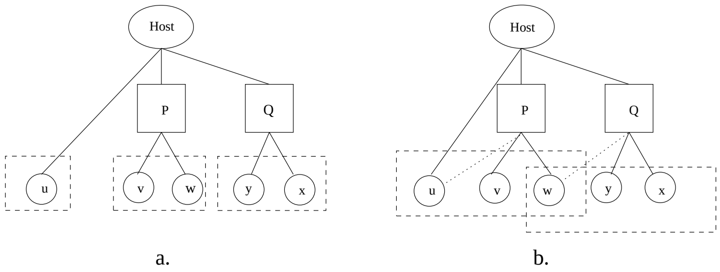

Fig. 1. Processes, units, and scoping rules. Solid lines represent the containment relation among sites, processes, and units, while dotted lines represent references to units. Dashed rectangles represent a common scope for units.

the same location (e.g., to specify the sharing of a common resource), we allow a process to *reference* a unit contained in another process at the same location. In such a case, the referenced 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 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: since units x, y, and w are in the same scope, sharing applies. Notice that units x and y are not in the scope of unit v.

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. A process has an activation status that can be manipulated by specific operations. The code units inside the scope of the process can only be executed when the process is *active*. Processes constrain the mobility of units as well: the movement of a process implies the movement of all the units contained in it. 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 and the referenced unit are not on the same site. It is important to notice, however, that references to units are not discarded at the time of the move; 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 terminate processes, and to establish or sever a reference between a process and a unit. In the next section we present the structure of the model in some detail.

3 Overall Model Structure

In this section we introduce our model for fine-grained mobility and examine its relation to the Mobile UNITY notation. A Mobile UNITY specification is com-

```
System Swapping
  Program Q(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 :: Q(i).λ = location(i)⟩
  Interactions
      Q(i).x ≈ Q(j).x      when Q(i).λ = Q(j).λ
                           engage max{Q(i).x, Q(j).x}
end
```

Fig. 2. A simple Mobile UNITY system exhibiting random movement.

posed of several *programs*, a **Components** section, and an **Interactions** section. The program is the basic unit of definition and mobility in Mobile UNITY. Figure 2 shows a Mobile UNITY system for reordering values of variables. Distribution of components is taken into account through the distinguished location variable 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 of Figure 2, x and y are initialized to an arbitrary value less than 10. The **assign** section contains the program statements. In the example, statement s is an 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 statements m_1 and m_2 account for mobility, by modifying non-deterministically the location of Q.

The **Components** section defines the components existing during the life of the system. Mobile UNITY does not allow dynamic creation of new components. Each Mobile UNITY program contains an index (i.e., i in the example) after the name of the program (i.e., Q). This allows for the creation of multiple instances of the same program in the **Components** section. In Figure 2, for instance, N different instances of program Q are instantiated and placed at various initial locations by using a function location (whose details are left out), and the index value[2].

The **Interactions** section contains statements that provide 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

[2] The three-part notation $\langle$**op** *quantified_variables* : *range* :: *expression*$\rangle$ is used throughout the paper: 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 to which **op** is applied.

Program $p('x,'Q,i)$	**Program** $p('s,'Q,i)$
declare x: integer	**declare** x: integer $[]$ y: integer
initially $x < 10$	**initially** $x = \perp$ $[]$ $y = \perp$
assign skip	**assign** $x, y := y, x$ **if** $x \geq y$
end	**end**

Fig. 3. Two units resulting from a reinterpretation of the program shown in Figure 2.

program instances end up sharing the values of variables x, depending upon their initial location and their subsequent moves. The Mobile UNITY construct $\approx$ defines transient sharing of values for as long as the **when** condition holds. The **engage** statement defines a common value to be assigned (atomically) to both 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, as in our example.

A Mobile UNITY computation consists of a fair interleaving of 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. With this interpretation, the declaration of x in Figure 2 corresponds to the unit $p('x,\ 'Q,\ i)$ in Figure 3. The name of all the units is now the constant p. Each unit is indexed by its name, the name of the program in which it is defined, and by its instance discriminator. This representation is designed to facilitate the search for units present at some location using the name and/or place of definition. We use a quote to distinguish the actual components from their names, in particular for the first two indices which range over finite enumerations. This notation allows the same names to be present in different program contexts. 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 variables present in pure data units and those appearing in code units. For code units (i.e., units containing statements), the first index of the unit is the label of the statement defined in the program. For instance, Figure 3 shows a code unit with name $p('s,\ 'Q,\ i)$ derived from the statement labeled with s in Figure 2. The statement is copied in the **assign** section of the unit. All the variables used in the statement are declared and initialized as unbound, i.e., $\perp$. This initialization underscores the fact that this unit contains only code, and that the variables are mere placeholders, i.e., do not contain real values.

Finally, processes are needed to organize units into executable assemblies. Each process has an index, like a unit, 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 variable ω that can assume the 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 (implicitly) at the location $\lambda = \epsilon$. In this manner, whenever we need to duplicate or instantiate a new component we can simply change the location of some component in the ether from ϵ to an actual location.

The sharing defined in the **Interactions** section of Figure 2 is given by the designer. We introduce in our model an automatic sharing mechanism allowing variable sharing inside the scope of a single process. As mentioned in Section 2, variables with the same name in the same scope share the same value. Thus, if a process contains two units both declaring a variable x their values are shared by definition.

4 Mobility Constructs

The previous sections illustrated the overall structure of our model, and how it differs from Mobile UNITY, in terms of both syntactic differences in the way a specification is textually laid out and 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 a mobile code version of the well-known *leader election* problem for 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 on which the token is located. 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 in 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 each 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 detect its presence on their sites and replace the old strategy with the new one. Finally, when the token is injected into the ring, the actual leader election starts.

Our example, despite its simplicity, highlights many of the leitmotifs of mobile code: simultaneous migration of the code and state associated with a unit of execution, dynamic linking (and upgrade) of code, and location-dependent resource sharing. For instance, our solution 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, 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. Similarly, *TokenDefinition* specifies a data unit associated with the variable *token*. The values of these two variables are accessed (through sharing) by code units specified by the program *PollActions*. The latter contains a single statement *poll*, which describes the polling strategy. As discussed in the next section, the formal semantics of the model prescribes the execution of this statement to be 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

```
System LeaderElection
  Program NodeDefinition
    declare
      x: var integer
  end
  Program TokenDefinition
    declare
      token: var integer
  end
  Program PollActions
    declare
      token: integer [] x: integer [] voted: boolean
    assign
      poll: token, voted := min(x, token), true
  end
  Program VoterActions
    declare
      voted: var boolean [] startup: var boolean [] token: integer [] x: integer [] k: integer
    initially
      voted = false [] startup = true
    assign
      startVoter:  ⟨  put(voter, thisNode, next(thisNode))  if next(thisNode) ≠ node(0)
                    || reference(x, thisNode) || startup := false⟩  if startup
    [] linkCode:    ⟨  move(poll, thisNode, here)
                    || put(poll, thisNode, next(thisNode))  if next(thisNode) ≠ node(0)
                    || destroy(poll, here)⟩  if exists(poll, thisNode)
    [] passToken:  move(token, thisNode, here)  if exists(token, thisNode)
                 || ⟨  move(token, here, next(thisNode))
                    || voted := false⟩  if voted ∧ exists(token, here)
  end
  Components
      ⟨[] i : 0 ≤ i < N :: newData(NodeDefinition, x, node(i), i)⟩
    [] newData(TokenDefinition, token, node(0), ⊥)
    [] newCode(PollActions, poll, node(0))
    [] newProcess(VoterActions, voter, node(0), ACTIVE)
end
-------------------------------------------------------------------------------
                                   here  ≡  λ
Auxiliary definitions:         thisNode  ≡  head(λ)
                                next(n)  ≡  the node following n in the ring
```

Fig. 4. Specifying leader election in Mobile UNITY extended with fine-grained mobile code constructs. The **Interactions** section is assumed to embody the semantics of the refined model (see Section 5).

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. It 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 guarantees that each node hosts a single voter. The statement uses some of the auxiliary definitions shown at the bottom of the figure. In particular, here and thisNode are just renamings of the location variable λ in the voter and of the head function that operate on it, respectively. They serve the sole purpose of improving readability. While the location of a process is always set to the name of a site (as processes reside directly on the site),

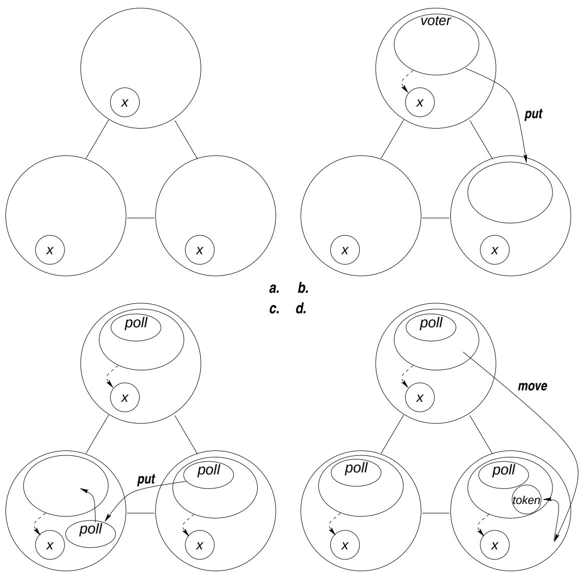

Fig. 5. Leader election with mobile code.

unit location can refer to sites or to processes. In the latter case, the location is defined as the concatenation of the name of the site the unit reside on and of the name of the process that holds it. This is useful in invoking the **put** operation whose most general form is **put**$(name, prog, id, location_{dest})$ where the first three parameters are the three indices of the component to be copied and $location_{dest}$ is a location that represents the destination of the copy. Another form, **put**$(name, location_{cur}, location_{dest})$, is also provided. It is actually used in the example to "query" the scope defined by $location_{cur}$ for the second and third indices of the component given the *name* (i.e., first index).

As will become clearer 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. Like most of the operations provided in our model, the **put** operations is defined on components, i.e., both on processes and units. 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 a 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. This represents a "weak" form of copying. Our model provides also a stronger notion with the **clone** operation, which 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.

Similarly, the other three statements in the **Components** section create on the first node the data unit for the token, the code unit for the poll strategy, and the voter process, respectively. 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 return 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. 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, formally introduced in the next section, 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

$$
\begin{aligned}
\mathsf{find}(u,l) &\equiv \langle \mathsf{min}\ i,j : u_{i,j}.\lambda = l :: (u,i,j)\rangle \\
\mathsf{find}(u,i,l) &\equiv \langle \mathsf{min}\ j : u_{i,j}.\lambda = l :: (u,i,j)\rangle \\
\mathsf{exists}(u,l) &\equiv \langle \exists i,j :: u_{i,j}.\lambda = l\rangle \\
\mathsf{exists}(u,i,l) &\equiv \langle \exists j :: u_{i,j}.\lambda = l\rangle
\end{aligned}
$$

Fig. 6. Specification of the functions find and exists.

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 a concrete to an abstract syntax is completed, the parts of the model still missing 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 designer 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 scoping, statement scheduling, and mobility constructs. From now on we use the compact notation $c_{i,j}$ to mean $p(c,i,j)$, i.e., the instance j of the component named c extracted from program i. Throughout this section we also assume that:

– Each component, (i.e., data unit, code unit, or process) $c_{i,j}$ is characterized by its location ($c_{i,j}.\lambda$), request field ($c_{i,j}.\rho$) designed to hold mobility commands the system is expected to execute on its behalf, and type ($c_{i,j}.\tau \in \{\text{DATAUNIT}, \text{CODEUNIT}, \text{PROCESS}\}$).
– Each process $q_{i,j}$ is also characterized by an implicitly specified set of contained units (those located within the process), a set of referenced units ($q_{i,j}.\gamma$), and its activation status ($q_{i,j}.\omega \in \{\text{ACTIVE}, \text{INACTIVE}, \text{TERMINATED}\}$).

The designer does not need to refer to any of these attributes even though they are essential to the formal semantic definition.

When writing code, the designer will typically refer to a component's name (e.g., c) rather than its fully qualified name (e.g., $c_{i,j}$) consisting of the three indices (i.e., c, i, j) defining the component name, program, and index, respectively. Given the name, the other identifiers can be extracted easily by employing the functions find and exists defined in Figure 6.

The find function finds an instance of the component named u on the location l. The name of the program the unit is derived from (i.e., i) can be added as a parameter in order to constrain the search only to units derived from a particular program definition; the same is true for the function exists. Processes, like other units, also have three indices: the first index is the name of the process, the second is the name of the program the units in the process are derived from (e.g., the process *voter* created with **newProcess** in the **Components** section of Figure 4), and the third is the instance discriminator.

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 7 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[3] $u_{i,h}.x$ and $w_{j,k}.x$ share the same value when $u_{i,h}$ is a data unit and $w_{j,k}$ is a code unit, and the two units are within the same process, 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, respectively—variables in code units are not supposed to carry a value unless they are sharing it with a data unit. The function $\mathsf{sharing}$ tells if two units have a common "parent" (a parent can be the process within which they are located or the one which references them), i.e., the units are in the same scope. In turn, $\mathsf{sharing}$ uses the functions $\mathsf{childOf}(v_{j,k}, u_{i,h})$, that determines whether $v_{j,k}$ is child of $u_{i,h}$ (i.e., $v_{j,k}$ is a unit contained in $u_{i,h}$), and $\mathsf{referencedBy}(v_{j,k}, u_{i,h})$, that determines whether $v_{j,k}$ is referenced by $u_{i,h}$.

Statement 2 defines sharing between variables in two data units. The variables must have the same name in the same scope. Sharing takes place under the same conditions of statement 1, except 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

[3] The formulae in Figure 7 and following assume that variable sharing is well-defined, i.e., it takes places only among variables which actually appear in the specification of a unit according to the program definition. Also, distinguished variables like λ and τ are never shared. The formal definition of these conditions is omitted for the sake of brevity.

$$
\begin{array}{lrl}
(1) & u_{i,h}.x \approx w_{j,k}.x & \textbf{when } u_{i,h}.\tau = \textsc{dataUnit} \wedge w_{j,k}.\tau = \textsc{codeUnit} \ \wedge \\
 & & \qquad (u_{i,h}.\lambda = w_{j,k}.\lambda \neq \mathsf{head}(u_{i,h}.\lambda) \ \vee \\
 & & \qquad (\mathsf{sharing}(u_{i,h}, w_{j,k}) \wedge \mathsf{head}(u_{i,h}.\lambda) = \mathsf{head}(w_{j,k}.\lambda))) \\
 & & \textbf{engage } u_{i,h}.x \\
 & & \textbf{disengage } u_{i,h}.x, \ \perp \\
(2) & u_{i,h}.x \approx w_{j,k}.x & \textbf{when } u_{i,h}.\tau = w_{j,k}.\tau = \textsc{dataUnit} \ \wedge \\
 & & \qquad (u_{i,j}.\lambda = w_{j,k}.\lambda \neq \mathsf{head}(w_{j,k}.\lambda) \ \vee \\
 & & \qquad (\mathsf{sharing}(u_{i,h}, w_{j,k}) \wedge \mathsf{head}(u_{i,h}.\lambda) = \mathsf{head}(w_{j,k}.\lambda))) \\
 & & \textbf{engage } \mathsf{max}(u_{i,h}.x, w_{j,k}.x) \\
(3) & \textbf{inhibit } u_{i,h}.s & \textbf{when } u_{i,h}.\tau = \textsc{codeUnit} \ \wedge \\
 & & \qquad (\langle \forall p, m, n : p_{m,n}.\tau = \textsc{process} \wedge (\mathsf{childOf}(u_{i,h}, p_{m,n}) \ \vee \\
 & & \qquad \mathsf{referencedBy}(u_{i,h}, p_{m,n})) :: p_{m,n}.\omega \neq \textsc{active} \rangle \vee \\
 & & \qquad \langle \exists x :: u_{i,h}.x = \perp \rangle)
\end{array}
$$

Auxiliary definitions:

$$
\begin{aligned}
\mathsf{sharing}(u_{i,h}, w_{j,k}) &= \langle \exists p, m, n :: (\mathsf{childOf}(w_{j,k}, p_{m,n}) \wedge \mathsf{referencedBy}(u_{i,h}, p_{m,n})) \ \vee \\
&\qquad\qquad (\mathsf{childOf}(u_{i,h}, p_{m,n}) \wedge \mathsf{referencedBy}(w_{j,k}, p_{m,n})) \rangle \\
\mathsf{childOf}(v_{j,k}, u_{i,h}) &= \begin{cases} \mathit{true} & \text{if } v_{j,k}.\lambda = u_{i,h}.\lambda \circ (u, i, h) \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathsf{referencedBy}(v_{j,k}, u_{i,h}) &= \begin{cases} \mathit{true} & \text{if } (v, j, k) \in u_{i,h}.\gamma \\ \mathit{false} & \text{otherwise} \end{cases}
\end{aligned}
$$

Fig. 7. Establishing bindings among units using transient variable sharing and statement inhibition.

other reactive statements. Transient sharing is ultimately defined using reactive statements [6], but this is outside the scope of this paper.

Statement Scheduling. In Mobile Unity, each statement is assumed to be executed infinitely often in an infinite execution, i.e., weakly fair selection of statements is the basis for the scheduling process. The coordination constructs of Mobile Unity include a construct for guard strengthening called **inhibit**. 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 7 to inhibit statements not in the scope of an active process, and statements that have unbound variables. A variable appearing in a statement is always *unbound* if it is not shared with a variable present in a data unit.

Mobility Constructs. The designer views the **move** construct as a mechanism by which a component at one location is relocated to another. The new location may be a known site or a known process. This form of the **move** construct:

$$\textbf{move}(\mathit{compName}, \mathit{currentLocation}, \mathit{newLocation})$$

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[4]

$$\mathbf{move}(\mathsf{find}(compName, currentLocation), newLocation).$$

If multiple instances of the same unit exist one is selected[5]. 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 q with identifier (i, j) destined for location l:

$$\mathbf{move}(q, i, j, l).$$

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 8):

$$\rho := (\textsc{req}, \textsc{move}, q, i, j, (l))$$

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 first thing that happens is to have the request transferred in the form of a command to the process q. The result is that $q_{i,j}.\rho$ is assigned the request with a propagation status of EXEC:

$$q_{i,j}.\rho := (\textsc{exec}, \textsc{move}, q, i, j, (l))$$

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 9.

If, for the sake of simplicity, we assume that the only units contained by q are $d_{m,h}$ and $s_{k,n}$, the next reaction being triggered leads to having the process ready to start the move, a fact indicated by dropping the propagation status

$$q_{i,j}.\rho := (\textsc{move}, (l))$$

while simultaneously propagating the command to the contained units (see Figure 9), e.g.,

$$\begin{aligned} d_{m,h}.\rho &:= (\textsc{exec}, \textsc{move}, d, m, h, (l \circ (q, i, j))) \\ s_{k,n}.\rho &:= (\textsc{exec}, \textsc{move}, s, k, n, (l \circ (q, i, j))) \end{aligned}$$

Figure 9 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

[4] Throughout, we assume that $\mathbf{move}((q, i, j), l)$ is unambiguously reducible to $\mathbf{move}(q, i, j, l)$.

[5] We chose to pick up the instance with minimum index.

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

Auxiliary definitions:

$$
\begin{aligned}
\mathsf{getid}(name) &\equiv \mathsf{find}(name, \epsilon) \\
\mathsf{getid}(name, i) &\equiv \mathsf{find}(name, i, \epsilon)
\end{aligned}
$$

Fig. 8. Mapping mobility constructs to Mobile UNITY statements.

$$
\begin{aligned}
&(4) \quad w_{j,k}.\rho = \bot \ \ \textbf{if} \ w_{j,k} \neq u_{i,h} \parallel u_{i,h}.\rho = (\text{EXEC}, command, u, i, h, args) \\
&\qquad \textbf{reacts-to} \ w_{j,k}.\rho = (\text{REQ}, command, u, i, h, args) \\
&(5) \quad u_{i,h}.\rho = (command, args) \parallel \langle \| \ v,n,m : \mathsf{childOf}(v_{n,m}, u_{i,h}) \land \mathsf{toPropagate}(command) :: \\
&\qquad\qquad v_{n,m}.\rho = (\text{EXEC}, command, v, n, m, \mathcal{F}(command, u, i, h, args)) \\
&\qquad \textbf{reacts-to} \ u_{i,h}.\rho = (\text{EXEC}, command, u, i, h, args) \rangle
\end{aligned}
$$

Return values for $\mathcal{F}$:

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

Fig. 9. Modeling the actions of the run-time support.

$$
\begin{aligned}
d_{m,h}.\rho &:= (\text{MOVE}, d, m, h, (l \circ (q,i,j))) \\
s_{k,n}.\rho &:= (\text{MOVE}, s, k, n, (l \circ (q,i,j)))
\end{aligned}
$$

The last step is the change in location of each of the units (Figure 10). Given the semantics of Mobile UNITY, this may happen in any order but the reactive statements will be executed again and again until fixed point is reached, i.e.,

$$q_{i,j}.\lambda = l \ \land \ d_{m,h}.\lambda = l \circ (q,i,j) \ \land \ s_{k,n}.\lambda = l \circ (q,i,j)$$

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 9 is designed to control the propagation process: the propagating constructs are **move**, **put**, **clone**, and **destroy**. The construct getid returns the three-part identity of a component located in the ether. A minimal lexicographical value for the triplet is selected. The complete list of commands and the corresponding formalization appear in Figures 8 and 10.

(6)	$u_{i,h}.\lambda := l$ **if** $(u_{i,h}.\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l)) \wedge u_{i,h}.\omega \neq \text{TERMINATED} \wedge u_{i,h}.\lambda \neq \epsilon \parallel$ $u_{i,h}.\rho := \perp$ **reacts-to** $u_{i,h}.\rho = (\text{MOVE}, (l))$
(7)	$u_{j,k}.\lambda, u_{j,k}.\omega := l, u_{i,h}.\omega$ **if** $(u_{i,h}.\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l)) \wedge u_{i,h}.\lambda \neq \epsilon \parallel$ $u_{i,h}.\rho := \perp$ **reacts-to** $u_{i,h}.\rho = (\text{PUT}, ((u, j, k), l))$
(8)	$u_{j,k}.\lambda, u_{j,k}.\omega := l, u_{i,h}.\omega$ **if** $(u_{i,h}.\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l)) \wedge u_{i,h}.\lambda \neq \epsilon \parallel$ $u_{i,h}.\rho := \perp \parallel \langle \forall x :: u_{j,k}.x := u_{i,h}.x \rangle$ **reacts-to** $u_{i,h}.\rho = (\text{CLONE}, ((u, j, k), l))$
(9)	$u_{i,h}.\lambda := \perp$ **if** $u_{i,h}.\lambda \neq \epsilon \parallel u_{i,h}.\rho := \perp$ **reacts-to** $u_{i,h}.\rho = (\text{DESTROY}, ())$
(10)	$u_{i,h}.\omega := \text{ACTIVE}$ **if** $u_{i,h}.\omega = \text{INACTIVE} \wedge u_{i,h}.\tau = \text{PROCESS} \wedge u_{i,h}.\lambda \neq \epsilon \parallel u_{i,h}.\rho = \perp$ **reacts-to** $u_{i,h}.\rho = (\text{ACTIVATE}, ())$
(11)	$u_{i,h}.\omega := \text{INACTIVE}$ **if** $u_{i,h}.\omega = \text{ACTIVE} \wedge u_{i,h}.\tau = \text{PROCESS} \wedge u_{i,h}.\lambda \neq \epsilon \parallel u_{i,h}.\rho := \perp$ **reacts-to** $u_{i,h}.\rho = (\text{DEACTIVATE}, ())$
(12)	$u_{i,h}.\omega := \text{TERMINATED}$ **if** $u_{i,h}.\omega \neq \text{TERMINATED} \wedge u_{i,h}.\tau = \text{PROCESS} \wedge u_{i,h}.\lambda \neq \epsilon \parallel$ $u_{i,h}.\rho := \perp$ **reacts-to** $u_{i,h}.\rho = (\text{TERMINATE}, ())$
(13)	$u_{i,h}.\lambda := l$ **if** $u_{i,h}.\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l) \parallel u_{i,h}.\rho := \perp$ **reacts-to** $u_{i,h}.\rho = (\text{NEW}, l)$
(14)	$u_{i,h}.\gamma := u_{i,h}.\gamma \cup \{(v, j, k)\}$ **if** $v_{j,k}.\tau \neq \text{PROCESS} \wedge u_{i,h}.\tau = \text{PROCESS} \wedge u_{i,h}.\lambda \neq \epsilon \wedge$ $v_{j,k}.\lambda \neq \epsilon \parallel u_{i,h}.\rho = \perp$ **reacts-to** $u_{i,h}.\rho = (\text{REFERENCE}, (v, j, k))$
(15)	$u_{i,h}.\gamma := u_{i,h}.\gamma \setminus \{(v, j, k)\} \parallel u_{i,h}.\rho := \perp$ **reacts-to** $u_{i,h}.\rho = (\text{UNREFERENCE}, (v, j, k))$

Fig. 10. Migrating components.

6 Conclusions

This paper can be regarded as a follow-up on our earlier work on modeling mobile code paradigms using Mobile UNITY [10]. By contrast, 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. Our primary goal was to demonstrate the feasibility of specifying and reasoning about computations involving fine-grained mobility. Nevertheless, the study has been instrumental in helping us develop a better understanding of 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 (the find function) appears to align very well with the current trend in distributed object processing. The next step is to revisit fine-grained mobility from a more pragmatic perspective, one which will encompass both the design of a fine-grained mobile code system and its use in distributed applications.

Acknowledgments. This paper is based upon work supported in part by the National Science Foundation (NFS) under grant CCR-9624815.

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. L. Cardelli. A language with distributed scope. In *Proc. 22^{nd} ACM Symp. on Principles of Programming Languages (POPL)*, 1995.
3. L. Cardelli and A. Gordon. Mobile Ambients. *Theoretical Computer Science*, 240(1), 2000. To appear.
4. R. Gray. Agent Tcl: A transportable agent system. In *Proc. of the CIKM Workshop on Intelligent Information Agents*, 1995.
5. J. Kiniry and D. Zimmerman. A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing*, 1(4), 1997.
6. P.J. McCann and G.-C. Roman. Compositional Programming Abstractions for Mobile Computing. *IEEE Trans. on Software Engineering*, 24(2), 1998.
7. P.J. McCann and G-.C. Roman. Modeling Mobile IP in Mobile UNITY. *ACM Trans. on Software Engineering and Methodology*, 1999. To appear.
8. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes I. *Information and Computation*, 100(1), 1992.
9. 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.
10. 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.
11. J. White. Telescript Technology: Mobile Agents. In J. Bradshaw, editor, *Software Agents*. AAAI Press/MIT Press, 1996.