Report Number: WUCS-00-02

2000-01-01

CodeWeave: Exploring Fine-Grained Mobility of Code

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

This paper explores the range of constructs and issues facing the designer of mobile code systems which allow for the unit of mobility to be finer-grained than that of execution. Mobile UNITY, a notation and proof logic for mobile computing, provides for this research a clean abstract setting, i.e., unconstrained by compilation and performance considerations traditionally associated with programming language design. Within the context of Mobile UNITY, we take the extreme view that every line of code and every variable declaration is potentially mobile, i.e., it may be duplicated and/or moved from one program contxt to another on the... Read complete abstract on page 2.

Recommended Citation

Mascolo, Cecilia; Picco, Gian Pietro; and Roman, Gruia-Catalin, "CodeWeave: Exploring Fine-Grained Mobility of Code" Report Number: WUCS-00-02 (2000). *All Computer Science and Engineering Research.* https://openscholarship.wustl.edu/cse_research/280

Department of Computer Science & Engineering - Washington University in St. Louis
Campus Box 1045 - St. Louis, MO - 63130 - ph: (314) 935-6160.

CodeWeave: Exploring Fine-Grained Mobility of Code

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

Complete Abstract:

This paper explores the range of constructs and issues facing the designer of mobile code systems which allow for the unit of mobility to be finer-grained than that of execution. Mobile UNITY, a notation and proof logic for mobile computing, provides for this research a clean abstract setting, i.e., unconstrained by compilation and performance considerations traditionally associated with programming language design. Within the context of Mobile UNITY, we take the extreme view that every line of code and every variable declaration is potentially mobile, i.e., it may be duplicated and/or moved from one program contxt to another on the same host or across the network. We also assume that complex code systems may move with equal ease. The result is CodeWeave, a model for abstract exploration of new forms of code mobility prior to their integration into programming systems and middleware.

CodeWeave: Exploring Fine-Grained Mobility of Code

Cecilia Mascolo, Gian Pietro Picco and
Gruia-Catalin Roman

WUCS-00-02

January 2000

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis MO 63130

CodeWeave: Exploring Fine-Grained Mobility of Code

Cecilia Mascolo*†, Gian Pietro Picco‡†, and Gruia-Catalin Roman†

Abstract

This paper explores the range of constructs and issues facing the designer of mobile code systems which allow for the unit of mobility to be finer-grained than that of execution. Mobile Unity, a notation and proof logic for mobile computing, provides for this research a clean abstract setting, i.e., unconstrained by compilation and performance considerations traditionally associated with programming language design. Within the context of Mobile Unity, we take the extreme view that every line of code and every variable declaration is potentially mobile, i.e., it may be duplicated and/or moved from one program context to another on the same host or across the network. We also assume that complex code systems may move with equal ease. The result is CodeWeave, a model for abstract exploration of new forms of code mobility prior to their integration into programming systems and middleware.

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 [11, 9] 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 [19] is representative of a class of languages in which fully encapsulated program units called mobile 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 improve our 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 [14] 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 [3] is also algebraic in style but emphasizes the manipulation of and access to administrative domains captured by a notion of scoping. Mobile Unity [13] 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.

*Dept. of Computer Science, University College London, Gower Street, London WC1E 6BT, UK. E-mail: c.mascolo@cs.ucl.ac.uk.

†Dept. of Computer Science, Washington University, Campus Box 1045, One Brookings Drive, Saint Louis, MO 63130-4899, USA. E-mail: roman@cs.wustl.edu.

‡Dip. di Elettronica e Informazione, Politecnico di Milano, P.za Leonardo da Vinci 32, 20133 Milano, Italy. E-mail: picco@elet.polimi.it.

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. However, unit reference and unit containment have distinct semantics with respect to both scoping rules and mobility.

The resulting model is one of many that can be constructed using the methods outlined in this paper. The model is novel in several aspects: it allows for the movement of code fragments that are much smaller than the units of execution; it facilitates dynamic restructuring of complex nested code structures involving either a single or multiple sites; and it highlights important distinctions among fundamental concepts of practical significance (i.e., containment, reference, unit of definition, unit of execution, unit of mobility, and so on). Finally, as showed in a later section, the model is sufficiently expressive to be able to capture in a straightforward fashion most code mobility constructs already in use today.

Mobile UNITY [13] provides the notational and formal foundation for this study. The new model, CODEWEAVE, 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 semantics 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. Previous usage of Mobile UNITY in code mobility [17] dealt with the specification and verification of mobile code paradigms (e.g., code on demand, remote evaluation, and mobile agent) in which the smallest unit of mobility coincided with the smallest unit of execution.

The structure of the paper is the following. Section 2 contains an informal overview of CODEWEAVE. 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. In Section 6 we illustrate an enhancement of the model that uses processes as nested scopes for the units, and in Section 7 we show how some existing mobile code mechanisms and technologies can be captured using our approach. In Section 8 we illustrate some related work, and in Section 9 we 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. 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 needs not to 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.

As 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 containers that reside on sites and define restricted scopes for the units. Units can be placed inside a process, and are said to be *contained* by the process[1]. The scope of a unit contained in a process is the process itself, i.e., a contained unit can only access units that are contained in the same process. 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) 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 the two units 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 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 on the site). These units can be thought of as library

[1] The model presented in this section is kept simple by not allowing processes to contain other processes. We investigate this enhancement in Section 6.

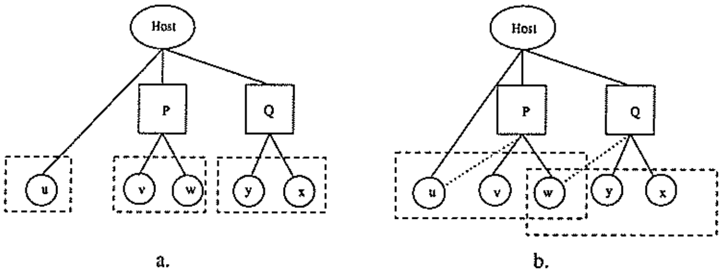

Figure 1: Processes, units, and scoping rules. Solid lines represent the containment relation among sites, processes, and units, while dotted lines represent references to units. Dashed rectangles represent a common scope for units.

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, thus placing units u, v, and w 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 (i.e., whether it is active, inactive, or terminated) constrains the execution of the code belonging to units inside its scope. 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 contained in it. 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 co-located on any site the binding is re-established. This particular reference and binding policy is somewhat arbitrary and is used for illustration purposes only. Other schemes may be constructed by making only small adjustments to the model as presented. CODEWEAVE also provides mechanisms to generate and duplicate processes and units, to explicitly terminate processes, and to establish or sever a reference between a process and a unit. In the next section we introduce the Mobile UNITY notation and explain the mechanisms of constructing CODEWEAVE as a specialization of Mobile UNITY.

3 Overall Model Structure

We start this section with a brief review of the Mobile UNITY notation [13]. The presentation provides just enough notation to enable us to introduce the basic elements of CODEWEAVE. The general strategy will be to reinterpret each Mobile UNITY program as a set of programs, one for each variable and statement in the original code. This is accomplished in a very mechanistic fashion and, in turn facilitates the mobility of the smallest code fragments one can conceive in the context of Mobile UNITY.

Throughout this entire section we use a leader election program for illustration purposes. N nodes are arranged in a ring each holding a value x. A mobile agent moves around the ring carrying a *token* that is used to compute the lowest value of the variables x stored on each node. The token value is updated at each node by comparing it with the local value of x. The algorithm is guaranteed to find the leader in exactly one round but for simplicity we allow the agent to circulate indefinitely around the ring.

3.1 Mobile Unity

A Mobile UNITY specification consists of several *programs*, a **Components** section and an **Interactions** section. The **Program** is the basic unit of definition and mobility in a Mobile UNITY system. Distribution of components is taken into account through the distinguished location attribute λ associated to each program. Changes in the value of λ denote movement. Figure 2 shows a Mobile UNITY solution for the *leader election* problem. The system contains two programs, *NodeValue* and *Agent*. The **declare** section of each program contains the declaration of its program variables. The symbol $[]$ acts as a separator. The **initially** section constrains the initial values of the variables. In program *NodeValue* of Figure 2, x is initialized using a function id which, given an index i, returns a unique value associated to it. In the program *Agent*, two variables are declared, *token* and x. The variable *token* is initialized to

```
System LeaderElection
  Program NodeValue(i) at $\lambda$
    declare
      $x$: integer
    initially
      $x = \mathsf{id}(i)$
    assign
      skip
  end
  Program Agent at $\lambda$
    declare
      $x$: integer [] $token$: integer
    initially
      $x = \perp$ [] $token = \mathsf{numNodes}$
    assign
      $poll$: $token := \min(x, token)$  if $x \neq \perp \wedge token \neq \perp$ || $\lambda := \mathsf{next}(\lambda)$
  end
  Components
      $\langle [] \; i : 0 \leq i < \mathsf{numNodes} :: NodeValue(i).\lambda = \mathsf{location}(i) \rangle$ [] $Agent.\lambda = \mathsf{location}(0)$
  Interactions
      $NodeValue(i).x \approx Agent.x$      when $NodeValue(i).\lambda = Agent.\lambda$
                                            engage $NodeValue(i).x$
                                            disengage $NodeValue(i).x, \perp$
end
```

Auxiliary definitions:		
	numNodes	≡ the number of nodes in the system
	id(n)	≡ a unique identifier generated from n
	next(n)	≡ the location of the next node in the ring
	location(n)	≡ the location of node n

Figure 2: Leader election using a mobile agent modeled with Mobile UNITY.

the number of hosts in the system, numNodes, and x is left undefined. In the **assign** section of program *Agent* the statement named *poll* sets the value of *token* to the minimum between its value and that of x, if x is not undefined, and moves the agent to the next node by changing the value of the location attribute λ. The function next returns the next node of the ring. The symbol || makes the two statements on its left and right to be executed synchronously.

The Mobile UNITY **Components** section defines the components existing throughout the life of the system. Mobile UNITY does not allow dynamic creation of new components. In Mobile UNITY a program definition may contain an index after the name of the program. This allows for multiple instances of the same program to be defined in the **Components** section. In Figure 2, multiple instances of program *NodeValue* are created and placed at various initial locations based on their index value[2], initialized using the function location, while only one instance of program *Agent* is created (the index is dropped for the sake of readability).

All the variables of a Mobile UNITY component are considered local to the component. No communication takes place among components in the absence of interaction statements spanning the scope of multiple components. The **Interactions** section contains statements that provide communication and coordination among components. In the example of Figure 2, the **Interaction** section allows the *sharing* of values between the two variables named x in the programs *NodeValue(i)* and *Agent* when they happen to be at the same location: the dot notation is used here to address the variable in the program, and the index i is supposed to be universally quantified. 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 variables 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 value of the x on the node. It contains the actual value to be used for computing the leader. It is also possible to specify a **disengage** statement that defines the values assumed by the two variables, respectively, when the **when** predicate transitions to false. If no **disengage** statement is specified, the variables retain the values they had before the **when** condition became false. In the example, the disengagement value for the x variable on the node is its current value, while the value of the x carried by the agent is set to undefined as it has to carry no value.

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

```
System LeaderElection
  Program p('x,'NodeValue, i) at λ
    declare
      x: integer
    initially
      x = id(i)
    assign
      skip
  end
  Program p('token,'Agent, i) at λ
    declare
      token: integer
    initially
      token = numNodes
    assign
      skip
  end
  Program p('poll,'Agent, i) at λ
    declare
      x: integer [] token: integer
    initially
      x = ⊥ [] token = ⊥
    assign
      token := min(x, token)  if x ≠ ⊥ ∧ token ≠ ⊥ || λ := next(λ)
    || p('token, find('token, λ) ↑ 1, find('token, λ) ↑ 2).λ = next(λ)
  end
  Components
      ⟨ [] i : 0 ≤ i < numNodes :: p('x,'NodeValue, i)⟩.λ = location(i)
    [] p('token,'Agent, 1).λ = location(0) [] p('poll,'Agent, 1).λ = location(0)
  Interactions
      p('x, i, j).x ≈ p('poll, h, k).x                 when p('x, i, j).λ = p('poll, h, k).λ
                                                        engage p('x, i, j).x
                                                        disengage p('x, i, j).x, ⊥
      p('token, i, j).token ≈ p('poll, h, k).token     when p('token, i, j).λ = p('poll, h, k).λ
                                                        engage p('token, i, j).token
                                                        disengage p('token, i, j).token, ⊥
end
------------------------------------------------------------------------------------------
                                numNodes  ≡ the number of nodes in the system
Auxiliary definitions:             id(n)  ≡ a unique identifier generated from n
                                 next(n)  ≡ the location of the next node in the ring
                             location(n)  ≡ the location of node n
```

Figure 3: Fine-grained restructuring of the *LeaderElection* System.

3.2 Reinterpretation of the Mobile Unity syntax

Mobile UNITY considers a program to be the smallest unit of mobility. In Mobile UNITY every program has a location attribute and the modification of this attribute denotes the movement of all the code and data in the program. In CODEWEAVE we seek to introduce a finer granularity, one that allows the movement of lines of code or variables as isolated entities. 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 become units of mobility. The Mobile UNITY syntax of a system is preserved. Units generated from the system are formalized using the Mobile UNITY syntax as well. Figure 3 shows a possible mechanical transformation for the Mobile UNITY program in Figure 2 into a semantically equivalent CODEWEAVE formulation. The simple way to think about this translation process is as follows. The programmer writes a system description that looks syntactically like a Mobile UNITY program but directs a translator to interpret the code as a CODEWEAVE program. The translator generates new Mobile UNITY code and alters the **Interactions** section as needed. The result is a Mobile UNITY program that expresses precisely the Mobile UNITY semantics. The mechanics of this process are explained next.

The variables declared in Figure 2 are interpreted as *data units* in Figure 3. The data unit *p('x,'NodeValue, i)* in Figure 3, for instance, is generated from the declaration of x in program *NodeValue(i)* (see Figure 2). The name for all the units is the constant p, while the three indices after p qualify the unit; 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 [3]. This representation allows the same names to be present in different program contexts. It is possible, for instance, to define two statements labeled *poll* in two different programs of the same system. The two code units derived will have the same name (i.e., *poll*, the first index), but the second index would be instantiated to different program names. The declaration of x in program *Agent(i)* of Figure 2 does not denote storage. It is only a placeholder needed to accompany the code in the statement *poll*. Therefore, the declaration of the x in *Agent(i)* is not translated as a data unit in Figure 3. Such distinctions can be made by some annotation process (e.g., storage units are tagged by the keyword **var**) or by more sophisticated usage analysis (e.g., initialized variables must be viewed as storage units). The computation in *poll* will actually use the data in the x variable located on each node. This is made possible by the sharing mechanism defined in the **Interactions** section. Notice that a unit capturing a variable declaration also contains the corresponding initialization statement for the declared variable. The assign section of a data unit does not contain any statement as the unit only declares a variable.

In order to overcome the inability to create dynamically 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 a location $\lambda = \epsilon$. In this manner, whenever we need to duplicate or instantiate a new component, we simply change the location of some component in the ether from ϵ to an actual location.

The *code unit p('poll,'Agent, i)* of Figure 3 is generated from the statement *poll* in Figure 2. The first index of the code unit is the label of the statement. The second index is the name of the program the unit comes from, and the third is the index that allows multiple instances of the same unit. The code of the *poll* statement is part of the **assign** section. All the variables used in the statement are declared (in the **declare** section) and initialized as undefined. This initialization underlines the fact that this unit only contains code and that the variables are mere placeholders (i.e., they do not contain real values until placed in a context that provides sharing with data units).

As we want the *token* unit and the *poll* unit to move together, like in the example in Figure 2 where they move within the program context, we now have to modify the *poll* code by adding an explicit command for the movement of the unit *token* as well. The function find returns, given the first index of a unit, the last two indices for a unit present at a given location: the projection operator † is used to address the fields of the tuple returned by of find. The search can be done in two ways, by name or by name and place of definition, i.e., looking for the last two indices or only for the last index. The example shows a generic find that returns both last indices (as the system contains only one definition of the data unit *token*). The statement in the unit *poll* uses the value of the x present at each site to perform its computation. The **Interactions** section enforces the sharing of the value of x (and of *token*) between the data unit carrying the value and the placeholder variable in the code unit *p('poll,'Agent,i)*. The **Components** section in Figure 3 places the units in the same location as those of Figure 2.

The re-interpretation of the Mobile UNITY system as a fine-grained mobile system allows units to move as separate entities and code and data to be stored in different components. In the resulting model we lose the notion of scoping previously associated with the individual programs. It may appear that, since data and code are separated, their simultaneous movement and value sharing among variables have to be programmed explicitly, in the code and in the **Interactions** section. These difficulties can be avoided, however, if we introduce the notion of a container, which can be constructed dynamically, can move its entire contents of data and code units as a whole, and provides for automatic sharing of like-named variables appearing in data and code units placed inside the container. We will refer to this kind of containers as *processes* because we intend to use such components not only as dynamically structured programs but also as basic units of execution. As a matter of fact, as shown in the next section, code units will be prevented from executing whenever they reside outside the confines of a process. A process is regarded as a Mobile UNITY program and therefore formalized as *p(name, program, i)*, where the first index is the name of the process, the second is the name of the program from which the initial contained units are defined into, and the third index is the one allowing multiple instances.

The three indices defining a process are also interpreted as a location name and used in defining the location for units inside the process. While the location of a process is always the name of a host (as processes reside directly on the hosts), the location of a unit may be a string resulting from the concatenation of a host name and of three process indices they are in (if present inside a process).

With the introduction of processes we are now able to move lines of code, single variables, or complex groups of units The notion of scope introduced by processes also simplifies the sharing mechanisms between variables. Variables with the same name in the scope of the same process may be considered shared. The explicit sharing mechanism available in Mobile UNITY can be avoided by exploiting scoping. By providing a standard set of sharing rules designers do not

[3] The component token is, for instance, distinguished from its name, 'token, using the '. This permits, for example, the search for a component, given the name (see the use of find in Figure 2).

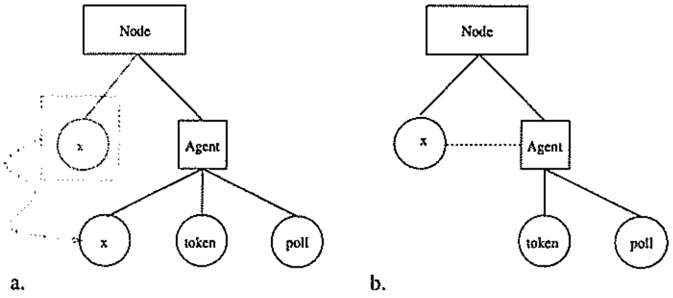

Figure 4: A node in the leader election solution with processes.

need to touch the **Interactions** section. For instance, the sharing among the *token* variable in the data unit *token* and the same variable in the code unit *poll* had to appear explicitly in Figure 3. Further refinement of our new model will allow binding by name to be established automatically between two variables within the same scope.

The explicit sharing between the variable x on the node and its counterpart in the *poll* code can also be eliminated if somehow we are able to pull within the scope of *poll* the variable x residing on the same node. This can be accomplished in two ways. First, we can explicitly move the variable x (on the node) within the scope of the process P (Figure 4.a) embodying the agent. Second, we can add a new construct called *reference*, which extends the scope of the agent process to include x without moving it (Figure 4.b). In the next section we introduce these and other mobility constructs and we revisit the leader election problem while refining the CodeWeave model.

4 Mobility Constructs

The previous section hinted at the key points of departure from Mobile Unity, and at the manner in which we will ultimately reduce a notation for fine-grained code mobility back to the essence of Mobile Unity. Central to CodeWeave 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 available in our model. In the next section, we will use Mobile Unity to provide formal semantics to these constructs.

In Section 3 we have shown a mobile agent solution for the leader election problem. We used the example to explain the re-interpretation of the Mobile Unity system in terms of units. In this section we refine the solution to the leader election problem given in Section 3 exploiting more fully than before the features and the constructs of our model: code, data and process mobility, as well as built-in scoping rules. The solution assumes 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 value with the token 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, with each voter passing 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 5, while Figure 6 shows its graphical

```
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 < numNodes :: newData(x, NodeDefinition, node(i), id(i))⟩
    [] newData(token, TokenDefinition, node(0), ⊥)
    [] newCode(poll, PollActions, node(0))
    [] newProcess(voter, VoterActions, node(0), ACTIVE)
end
--------------------------------------------------------------------------------
                                     here       ≡  λ
Auxiliary definitions:           thisNode       ≡  head(λ)
                                  next(n)       ≡  the node following n in the ring
```

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

representation. The specification uses the fine-grained mobile code constructs of our model. The upper part of the specification contains three program definitions. The programs in Figure 5 are not indexed with an i unlike in Figure 2 as, with the introduction of processes, programs are only units of definition and they are not instantiated.

NodeDefinition specifies a single data unit x associated with a node. The type declaration for this integer variable is prepended by the keyword **var** which characterizes the variable as a data unit. The initialization of the variable is not defined here: the predicate **newData**, used in the **Components** section will provide the initial value to be assigned when the unit is instantiated. In this way we allow different instances of the same unit to be initialized with different values. The first parameter of **newData** is the name of the variable, the second is the name of the program in which the unit is defined, the third is the location where the unit has to be placed, and the fourth is the initial value. The program *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 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

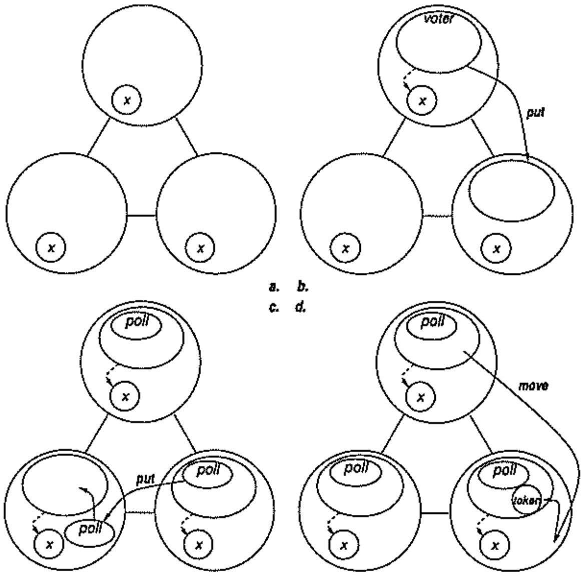

Figure 6: (a) Each node in the ring holds an identifier x. (b) The voter process establishes a reference to the local identifier and replicates itself on the next node. (c) The polling policy continues to be replicated on the next node and to be absorbed into the voter process. (d) The token travels from one node to the next.

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. This 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 operates 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), 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*, which is the first index.

As will become 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. Like most of the operations provided in our model, the **put** operation is defined on any component, 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 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 macro **newData** to indicate the creation of 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 this data unit as a result of the operation.

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 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. Note how, similarly to what was described for **put**, only the name of the data unit x is specified, while its indices are 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 CODEWEAVE 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 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. 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 describing CODEWEAVE there is a presumption that the **Interactions** section is used to define the semantics of the mobility constructs and, once the job is done, it can no longer be modified. A designer employing CODEWEAVE is indeed limited to the set of constructs we provide. However, the reader is reminded that the purpose of this paper is also to describe a general strategy for constructing models like CODEWEAVE. This means that a middleware designer would have to be in the position to modify the **Interactions** section when new constructs are desired.

In the remainder of the section we consider in turn the topics of scoping, statement scheduling, mobility constructs, and creation predicates. 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 ($c_{ij}.\lambda$), request field ($c_{i,j}.\rho$) designed to hold mobility commands the system is expected to execute on its behalf, and type ($c_{i,j}.\tau \in \{$DATAUNIT, CODEUNIT, PROCESS$\}$).
- Each process $q_{i,j}$ is also characterized by an implicitly specified set of contained units (those whose λ value refers to the process), a set of referenced units ($q_{i,j}.\gamma$), and the process activation status ($q_{i,j}.\omega \in \{$ACTIVE, INACTIVE, 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 name (e.g., c) rather than to 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

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

Figure 7: Specification of the functions find and exists.

$$
\begin{array}{llll}
 & u_{i,h}.x \approx w_{j,k}.x & \mathbf{when}\ u_{i,h}.\tau = \text{DATAUNIT} \wedge w_{j,k}.\tau = \text{CODEUNIT}\ \wedge & \\
 & & \quad (u_{i,h}.\lambda = w_{j,k}.\lambda \neq \mathsf{head}(u_{i,h}.\lambda)\ \vee & \\
(1) & & \quad (\mathsf{sharing}(u_{i,h}, w_{j,k}) \wedge \mathsf{head}(u_{i,h}.\lambda) = \mathsf{head}(w_{j,k}.\lambda))) & \\
 & & \mathbf{engage}\ u_{i,h}.x & \\
 & & \mathbf{disengage}\ u_{i,h}.x,\ \perp & \\
 & u_{i,h}.x \approx w_{j,k}.x & \mathbf{when}\ u_{i,h}.\tau = w_{j,k}.\tau = \text{DATAUNIT}\ \wedge & \\
(2) & & \quad (u_{i,j}.\lambda = w_{j,k}.\lambda \neq \mathsf{head}(w_{j,k}.\lambda)\ \vee & \\
 & & \quad (\mathsf{sharing}(u_{i,h}, w_{j,k}) \wedge \mathsf{head}(u_{i,h}.\lambda) = \mathsf{head}(w_{j,k}.\lambda))) & \\
 & & \mathbf{engage}\ \mathsf{max}(u_{i,h}.x, w_{j,k}.x) & \\
 & \mathbf{inhibit}\ u_{i,h}.s & \mathbf{when}\ u_{i,h}.\tau = \text{CODEUNIT}\ \wedge & \\
(3) & & \quad (\langle \forall p, m, n : p_{m,n}.\tau = \text{PROCESS} \wedge (\mathsf{containedIn}(u_{i,h}, p_{m,n})\ \vee & \\
 & & \quad \mathsf{referencedBy}(u_{i,h}, p_{m,n})) :: p_{m,n}.\omega \neq \text{ACTIVE}\rangle \vee & \\
 & & \quad \langle \exists x :: u_{i,h}.x = \perp\rangle) &
\end{array}
$$

Auxiliary definitions:

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

Figure 8: Establishing bindings among units using transient variable sharing and statement inhibition.

defined in Figure 7. 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 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 5), and the third is the instance discriminator.

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 8 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[4] $u_{i,h}.x$ 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 sharing tells if two units are

[4] The formulae in Figure 8 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.

either contained in or referenced by the same process, i.e., the units are in the same scope. In turn, sharing uses the functions $\mathsf{containedIn}(v_{j,k}, u_{i,h})$, that determines whether $v_{j,k}$ 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 different semantics for value reconciliation. As no **disengage** is specified, the variables retain the values they had before the **when** condition became false.

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 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 8 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.

5.3 Mobility Constructs

The designer views the **move** construct as a means 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:

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

is actually a special instance of the more general form in which the identity of the component is already known. One can simply determine the identity by employing the function find as in[5]

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

If multiple instances of the same component exist, one is selected[6]. In order to explore the manner in which we assign semantics to the mobility constructs of CODEWEAVE we focus our presentation on the general form of the construct. Moreover, we assume that the component in question is a process (q, 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 9):

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

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, once enabled, execute to fixed point before the system is allowed to take any other action. 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 [7]. Actually transient sharing is ultimately defined using reactive statements [13], but this is outside the scope of this paper.

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

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

[7] Non-reactive statements are executed in a fair interleaving manner.

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

Auxiliary definitions:

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

Figure 9: Mapping mobility constructs to Mobile UNITY statements.

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

Return values for $\mathcal{F}$:

$$\begin{aligned}
\mathcal{F}(\text{MOVE}, u, i, v, n, m, (h,l)) &= \quad (m, l \circ (u,i,h))\\
\mathcal{F}(\text{PUT}, u, i, v, n, m, (k,l)) &= \quad (\mathsf{getid}(v,n), l \circ (u,i,k))\\
\mathcal{F}(\text{CLONE}, u, i, v, n, m, (k,l)) &= \quad (\mathsf{getid}(v,n), l \circ (u,i,k))\\
\mathcal{F}(\text{DESTROY}, u, i, v, n, m, ()) &= \quad ()
\end{aligned}$$

Figure 10: Modeling the actions of the run-time support.

The first reaction taking place transfers the command **move** to the process q. The result is that $q_{i,j}.\rho$ is assigned the request with a propagation status of EXEC:

$$q_{i,j}.\rho := (\text{EXEC}, \text{MOVE}, q, i, (j,l))$$

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 10.

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

$$q_{i,j}.\rho := (\text{MOVE}, (j,l))$$

while simultaneously propagating the command to the contained units (see Figure 10), e.g.,

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

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

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

The last step is the change in location of each of the units (Figure 11). 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.,

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

$$\begin{aligned}&u_{i,h}.\lambda := l \ \ \textbf{if}\ (u_{i,h}.\tau = \textsc{process} \Rightarrow l = \mathsf{head}(l)) \wedge u_{i,h}.\omega \neq \textsc{terminated} \wedge u_{i,h}.\lambda \neq \epsilon \parallel\\ &u_{i,h}.\rho := \bot\ \textbf{reacts-to}\ u_{i,h}.\rho = (\textsc{move}, (h, l))\end{aligned} \tag{6}$$

$$\begin{aligned}&u_{i,k}.\lambda, u_{i,k}.\omega := l, u_{i,h}.\omega \ \ \textbf{if}\ (u_{i,h}.\tau = \textsc{process} \Rightarrow l = \mathsf{head}(l)) \wedge u_{i,h}.\lambda \neq \epsilon \parallel\\ &u_{i,h}.\rho := \bot\ \textbf{reacts-to}\ u_{i,h}.\rho = (\textsc{put}, (k, l))\end{aligned} \tag{7}$$

$$\begin{aligned}&u_{i,k}.\lambda, u_{i,k}.\omega := l, u_{i,h}.\omega \ \ \textbf{if}\ (u_{i,h}.\tau = \textsc{process} \Rightarrow l = \mathsf{head}(l)) \wedge u_{i,h}.\lambda \neq \epsilon \parallel\\ &u_{i,h}.\rho := \bot \parallel \langle \forall x :: u_{i,k}.x := u_{i,h}.x\rangle\ \textbf{reacts-to}\ u_{i,h}.\rho = (\textsc{clone}, (k, l))\end{aligned} \tag{8}$$

$$u_{i,h}.\lambda := \bot \ \ \textbf{if}\ u_{i,h}.\lambda \neq \epsilon \parallel u_{i,h}.\rho := \bot\ \textbf{reacts-to}\ u_{i,h}.\rho = (\textsc{destroy}, ()) \tag{9}$$

$$\begin{aligned}&u_{i,h}.\omega := \textsc{active} \ \ \textbf{if}\ u_{i,h}.\omega = \textsc{inactive} \wedge u_{i,h}.\tau = \textsc{process} \wedge u_{i,h}.\lambda \neq \epsilon \parallel u_{i,h}.\rho = \bot\\ &\textbf{reacts-to}\ u_{i,h}.\rho = (\textsc{activate}, ())\end{aligned} \tag{10}$$

$$\begin{aligned}&u_{i,h}.\omega := \textsc{inactive} \ \ \textbf{if}\ u_{i,h}.\omega = \textsc{active} \wedge u_{i,h}.\tau = \textsc{process} \wedge u_{i,h}.\lambda \neq \epsilon \parallel u_{i,h}.\rho := \bot\\ &\textbf{reacts-to}\ u_{i,h}.\rho = (\textsc{deactivate}, ())\end{aligned} \tag{11}$$

$$\begin{aligned}&u_{i,h}.\omega := \textsc{terminated} \ \ \textbf{if}\ u_{i,h}.\omega \neq \textsc{terminated} \wedge u_{i,h}.\tau = \textsc{process} \wedge u_{i,h}.\lambda \neq \epsilon \parallel\\ &u_{i,h}.\rho := \bot\ \textbf{reacts-to}\ u_{i,h}.\rho = (\textsc{terminate}, ())\end{aligned} \tag{12}$$

$$\begin{aligned}&u_{i,h}.\lambda := l \ \ \textbf{if}\ u_{i,h}.\tau = \textsc{process} \Rightarrow l = \mathsf{head}(l) \parallel u_{i,h}.\rho := \bot\\ &\textbf{reacts-to}\ u_{i,h}.\rho = (\textsc{new}, l)\end{aligned} \tag{13}$$

$$\begin{aligned}&u_{i,h}.\gamma := u_{i,h}.\gamma \cup \{(v, j, k)\} \ \ \textbf{if}\ v_{j,k}.\tau \neq \textsc{process} \wedge u_{i,h}.\tau = \textsc{process} \wedge u_{i,h}.\lambda \neq \epsilon \wedge\\ &v_{j,k}.\lambda \neq \epsilon \parallel u_{i,h}.\rho = \bot\ \textbf{reacts-to}\ u_{i,h}.\rho = (\textsc{reference}, (v, j, k))\end{aligned} \tag{14}$$

$$u_{i,h}.\gamma := u_{i,h}.\gamma \setminus \{(v, j, k)\} \parallel u_{i,h}.\rho := \bot\ \textbf{reacts-to}\ u_{i,h}.\rho = (\textsc{unreference}, (v, j, k)) \tag{15}$$

Figure 11: Migrating components.

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 10 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 9 and 11.

5.4 Creation Predicates

The three macros **newData**, **newCode**, **newProcess** are used in Figure 5 for the instantiation of new components. **newData** is defined in two forms. The first allows the setting of the initial value as a parameter (i.e., v)[8]. The second uses the initial value defined in the program. The constructs used in Figure 5 are special instances of more general form: for instance, $\textbf{newData}(u, n, l, v)$ is a special form of $\textbf{newData}(u, n, \mathsf{getid}(u, n), l, v)$. The function getid (shown in Figure 9) has two parameters in this case as we know one of the indices (i.e., the program name n). Figure 12 contains the semantics of these predicates. **newData** states that a new data unit is located at location l and that setting the initial value for its variable is v. **newCode** states that a new code unit is located at location l. The predicate **newProcess** locates a process at location l, with status s. The predicates **newData** and **newCode** are used to define the initial location of all the units that have to be inside the process. The auxiliary functions dataDefined and codeDefined check that the unit u' is defined in program n of the specification.

6 Enhancing the Model with Scoping

In CodeWeave, as outlined so far, processes may only contain data and code units, i.e., processes cannot be nested into each other generating a hierarchy. Nested processes however may offer interesting properties in terms of scoping and added flexibility in the organization of resources and systems. Nested structures are not common in mobile code systems. Most of the mobile agent architectures [21] provide a flat model where agents (i.e., the execution units) move

[8] The **newData** predicate is used in the **Components** section in order to define the instantiation of new data units. The implicit quantification over the variables used in the **Components** section is generally restricted to some proper range. In case of variable names the range is set to the names appearing in the unit (i.e., the case of x).

$$
\begin{array}{ll}
\mathbf{newData}(u,n,k,l,v) & \equiv u_{n,k}.\lambda = l \wedge u_{n,k}.x = v \\
\mathbf{newData}(u,n,k,l) & \equiv u_{n,k}.\lambda = l \wedge u_{n,k}.x = \mathsf{initial}(n,x) \\
\mathbf{newCode}(u,n,k,l) & \equiv u_{n,k}.\lambda = l \\
\mathbf{newProcess}(u,n,k,l,s) & \equiv u_{n,k}.\lambda = l \wedge u_{n,k}.\omega = s \wedge \\
 & \langle \forall \bar{u} : \mathsf{dataDefined}(\bar{u},n) :: \mathbf{newData}(\bar{u},n,\mathsf{getid}(\bar{u},n), l \circ (u,n,k))\rangle \wedge \\
 & \langle \forall \bar{u} : \mathsf{codeDefined}(\bar{u},n) :: \mathbf{newCode}(\bar{u},n,\mathsf{getid}(\bar{u},n), l \circ (u,n,k))\rangle
\end{array}
$$

Figure 12: Constructs for the instantiation of components.

from host to host. An exception is Telescript [19] in which *places* (i.e., executing units) are arranged in a hierarchical topology, and agents move from place to place. Several models developed for mobile systems use nested environments. Ambient and Seal calculi [3, 18], for instance, model nested scope with a process algebra based formalism (more details and comparisons can be found in Section 8) and they exploit the nesting structure for security purposes.

In this section, we present an extension of the basic CodeWeave by allowing processes to contain other processes thus permitting the formalization of sophisticated hierarchies of structures and exploiting the natural potential of CodeWeave. The extension, in fact, turns out to be a natural step, merely a simple refinement of the notion of location. A location is no longer the concatenation of at most two names (i.e., the host name and, if needed, the process name), but an arbitrarily long concatenation of names, reflecting process nesting.

A lexical scoping mechanism is established among the processes. Every nested process acts as a block structured context: all the data and code in the block are considered *local* to the process and cannot be accessed from the upper blocks. However, the inner blocks can access the content of the outer blocks. The binding mechanism presented in Table 8 is readily extended to accommodate the new hierarchical process structure. Figure 13 shows an example of binding in the tree. Variables with the same name in the scope of the same process (like x in the units v and u) are bound and share the same value (as in plain CodeWeave). As one might expect, structural changes due to mobility of code fragments lead to corresponding changes in scope and data access. Let us consider again Figure 13. Notice that the data unit w contains a declaration for x. While the data unit v is present, the variable x in code unit u is bound to the declaration of x in v. If unit v moves away, the x in code unit v becomes bound to x in w. In general, the binding mechanism binds a variable in a code unit to the "closest" declaration for that variable found in the path to the root of the tree (i.e. the host).

The access to referenced units must also be adjusted for use with the nested scope. We constrain the scope of the referenced unit only to the peer units in the referencing process (i.e., among units that are directly subordinate to the same process). However it is possible to access a referenced unit from lower levels in the tree. Let us consider, for instance, Figure 14.a: the variable x in the code unit v is bound to the declaration of x in data unit z (we assume that the unit z and process P are on the same host) as the data unit is referenced by process P. The declaration of x in z, however, cannot be bound to the x in code unit u as this is at a lower level (i.e., it is not a peer unit) even though process Q references z. In some cases the designer may want to let the x in u to be bound to the x in z: to do this she can either put another reference to x directly from process Q or exploit the reference from process S and introduce a new data unit w declaring x at the peer level of z in process S. In this case the binding mechanism allows the sharing between the two declarations of x in data units in the same scope (i.e., w and z), and w is the closest data

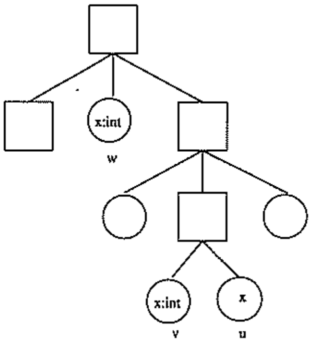

Figure 13: Scoping in the enhanced model: units containing variables with declaration are data units. Units containing variables without declaration are code units.

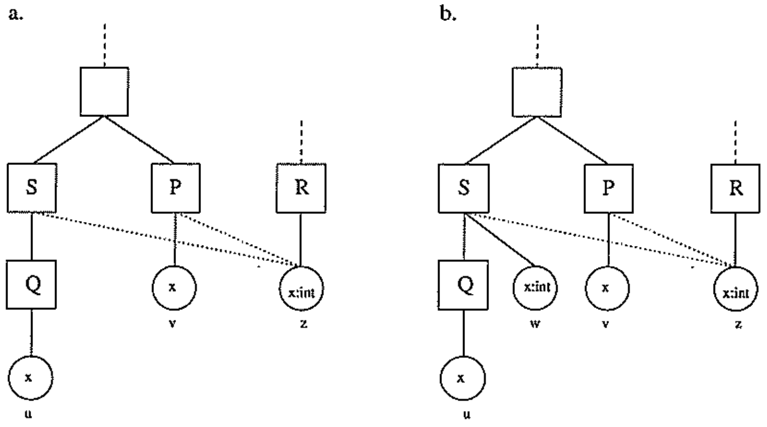

Figure 14: Referencing in the hierarchical model.

unit declaring x with respect to unit u, then the binding is established between the two x variables (see Figure 14.b).

After having extended the scoping strategy of CodeWeave it is possible to add new operations that exploit the nested scope. For example an operation can be added to be able to constrain the access to the content of a unit only to peer units. In plain CodeWeave the access strategy is a hierarchical access, where lower level units may be bound to an upper level unit (in case it contains the closest declaration for a certain variable). The new operation could constrain the accessibility of a unit only to peer units in order, for instance, to hide the content of a unit from lower levels. In order to do so units should have an attribute indicating their access type: the automatic sharing mechanism looking for the closest instance of a variable would have to consider this attribute in computing it. For instance, let us consider again Figure 14.b in this context: if the unit w had access right set to PeerAccess, unit u would never be able to share the value of its variable, as it is not a peer unit.

The hierarchical model requires little changes in the semantics of the constructs defined in Table 11. As the model in general is enhanced only by modifying the notion of location, the changes required in the semantics are minimal. The operations **move**, **put**, **clone** are now more flexible as they can place a process inside another process, not only on hosts. Let us consider, for instance, the **move** operation (6) in Table 11. The check $u_{i,h}.\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l)$ ensures that a process is only be moved on a host, and not inside another process (i.e., l is a host location). In the modified **move** for extended CodeWeave this check disappears and the new formalization of **move** is:
$u_{i,h}.\lambda := l \;\text{ if } (u_{i,h}.\omega \neq \text{TERMINATED} \wedge u_{i,h}.\lambda \neq \epsilon) \parallel u_{i,h}.\rho := \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{MOVE}, (l))$
In addition, the operations **activate**, **deactivate**, and **terminate** have to be propagated to all the child processes of the input process. For this reason the function $\mathcal{F}$ has to be defined also for these constructs (it simply returns the () value).

7 Reality Check

In this section we challenge CodeWeave by arguing about its expressive power in relation to common patterns of code relocation, and by investigating the impact of the model assumptions on a real implementation. Finally, we highlight some opportunities for language design inspired by the constructs we introduced in our model.

7.1 Expressiveness

The goal of our model is to offer a formal tool that can be used to specify the semantics of operations and mechanisms provided by mobile code systems. The characteristics of the system specified depend heavily on the particular choice of features, mechanisms, policies the designer wants to introduce in it. In this respect, we sought out the fundamental abstractions needed for the modeling task, deliberately leaving to the designer the chore of composing (or extending) these abstractions to define details and policies that are specific of a particular design strategy.

Notably, this includes mechanisms often found in mobile code systems, like dynamic linking. In general, dynamic linking takes place when, during the execution of a program, the runtime support encounters a name for which a definition is not currently known. The runtime support must then somehow resolve the name, by retrieving and linking the corresponding code fragment into the address space of the requesting execution unit. In mobile code systems, the code retrieved is not necessarily present on the node where the runtime support resides, and thus gets transferred during the linking process. A dynamic linking mechanism that provides a high degree of programmability is embodied in the Java class loader, and similar mechanisms exist in Tcl derivatives [9]. However, the details concerned with the policies used for name resolving vary considerably.

CODEWEAVE provides all the necessary building blocks for modeling dynamic linking: a way to request the retrieval of a new code fragment (the **new** operation), a notion of address space (the process concept), a way to determine whether a given fragment is in the address space (the functions find and exists), and relocation primitives to specify the actual code transfer. The details about the name resolution policies, however, are left to the language designer that can specify them by appropriately strengthening the guards of the base constructs or add new statements in the **Interactions** section.

As an example, let us consider the default policy of the Java class loader. When a class name must be resolved, the name is searched among the classes already loaded and among the bytecode files present in the local file system. If the class is found, the class is loaded, otherwise an exception is raised. These policy can be expressed by redefining the **new** construct as follows:

$$\begin{aligned}\mathbf{new}(u,n,\lambda) \equiv \mathbf{reference}(u,n,\lambda) &\quad \textbf{if } \mathsf{exists}(u,n,\lambda)\\ \sim \mathbf{put}(u,n,\lambda) &\quad \textbf{if } \mathsf{exists}(u,n,\mathsf{head}(\lambda))\end{aligned}$$

where we did not explicitly model the generation of an exception. This policy for class loading can be further refined to accommodate, for instance, the one chosen by Web browsers. In this case, **new** can be redefined as follows

$$\begin{aligned}\mathbf{new}(u,n,\lambda) \equiv \mathbf{reference}(u,n,\lambda) &\quad \textbf{if } \mathsf{exists}(u,n,\lambda)\\ \sim \mathbf{put}(u,n,\lambda) &\quad \textbf{if } \mathsf{exists}(u,n,\mathsf{head}(\lambda))\\ \sim \mathbf{put}(u,n,\lambda) &\quad \textbf{if } \mathsf{exists}(u,n,\lambda_{URL})\end{aligned}$$

to express the fact that when the class is not found locally, a copy of it gets retrieved from the host that provided the Web page, indicated above as λ_{URL}. Similar considerations hold for the class loader managing the retrieval of the classes for parameters in a Java Remote Method Invocation [10].

Clearly, the building blocks provided by our model can be employed to represent many other relevant parts of mobile code systems. For instance, in our leader election example we showed, albeit in a simplified fashion, that by substituting the code that performs the comparison we could deal with the caching and versioning of classes.

It is worth noting at this point that our fine-grained model subsumes, rather than replace, the more coarse-grained perspective where the units of execution and of mobility coincide. For instance, mobile agent systems are still modeled naturally by using the process abstraction. Thus, for instance, it is fairly straightforward to model the semantics of the `dispatch` and `retract` operation, which perform migration of an aglet in the Aglets framework [12], and even aspects related to activation and deactivation of aglets. Furthermore, the ability to represent nested processes enable even the modeling of complex structures built by sites, places, and agents like those introduced in Telescript [19], similarly to what Ambients [3] provide.

The availability of fine-grained mechanisms opens up the opportunity for a design style where mobile agents are represented with increasing levels of refinement. Existing systems almost never allow a mobile agent to move with all its code, due to performance reasons. Different systems employ different strategies, ranging from a completely static code relocation strategy that separates at configuration time the code that must be carried by the agent from the one that remains on the source host (like in Aglets [12]), to completely dynamic and under the control of the programmer (like in μCODE [16]). Hence, our approach allows modeling of a mobile agent application at different levels of detail, starting with a high level description where the mobile components and their behavior is specified, and then refining this view by using our constructs to specify precisely which constituents of the agents are allowed to move and which are not.

The introduction of the reference concept brings additional expressive power to CODEWEAVE. The rationale for introducing references was motivated by the need to introduce the means by which the same component is shared among multiple processes and, consequently, to be able to express directly the design strategies ruling how bindings are established or severed upon migration, as discussed for instance in [6]. Analogously to the taxonomy presented in [6], there are at least three kinds of references that can be exploited:

- *Local references.* They are specific to a site and get cut off by migration. This is the solution typically adopted by mobile code systems to deal with system-dependent resources like files or sockets.

- *Contextual references.* They are re-adjusted upon arrival at the destination site through some matching process. Typically, the matching is performed on the type of the object bound by the reference, to support *ubiquitous resources* [6].
- *Permanent references.* They are global, absolute, and survive movement. This is the case of *network references* à la Obliq [2].

In this paper, we constrained references to be local, to simplify the presentation of the model and also because we felt that the expressiveness of this kind of reference was already capturing many of the issues we wanted to investigate. For instance, local references allow us to express what happens upon migration to the references of a process, through our binding mechanism. Moreover, they allow us to rule the bindings between units that belong to a process hence, to model dynamic linking and memory allocation. Nevertheless, we acknowledge that the reference concept is more far reaching than we exploited in this paper, and we need to point out that the other two kinds of references, contextual and permanent, can be modeled straightforwardly with simple modifications to the **Interactions** section.

It is interesting to note that permanent references are very common in popular middleware like Java/RMI or CORBA, where they are established among distributed objects. This is only one of the aspects that show how CodeWeave, although developed for the purpose of specifying mobility, actually bridges this research area to traditional distributed systems research. For instance, in Java/RMI the programmer can specify whether the value of a parameter in a remote method call must be passed by copy or by reference, and let the server object download dynamically the code of the corresponding object in the case when it is a subtype of the one provided in the signature. Furthermore, remote objects serving method calls can get activated and deactivated on demand, to reduce computational load at the server host. These aspects, which exhibit a mixture of traditional distributed computing and code mobility, are crucial to building correct applications in Java/RMI and, although they have not yet been modeled formally with CodeWeave, they seem to match its basic concepts. For instance, the server thread serving method calls can be represented by a process; parameter passing can be expressed by performing a copy operation followed by a move or by establishing a reference or a data unit, representing the parameter value; code downloading can be represented by migration of a code unit into the server process; activation and deactivation can be represented using the activation status of the server process; additional rules describing the semantics of RMI can be placed in a specialized **Interactions** section.

7.2 Opportunities

The goal of our investigation is not only to come up with a model that is finer-grained than existing ones, and thus more apt to represent mobile code besides mobile agents, but also to uncover novel perspectives suggested by the process of formalizing the finest grains of mobility possible in a programming language. Hence, in the remainder of this section we take a speculative approach and envision ways, hitherto unexplored, to think about logical mobility and distributed systems.

Insofar we always assumed that moving a data or code unit in our model does not necessarily mean to do that in the language chosen for implementation, e.g., Java. The key point is that we are able to represent the movement of a unit of mobility (e.g., a Java object or class) that has a finer grain than the unit of execution (e.g., a Java thread). Nevertheless, the ability to move a single statement or variable in a real programming language is an intriguing perspective per se, and a one that may become feasible as the diffusion of scripting languages grows in popularity. For instance, it is very easy to send around XML tags (the analogous of program instructions) that can dynamically "plug-in" into an XML script already executing at the destination [4]. This may amplify the enhancements in flexibility and customizability brought by mobile code in the design of a distributed architecture. An even wilder scenario is the one where it is possible not only to add or substitute programming language statements that conform to the semantics of the language, but also to extend such semantics dynamically by migrating statements along with a representation of the semantics of the constructs they use. As recently proposed at a conference [8], this could open up an economic model where the concept of software component includes not only application code or libraries, but even the very constituents of a programming language.

Another interesting opportunity is grounded in the binding mechanism that rules execution of units within a process. In our model, a statement can actually execute only if it is within a process, and if all of its variables are bound to a corresponding data unit. This represents the intuitive notion that a program executes within a process only if the code is there and some memory has been allocated for the variables. In languages that provide remote dynamic linking, it is always a code fragment that gets dynamically downloaded into a running program. However, the symmetry between data and code units in our model suggests a complementary approach where not only the code gets dynamically downloaded, but also the data. Thus, for instance, much like a class loader is invoked to resolve the name of a class during execution of a program, similarly an "object loader" could be exploited to bring an object to be co-located with the program and thus enable resumption of the computation. Another, even wilder, follow-up on this idea is an alternative computation model where code and data are not necessarily brought together to enable a program to proceed execution, rather it is the program itself (or a whole swarm of them, to enhance probability of success, like in the model discussed in [20]) that migrates and proceeds to execute based on the set of components currently bound to it. This way, a program is like an empty "pallet" wandering on the net and occasionally performing some computation based on the pieces that fill its holes at a given point.

Finally, an aspect about our model that we left out is verification. CodeWeave is based on Mobile Unity, and is specialized without requiring modifications of the core semantics. Hence, our model inherits also the Mobile Unity temporal logic, an extension of the original Unity logic, and that has been already applied to the problem of verifying properties for mobile code systems [17]. The fine-grained perspective adopted in this paper, however, enables not only to reason about the location of mobile programs, as in [17], but also about the locations of their constituents. Hence, verification can be exploited not only to prove correctness properties of the system, but it can be exploited also to optimize the placement of components, placing them only on the nodes where they will be needed. For instance, the code of a mobile agent could be written in such a way that it does not need to carry with it a given class, because a formal proof has been developed such that, for the given system in the given state, the class has already been cached there. Clearly, this approach potentially enables bandwidth and storage savings, and is particularly amenable for use in environments where resources are scarce, e.g., wireless computing with PDAs.

8 Related Work

Many formalisms have been used and devised in order to deal with issues related with logical mobility. In particular process algebra is one of the most successfully used theory in this respect. π-calculus [14] is based on the notion of *channels.* Channels can move along other channels and the movement of computations is represented as the movement of channels that refer to them. π-calculus has also been extended in several directions in order to overcome for instance the lack of notion of location (join calculus [5]), or to add asynchronous mechanisms to the model [1]. In Klaim [15] π-calculus is combined with a tuple-space based language à la Linda [7] that provides the framework for formalizing the concept of location.

The granularity of movement in process algebra is based on the notion of *process* (i.e., computation code), the actual unit of mobility. Extensions of this idea have been devised: Ambient calculus [3] and Seal calculus [18] introduce an explicit notion of *environment* (*ambients* in Ambient calculus and *seal* in Seal calculus). Environments define the computation scope. The unit of mobility is the environment that can carry computations and other environments while moving. Environments are units of execution and mobility at the same time as they rely on the notion of process common to the process-algebra languages. Our approach focuses on the decoupling between the unit of mobility (i.e., the unit) and the unit of execution (i.e., the process) in order to separate mobility related issues from more execution related issues like activation/deactivation status. Actually, we provide specific operations (**activate**, **deactivate**, **terminate**) to act on the execution state of processes.

Security is an important aspect when dealing with highly dynamic systems involving mobile code. Ambient calculus and Seal calculus are based on a "step by step" movement mechanism where components move from one domain to another crossing one boundary at a time. Security features are based on this constrained mobility mechanism. Klaim [15] relies on a type system added on top of the model in order to perform static checks on access rights and operations of the system components. We can see possible future work in this direction especially in terms of resource access constrains: a first step could be adding operators constraining the visibility of units (as we discussed in Section 6).

In Ambient calculus every environment can decide to move with its content whenever it wants to, while Seal calculus prohibits this behavior for security reasons. In Seal calculus the environment decides on the movement of the contained entities. In our model the **move** construct is invoked in a code unit, that, to be executed should be part of an active process. The move can act on other entities, on the containing process, and on the code unit itself. We do not want to constrain the model by assuming for instance that the move can only act on inactive processes, it can only move local entities, or it cannot act on itself. Such constrains can be readily formalized in CodeWeave as needed.

One of the aims of our paper is to provide basic operations for mobile code systems. All the formalisms considered provide more or less explicit mobility constructs: in π-calculus mobility is much more implicit than in Ambient calculus, though. Ambient calculus also provides an *open* operation able to dissolve the boundaries of an environment. Seal calculus does not provide that operation for security reasons. We do not provide an *open* operation as it can be built on top of the basic primitives of the model. In fact an open can be formalized as a movement (i.e., *move*) of all the contained entities of a process and of a *destroy* of the process itself. We specify basic movement operations for entities of different granularity (i.e., data, code, and processes). The two cloning operations (i.e. *put* and *clone*) differ depending on initialization value of the copied entity. All the process algebra based models exploit the replication construct to formalize cloning. No notion of initialization values is provided.

Resource handling happens to be a fundamental issue in mobile code setting: we provide an operation to establish references to resources. Ambient and Seal calculus rely exclusively on the notion of scoping defined by the environment hierarchy for handling resources sharing. In CODEWEAVE processes act as containers and scope boundaries, however an explicit operation is provided (i.e., *reference*) to allow more general sharing that can be modeled by the designer.

9 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. 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 in 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 CODEWEAVE 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, e.g., to the development of novel mobility constructs and applications in which code changes are frequent.

References

[1] R. Amadio. An Asynchronous Model of Locality, Failure, and Process Mobility. In *Proc. of the 2nd Int. Conf. on Coordination Models and Languages (COORDINATION '97)*, volume 1282 of *LNCS*. Springer, 1997.

[2] L. Cardelli. A Language with Distributed Scope. In *Proc. 22nd 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] W. Emmerich, C. Mascolo, and A. Finkelstein. Incremental Code Mobility with XML. Technical Report 99-95, University College London, October 1999. Submitted for publication.

[5] C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In *Proc. 7th Int. Conf. on Concurrency Theory (CONCUR '96)*, volume 1119 of *LNCS*. Springer, 1996.

[6] A. Fuggetta, G.P. Picco, and G. Vigna. Understanding Code Mobility. *IEEE Trans. on Software Engineering*, 24(5), 1998.

[7] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[8] G. Glass. Agents and Internet Component Technology. Invited talk at *3rd Int. Conf. on Autonomous Agents (Agents'99)*, May 1999.

[9] R. Gray. Agent Tcl: A transportable agent system. In *Proc. of the CIKM Workshop on Intelligent Information Agents*, 1995.

[10] Javasoft. *Java Remote Method Invocation Specification*, Revision 1.50, JDK 1.2 edition, October 1998.

[11] J. Kiniry and D. Zimmerman. A Hands-On Look at Java Mobile Agents. *IEEE Internet Computing*, 1(4), 1997.

[12] D. B. Lange and M. Oshima, editors. *Programming and Deploying Java^{TM} Mobile Agents with Aglets^{TM}*. Addison-Wesley, 1998.

[13] P.J. McCann and G.-C. Roman. Compositional Programming Abstractions for Mobile Computing. *IEEE Trans. on Software Engineering*, 24(2), 1998.

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

[15] 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.

[16] G. Picco. μ-Code: A Lightweight and Flexible Mobile Code Toolkit. In K. Rothermel and F. Hohl, editors, *Proc. 2nd Int. Workshop on Mobile Agents*, volume 1477 of *LNCS*, pages 26–37, Stuttgart, Germany, 1998. Springer.

[17] G.P. Picco, G.-C. Roman, and P. McCann. Expressing Code Mobility in Mobile UNITY. In *Proc. 6th European Software Eng. Conf. (ESEC/FSE'97)*, volume 1301 of *LNCS*, pages 500–518. Springer, 1997.

[18] J. Vitek and G. Castagna. Seal: A Framework for Secure Mobile Computations. In *Internet Programming Languages*, LNCS. Springer, 1999.

[19] J. White. Telescript Technology: Mobile Agents. In J. Bradshaw, editor, *Software Agents.* AAAI Press/MIT Press, 1996.

[20] T. White and B. Pagurek. Towards Multi-Swarm Problem Problem Solving in Networks. In *Proc. 3rd Int. Conf. on Mulit-Agent Systems (ICMAS'98)*, pages 333–340, July 1998.

[21] D. Wong, N. Paciorek, and D. Moore. Java-based Mobile Agents. *Communications of the ACM*, 42(3):92–102, 1999.