CODEWEAVE: Exploring Fine-Grained Mobility of Code

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

Abstract

This paper is concerned with an abstract exploration of code mobility constructs designed for use in settings where the level of granularity associated with the mobile units exhibits significant variability. Units of mobility that are both finer and coarser grained than the unit of execution are examined. To accomplish this, we take the extreme view that every line of code and every variable declaration are potentially mobile, i.e., it may be duplicated or moved from one program context to another on the same host or across the network. We also assume that complex code assemblies may move with equal ease. The result is CODEWEAVE, a model that shows how to develop new forms of code mobility, assign them precise meaning, and facilitate formal verification of programs employing them. The design of CODEWEAVE relies greatly on Mobile UNITY, a notation and proof logic for mobile computing. Mobile UNITY offers a computational milieu for examining a wide range of constructs and semantic alternatives in a clean abstract setting, i.e., unconstrained by compilation and performance considerations traditionally associated with programming language design. Ultimately, the notation offered by CODEWEAVE is given exact semantic definition by means of a direct mapping to the underlying Mobile UNITY model. The abstract and formal treatment of code mobility offered by CODEWEAVE establishes a technical foundation for examining competing proposals and for subsequent integration of some of the mobility constructs both at the language level and within middleware for mobility.

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 travel 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 [12, 10] 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 [20] 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 [15] 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 [14] 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.

*C. Mascolo is with Dept. of Computer Science, University College London, Gower Street, London WC1E 6BT, UK. E-mail: c.mascolo@cs.ucl.ac.uk. G.P. Picco is with Dip. di Elettronica e Informazione, Politecnico di Milano, P.za Leonardo da Vinci 32, 20133 Milano, Italy. E-mail: picco@elet.polimi.it. G.-C. Roman is with Dept. of Computer Science, Washington University, Campus Box 1045, One Brookings Drive, Saint Louis, MO 63130-4899, USA. E-mail: roman@cs.wustl.edu.

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 and middleware 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 following questions: 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? What new and still more complex constructs can be built from the same set of basic 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 questions we posed.

Mobile UNITY [14] 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 [18] 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.

In CODEWEAVE the smallest 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.

CODEWEAVE is one of many models 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 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). Our contributions are both conceptual and technical. As shown 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. At the same time, CODEWEAVE has be employed as the basis for a working system [4] that employs XML to describe complex actions and their migration across sites.

The structure of the paper is the following. Section 2 provides a gentle introduction to Mobile UNITY, and illustrates how this notation can be exploited for modeling coarse-grained mobility, i.e., systems where the unit of mobility coincides with the unit of execution. Section 3 contains an informal presentation of our model for fine-grained mobility, CODEWEAVE. Section 4 illustrates how the model is given formal semantics based on Mobile UNITY. Section 5 illustrates an enhancement of CODEWEAVE that uses processes as nested scopes for units. Section 6 discusses how some existing mobile code mechanisms and technologies can be captured using our approach. Section 7 places CODEWEAVE in the context of related work. Finally, Section 8 ends the paper with concluding remarks.

2 Coarse-Grained Mobility in Mobile UNITY

This section provides the reader with a gentle introduction to the notation and computational model associated with Mobile UNITY, without going into the details of its proof logic and without making any attempt to be exhaustive in its coverage of the model. The reason for this detour is the simple fact that CODEWEAVE is fundamentally based on the concepts, notation, and proof logic for Mobile UNITY. Thus, a basic facility with Mobile UNITY is a prerequisite for gaining a good understanding of the CODEWEAVE concepts and notation. The Mobile UNITY advantages are its minimalist perspective on mobile computing, the explicit (yet abstract) modeling of space and location, the assertional proof logic, and the decoupled style of computing made possible by a coordination-centered approach to modeling component interactions. Finally, even though most think of Mobile UNITY as offering a coarse-grained view of mobility in which the unit of mobility and the unit of execution coincide, Mobile UNITY makes no assumptions about the size of programs making up a system model. As such, it becomes possible to consider programs whose scope is limited even to single variables and individual statements. It is latter feature that turned out to be the key to achieving the fine-grained mobility expressed in CODEWEAVE.

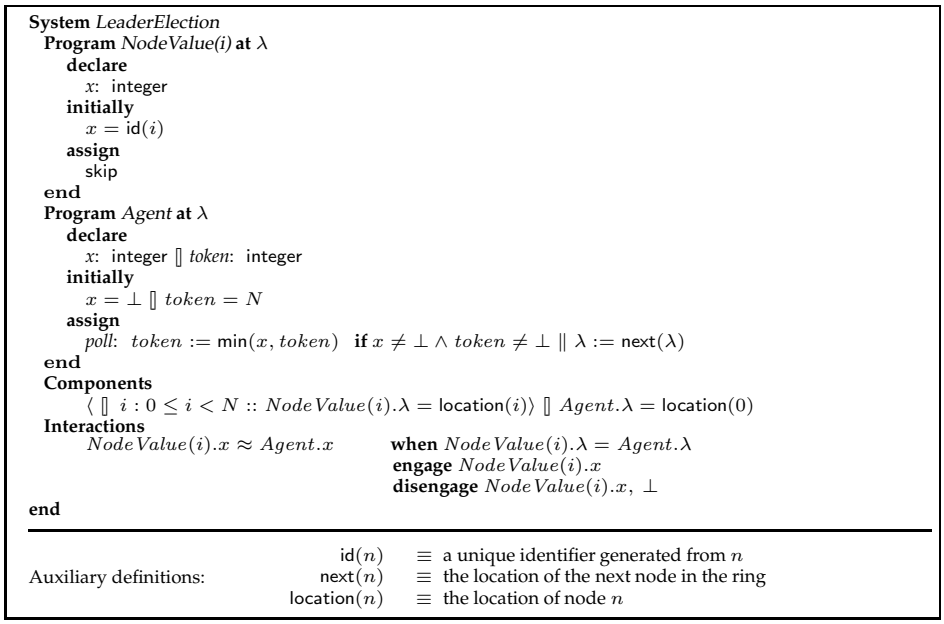

Figure 1: A mobile agent-based leader election algorithm expressed in Mobile UNITY.

The review of Mobile UNITY revolves around a leader election protocol that exploits mobile agents—a system exhibiting coarse-grained mobility. 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.

A Mobile UNITY [14] 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 1 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 1, 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 the number of hosts in the system, N, 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 1, multiple instances of program *NodeValue* are created and placed at various initial locations based on their index value[1], 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 1, 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

[1]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.

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 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.

Transient variable sharing is only one of many high-level coordination constructs that can be built using Mobile UNITY. It is the expressive power of the coordination primitives offered by Mobile UNITY that enables us to capture the semantics of CODEWEAVE by means of a simple two-step process that consists of a structural mapping and a behavior specification. The former entails a syntactic transformation of CODEWEAVE fine-grained components into Mobile UNITY programs while the latter (confined to the **Interactions** section) reduces the semantics of mobility and data access to a coordination problem readily captured using Mobile UNITY coordination constructs. However, before examining this coordination-centered strategy to semantic definition, we must introduce CODEWEAVE, its conceptual framework, and computational model.

3 Fine-Grained Mobility in CODEWEAVE

This section offers an informal presentation of CODEWEAVE. It starts with a high-level overview of the model. A refined version of the leader election example introduced in the previous section is used throughout this section to illustrate the features of CODEWEAVE. The protocol changes have been designed specifically for the purpose of helping the reader understand the way in which various patterns of fine-grained mobility can be exploited. Against the backdrop of the leader election example, we discuss the structure of CODEWEAVE programs, their dynamics, and the mobility constructs that can alter both. Section 3.5 walks through the same example in much greater detail. A formalization of CODEWEAVE follows in Section 4.

3.1 Model Overview

Central to CODEWEAVE is the interplay among the notions of location, containment, scope, and execution. 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. This brief introduction to CODEWEAVE is organized around three important themes: the structure of the model and the way in which the structure relates to the notion of space and spatial organization; the data access rules that govern the interactions among components and are controlled by the way in which the program is structured; and the laws governing the execution of program code.

In CODEWEAVE, we assume an underlying network composed of *sites*. Sites may represent physical hosts or separate logical address spaces within a host, e.g., an interpreter. Sites define *locations* within the network and may hold one more *units* representing code or data. A code unit need not contain a complete specification of a code fragment: it may even be a single line of code. The variables used in the code units are considered "placeholders" and they do not carry a value (i.e., their value is undefined). By contrast, units representing data contain single variable declarations and carry the actual values associated with the variable they represent. The model provides for data sharing between the units of code (devoid of storage ability) and the units of data (lacking executable code). One way for this to happen is by co-located units of both types within the scope of the same *process*.

The concept of process introduces a new set of locations that are below the site level, i.e., refines the structure the space in terms of a two level hierarchy. Processes reside on sites and are defined as containers for code and data units. Both types of units can be placed either directly on a site or inside a process. In the latter case, they are said to be *contained* by the process[2]. The scope of a unit contained in a process is the process itself, i.e., a unit can only access units that are contained in the same process. The binding mechanism defined by the model allows sharing among variables with the same name and within the same scope. The scope of a unit that is not contained in any process

[2]The model presented in this section is kept simple by not allowing processes to contain other processes. We investigate this enhancement in Section 5.

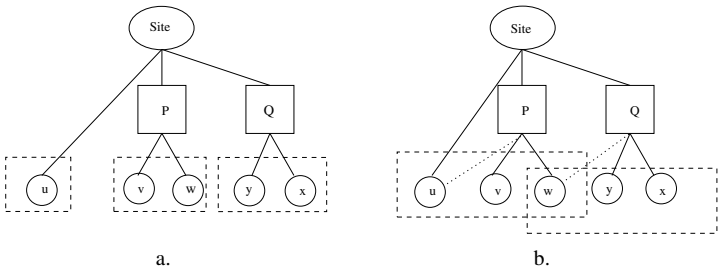

Figure 2: Processes, units, and scoping rules. Solid lines represent the containment relation among sites (i.e., hosts), processes, and units, while dotted lines represent references to units. Dashed rectangles represent a common scope for units.

(i.e., located directly on the site) is restricted to the unit itself. In Figure 2.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 it cannot share data with anyone else.

Sharing data among units within processes that reside on the same site is often useful, e.g., to specify the sharing of a common resource. For this reason, CODEWEAVE allows a process to *reference* a unit contained in another process on the same site. In this 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 classes or resources provided by the site to all processes located there. Figure 2.b shows an evolution of the system from Figure 2.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.

In CODEWEAVE, processes play a key role in controlling the execution of code units within their scope. Code units can execute only when present inside the scope of a process which is *active*. A process may be seen as a unit of execution but only in the sense that its status (i.e., whether it is active, inactive, or terminated) constrains the execution of the code inside its scope.

Mobility extends to both units, which may move among sites and processes, and processes, which move among sites. In addition, processes do affect the mobility of units since the movement of a process entails the movement of all the units contained in it. Referenced units however, are not moved along with the processes that refer to them as they are not contained by them. Furthermore, the data sharing rules inhibit 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 data sharing rule, also referred as the 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.

3.2 A Reference Example: Leader Election Using Fine-Grained Mobility

This subsection together with Section 3.5 define a backdrop for a gentle overview of the CODEWEAVE features. The example described here at a high level is revised in Section 3.5 in much greater detail. Here we seek to convey the essence of its mobility profile while later we examine its realization by walking through the constructs that make it possible.

The leader election protocol we presented in Section 2 is refined here by assuming 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

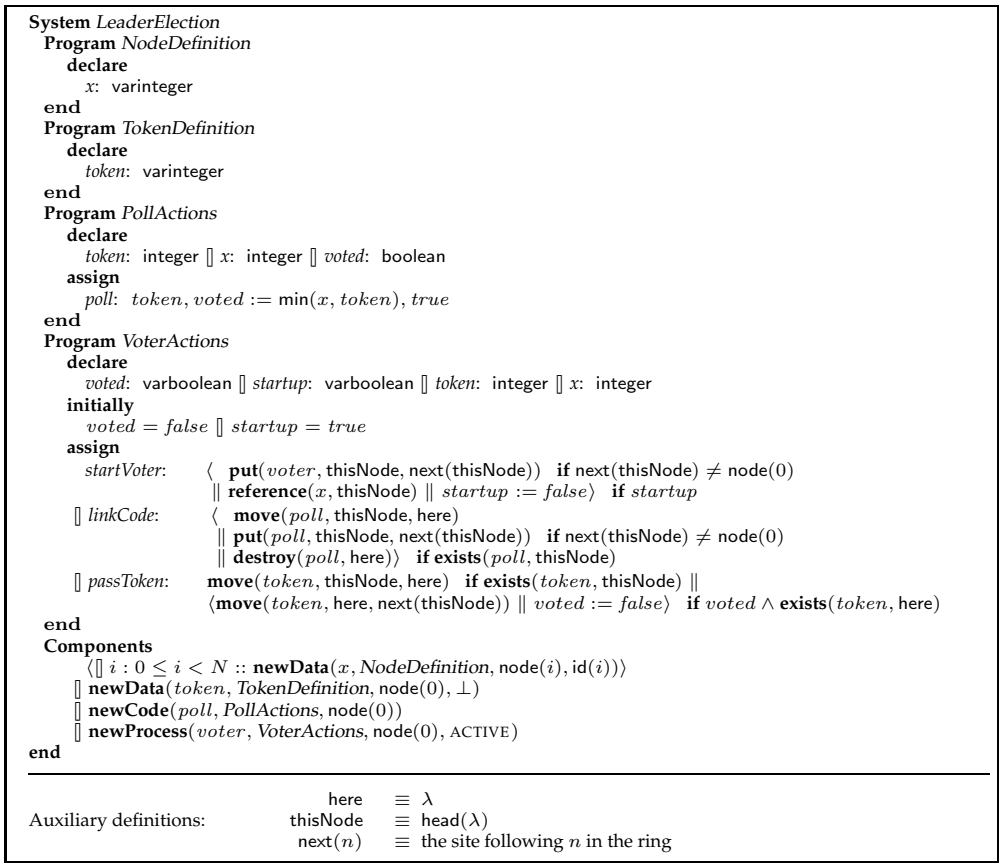

Figure 3: A protocol for leader election using fine-grained code mobility and specified using CODEWEAVE.

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 node 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.

3.3 Defining Code and Data Units

Figure 3 shows a specification of the leader election protocol using CODEWEAVE. The syntax of the specification is very similar to Mobile UNITY, but there are some significant differences.

To begin with, in Mobile UNITY the **Program** declaration is the unit of definition for a component, and hence plays a role in defining the unit of execution, scoping, and of mobility. In fact, each executing component is instantiated from a program, has direct access only to the variable defined in the program, and is moved as a whole. In CODEWEAVE, instead, a **Program** is merely a unit of definition: the other aspects, that is, execution, scoping, and mobility, are treated independently from the program declaration. Hence, in a CODEWEAVE **Program** declaration, statements and variables become independent units of mobility, which can be migrated into the scope of different processes, possibly with a very different structure, and executed there.

Moreover, the **Program** construct is used in CODEWEAVE also to define independent code and data units, that are

not initially part of any other component. This is shown in Figure 3, where the program declarations *NodeDefinition* and *TokenDefinition* define two independent data units, somewhat related with the variables x and $token$ of the program *Agent* in the original Mobile UNITY specification of Figure 1. Similarly, the program declaration *PollActions* defines a code unit that contains the $poll$ statement originally defined in Figure 1, together with the declarations for the variables it uses. Both *NodeDefinition* and *PollActions* declare a variable x. However, only the name x in the former is actually associated with a storage area holding the value for the variable, by virtue of the keyword var prepended to the variable declaration. This is actually what characterizes *NodeDefinition* as a data unit. The variable x in *PollActions*, instead, acts as a placeholder, and the associated value is accessible only if some data unit belongs to the same scope. Program declarations defining multiple code and data units at once are possible. For instance, the program *VoterActions* in Figure 3 implicitly defines the data units associated to the variables $voted$ and $startup$, and the code units associated to the statements $startVoter$, $linkCode$, and $passToken$. Although this feature is not exploited by our solution, these units could be relocated independently.

Since a **Program** is now merely a unit of definition, it does not make sense to index it with an instance discriminator or specify its location, as in the Mobile UNITY declaration **Program** *P(i)* **at** λ. Nevertheless, instances of code and data units must nonetheless be distinguishable, and there must be a way to tell where they reside. Each unit is uniquely identified by the triple (u, i, j), that we often write more compactly as $u_{i,j}$, where u is the name of the unit (e.g., x), i is the name of the program where the unit is defined (e.g., *NodeDefinition*) and j is the instance discriminator. This representation allows two units with the same name to be present in different program contexts. Processes are identified with a similar triple. As the reader may observe, contrary to the other two unit constituents, the instance discriminator j is not explicitly defined anywhere in the CODEWEAVE specification of Figure 3, and should be thought of as "automatically added" to a unit upon its instantiation. The details of how this is actually accomplished are treated in Section 4. Here, it is sufficient to hint at the fact that the semantics of CODEWEAVE can be expressed in terms of Mobile UNITY and, similarly, a CODEWEAVE specification can be "translated" into an equivalent Mobile UNITY specification, where each unit is represented by a Mobile UNITY component with its own instance discriminator and location variable λ.

3.4 Mobility Constructs

In this section we describe in more detail the set of constructs available in CODEWEAVE, which are summarized in Figure 4. In the figure, the constructs are defined by referring to a triple $(name, program, index)$ that, in general, may refer to a unit or to a process.

Definition through a **Program** declaration does not cause the instantiation of a unit or process. This can be performed explicitly by using the predicates at the top of Figure 4, which must be specified in the **Components** section of a specification. The predicate **newData** causes the instantiation, at location l, of a data unit with name u, using the definition provided in the program i, and with a previously unused instance discriminator j. The optional value v, if specified, provides the initial value of the variable associated to the unit. In this way we allow different instances of the same unit to be initialized with different values. Otherwise, the value defined by the program is taken as the initial value[3]. The predicate **newCode** achieves a similar goal for code units, without any need to worry about variable values. The predicate **newProcess** causes the instantiation, at location l of a process named u using the definition in the program i. Moreover, the initial status s of the process can be specified, as one of $\{$ACTIVE, INACTIVE, TERMINATED$\}$.

The mobility constructs are those at the bottom of Figure 4. The **move** operation causes the migration of component $u_{i,j}$ to the location l. If $u_{i,j}$ is a code or data unit, the location l can refer to a site or to a process. In the latter case, the location is defined as the concatenation of the name of the site the unit resides on and of the name of the process that holds it. On the other hand, if $u_{i,j}$ is a process then l must be a site[4]. Moreover, if the **move** operation is invoked on a process $u_{i,j}$, all the code and data units contained in the process are migrated along with it to the destination site. As we already mentioned, migration of a code or data unit affects bindings. For instance, if a data unit containing a variable v is being migrated from a process to another, sharing of v through variables with the same name becomes enabled for code units belonging to the target process and disabled for code units in the source process.

The appearance of a component at location l can be caused not only by migration, but also by explicit creation or cloning. Explicit creation is provided by the **new** operation, which is similar to the aforementioned creation predicates. Nevertheless, **new** allows for dynamic component instantiation, while the aforementioned creation predicates, placed in the **Components** section, allow only for static instantiation. Cloning is provided in CODEWEAVE through

[3] According to Mobile UNITY rules, if no value is provided by the program the variable is initialized to an arbitrary value belonging to its type.
[4] We lift this constraint in Section 5.

newData(u, i, j, l) or **newData**(u, i, j, l, v)	instantiate at location l a new data unit with name u and defined in program i, with a previously unused instance discriminator j, i.e., a unit $u_{i,j}$; set its value as specified in the program or, optionally, to the value v
newCode(u, i, l)	instantiate a new code unit $u_{i,j}$ at location l
newProcess(u, i, l, s)	instantiate a new process $u_{i,j}$ at location l, with initial status s
find(u, l) or **find**(u, i, l)	find and return the unit $u_{i,j}$ located at l, based on its name u and, optionally, the program i where it is defined
exists(u, l) or **exists**(u, i, l)	return $true$ if a unit named u (and, optionally, defined in a program i) is located at l, $false$ otherwise
move(u, i, j, l)	move the component $u_{i,j}$ to location l; if $u_{i,j}$ is a process, all its enclosed units are moved along with it
put(u, i, j, l)	create a copy of the component $u_{i,j}$ at location l; if $u_{i,j}$ is a process bindings are not preserved (variables are restored to their initial values)
clone(u, i, j, l)	create a copy of the component $u_{i,j}$ at location l;
new(u, i, j, l)	instantiate at location l a new component (unit or process) with name u and defined in program i, with a previously unused instance discriminator j, i.e., a component $u_{i,j}$;
reference(u, i, j, v, k, h)	establish a reference so that it becomes possible to access $v_{k,h}$ through $u_{i,j}$
unreference(u, i, j, v, k, h)	remove the reference between $u_{i,j}$ and $v_{k,h}$ if $u_{i,j}$ is a process bindings are preserved (variables retain their current values)
activate(u, i, j)	change the status of process $u_{i,j}$ to ACTIVE; the statements in its code units can now be selected for execution
deactivate(u, i, j)	change the status of process $u_{i,j}$ to INACTIVE; the statements in its code units cannot be selected for execution until the process becomes active again
terminate(u, i, j)	change the status of process $u_{i,j}$ to TERMINATED; the statements in its code units can never be selected for execution, i.e., the process cannot become active again
destroy(u, i, j)	remove the component $u_{i,j}$ from the system; if $u_{i,j}$ is a process, all its enclosed units are removed as well

For all the operations above, alternative and often preferable forms substitute some of the parameters characterizing a component (e.g., (u, i, j)) with a parameter representing the location where the target component is to be found (e.g., (u, i, l) or (u, l)). For instance, the **put** operation can be expressed also as **put**$(u, i, l_{cur}, l_{dest})$, meaning that a component with name u and defined in program i, regardless of its index j, is selected and cloned from location l_{cur} to location l_{dest}. Similarly for **put**(u, l_{cur}, l_{dest}), where the program where u is defined is disregarded as well.

Figure 4: Syntax and informal semantics of CODEWEAVE constructs.

two operations: **put** and **clone**. Both operations create, at location l, a new instance of a component $u_{i,j}$, i.e., a component with the same name u, the same program definition i, but a different instance discriminator $k \neq j$. If the component is a process, cloning is performed recursively also on all its constituent units. In the case of **put**, however, the bindings that a process may have established are not preserved as a consequence of this operation, i.e., all the variables are restored to their initial values. This represents a "weak" form of cloning. CODEWEAVE provides also a stronger notion with the **clone** operation, which preserves all the bindings owned by the process. As it will become clear in the next section, cloning takes place behind the scenes by picking a fresh component from the ether and setting its location to the one passed as a parameter.

Within the context of a given process, bindings are established and severed implicitly as units appear or disappear from within its scope. However, they can also be established explicitly across process boundaries by the specifier, using the **reference** and **unreference** operations. A reference operation involving a code and a data unit, call them $u_{i,j}$ and $v_{k,h}$, establishes a transient sharing among the variables contained in the two units. References are not affected by migration. The referenced variable $v_{k,h}$ is not migrated when the referencing variable $u_{i,j}$ is. Moreover, if $u_{i,j}$ is migrated in the context of a process that contains a variable named v and defined in a program k, but with a different instance discriminator h, the binding is re-established. As we mentioned in Section 3.1, this simply reflects a choice we made when defining the formal semantics of CODEWEAVE. Other sharing rules are easily implementable using the techniques we illustrate in Section 4.

The next three operations in Figure 4, **activate**, **deactivate**, and **terminate**, enable the specifier to control the execution status of processes. The operations **activate** and **deactivate** toggle the execution status from INACTIVE to ACTIVE and vice versa, and hence respectively enable or inhibit the execution of the code units contained in the process (with no effect on access to data units). The operation **terminate**, on the other hand, inhibits permanently such execution: once terminated, the process can no longer be activated.

Finally, the **destroy** operation allows the specifier to explicitly remove a component from the system (i.e., send it back to the ether). If **destroy** is invoked on a process, all the code and data units contained in it are removed as well.

Although thus far we provided a description of CODEWEAVE constructs in terms of the triple $u_{i,j}$ denoting a component, we already hinted at the fact that the third index is actually provided by an additional translation step. Moreover, keeping track of the instance discriminator would lead to cumbersome specifications. For this reason, several of the constructs provided in Figure 4 are made available in alternative forms that do not need to specify the instance discriminator. For instance, **move**$(u, i, l_{cur}, l_{dest})$ and **move**(u, l_{cur}, l_{dest}) are both legal operations, whose meaning can be understood best by relying on another construct in Figure 4, the **find** function. This function allows one to search a site l for a component, given the name u and, optionally, the program name i where the component is defined. The function returns the triple (u, i, j) corresponding to the found component: the value returned in the

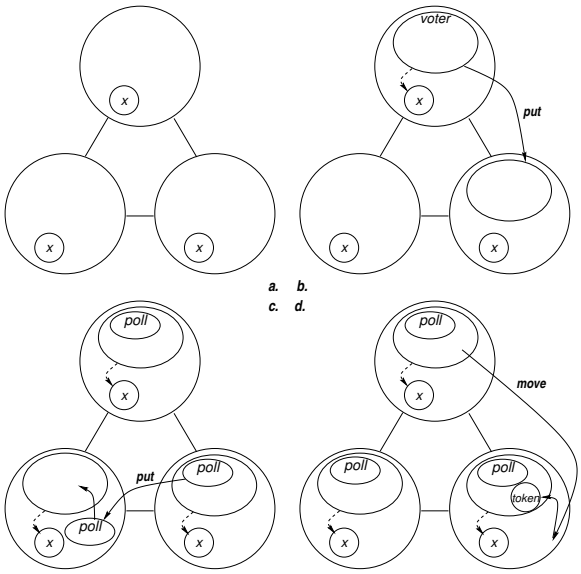

Figure 5: (a) Each node in the ring holds an identifier x. (b) The voter process establishes a reference to the local identifier and replicates itself on the next node. (c) The polling policy continues to be replicated on the next node and to be absorbed into the voter process. (d) The token travels from one node to the next.

case where the component does not exist is not defined. For this reason, **find** should always be used together with the function **exists**, which is defined with arity similar to **find** but returns a truth value according to whether or not a component matching the parameters is found on the given site. It is now evident how the aforementioned alternative forms of **move** can be reduced to the original one by properly applying the **find** function. For instance, it is possible to retrieve the missing information in **move**(u, l_{cur}, l_{dest}) by obtaining $u_{i,j} =$ **find**(u, l_{cur}), and hence reducing to the most general form **move**(u, i, j, l_{dest}). Similar notational shortcuts are available for creation predicates, e.g., to reduce the shorter form **newData**(u, i, l) to the most general **newData**(u, i, j, l). We come back to this issue in Section 4.

3.5 Leader Election Example Revisited

We are now ready to walk through the example shown in Figure 3, where many of the constructs we presented here are used in the specification of the leader election protocol. A graphical representation of the specification is provided in Figure 5.

Let us first focus on 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 $voter$ process that automatically contains instances for all the aforementioned units by using **newProcess**.

The data unit x defined in *NodeDefinition* and the data unit $token$ defined in *TokenDefinition* contain the variables with the same name, which are accessed (through sharing) by code units defined by the program *PollActions*. The latter contains a single statement *poll*, which describes the polling strategy. As discussed in Section 3.1, the execution of this statement is prevented when the corresponding code unit is not within the scope of any process. Thus, the comparison in *poll* is performed only when the corresponding code unit is co-located in a voter process that also

contains the data unit corresponding to *token*. In this case, the binding rules of the model, expressed using the transient variable sharing abstraction provided by Mobile UNITY, effectively force the same value in both *token* variables, hence enabling the comparison specified by *poll*. Simultaneously, an additional 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 thus far and an additional boolean *startup* that is used to determine whether it is necessary to perform some initialization tasks, i.e., cloning the voter itself on the next node to perform the initial deployment of processes in the ring, and acquiring a reference to the node's value. These tasks are performed simultaneously by the statement *startVoter*, which also resets *startup* to prevent the creation of multiple clones of the voter. In *startVoter*, cloning is performed by the **put** operation. It executes only if the voter that is invoking the operation does not immediately precede in the ring the site 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 Figure 3. 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. Moreover, the **put** operation is invoked by specifying only the name of the component to be cloned, together with the current location. Hence, according to what we described earlier, the current node is implicitly queried for a component matching the name *voter*, by invoking the **find** operation behind the scene.

The statement *startVoter* establishes also a reference to the variable *x*, whose value is contained in the corresponding data unit instantiated on each node. 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, according to what we described earlier, only the name of the data unit *x* is specified in the operation, while its indices are determined by implicitly querying the node.

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** 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.

4 Formalizing CODEWEAVE

In this section we present a formalization of the syntax and semantics of CODEWEAVE. Our general strategy is to reduce CODEWEAVE to a specialization of the standard Mobile UNITY notation and proof logic. The first step is to translate each CODEWEAVE component definition back to a single Mobile UNITY program. Once this mechanistic 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. This strategy allows us to express fine-grained code mobility at the appropriate abstraction level enabled by CODEWEAVE, and yet to inherit the proof logic associated with Mobile UNITY. Moreover, this strategy allowed us to test the expressiveness of Mobile UNITY as a specification language: the fact that the entire semantic specification can be reduced to a small set of coordination statements attests to the flexibility of Mobile UNITY.

It is interesting to note that, 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 specifier 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 first the topic of how to translate a CODEWEAVE specification back to a Mobile UNITY one, and then we turn our attention to scoping issues and statement scheduling, mobility constructs, and creation predicates.

4.1 From CODEWEAVE Components Back to Mobile UNITY Programs

In this section, we describe how a CODEWEAVE specification can be syntactically transformed into a Mobile UNITY one. Our treatment is informal, in the sense that we are not giving the formal translation rules as they are intuitive, tedious, and do not constitute a significant contribution. Instead, we illustrate the process by referring to the leader election example presented in Section 3.2.

In Section 3.3, we discussed how CODEWEAVE reinterprets Mobile UNITY syntax. In particular, we discussed how CODEWEAVE limits a **Program** statement to a mere unit of definition, depriving it of its role as a unit of execution, scoping, and mobility. Moreover, in Mobile UNITY every component has a location attribute, whose change in value denotes the movement of all the code and data within it. This is similar to CODEWEAVE processes. Nevertheless, in CODEWEAVE we seek to introduce a finer granularity, one that allows the movement of lines of code or variables as isolated entities—i.e., CODEWEAVE units. Here, we essentially map each CODEWEAVE unit and process back to a full-fledged Mobile UNITY program. This way, each CODEWEAVE component is now given the ability to move, execute, and interact independently according to the semantics of Mobile UNITY.

To understand how the syntactic transformation can be accomplished, it is useful to compare the CODEWEAVE specification of the leader election example in Figure 3 with the corresponding translation in Mobile UNITY shown in Figure 6. The first two **Program**s are the translation of the first two data units in Figure 3. A few things are worth noting:

- Each **Program** is identified using the *name(index)* form characteristic of Mobile UNITY. Here, however, the program name is fixed and by convention set to *p* for all programs. Moreover, the index qualifying a specific program instance is structured as a triple (u, i, j) defined as mentioned in Section 3.3, i.e., the first index contains the name of the CODEWEAVE component, the second index is the program definition where it was declared, and the last index is the instance discriminator. For instance, the name of the first program in Figure 6 is *p('x,'NodeDefinition,i)*. The first two indices range over finite enumerations, and we use a quote to distinguish the actual components from their names, e.g., to distinguish the component $token$ from its name *'token*. 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 defined in program i.

- Each **Program** contains the declaration of the variable characterizing the data unit. In addition, a number of other auxiliary variables are declared:

  - Each component, (i.e., data unit, code unit, or process) $c_{i,j}$ is characterized by a location $c_{ij}.\lambda$, by a request field $c_{i,j}.\rho$ that holds mobility commands the system is expected to execute on its behalf, and by a type $c_{i,j}.\tau \in \{\text{DATAUNIT}, \text{CODEUNIT}, \text{PROCESS}\}$.
  - In addition, each process $q_{i,j}$ is also characterized by a set of referenced units $q_{i,j}.\gamma$, and by the process activation status $q_{i,j}.\omega \in \{\text{ACTIVE}, \text{INACTIVE}, \text{TERMINATED}\}$.

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

  Here, it is however worth noting that the location variable λ is among these variables, as in any Mobile UNITY specification. The location of a process is always a site[5], while the location of a unit can be either a site or a process. In the latter case, the location of the unit is the concatenation of the site name where the process resides and of the identifier of the process the unit is contained in (i.e., the three process indices (q, i, j)). Moreover, 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.

[5]In Section 5 we discuss how to remove this constraint.

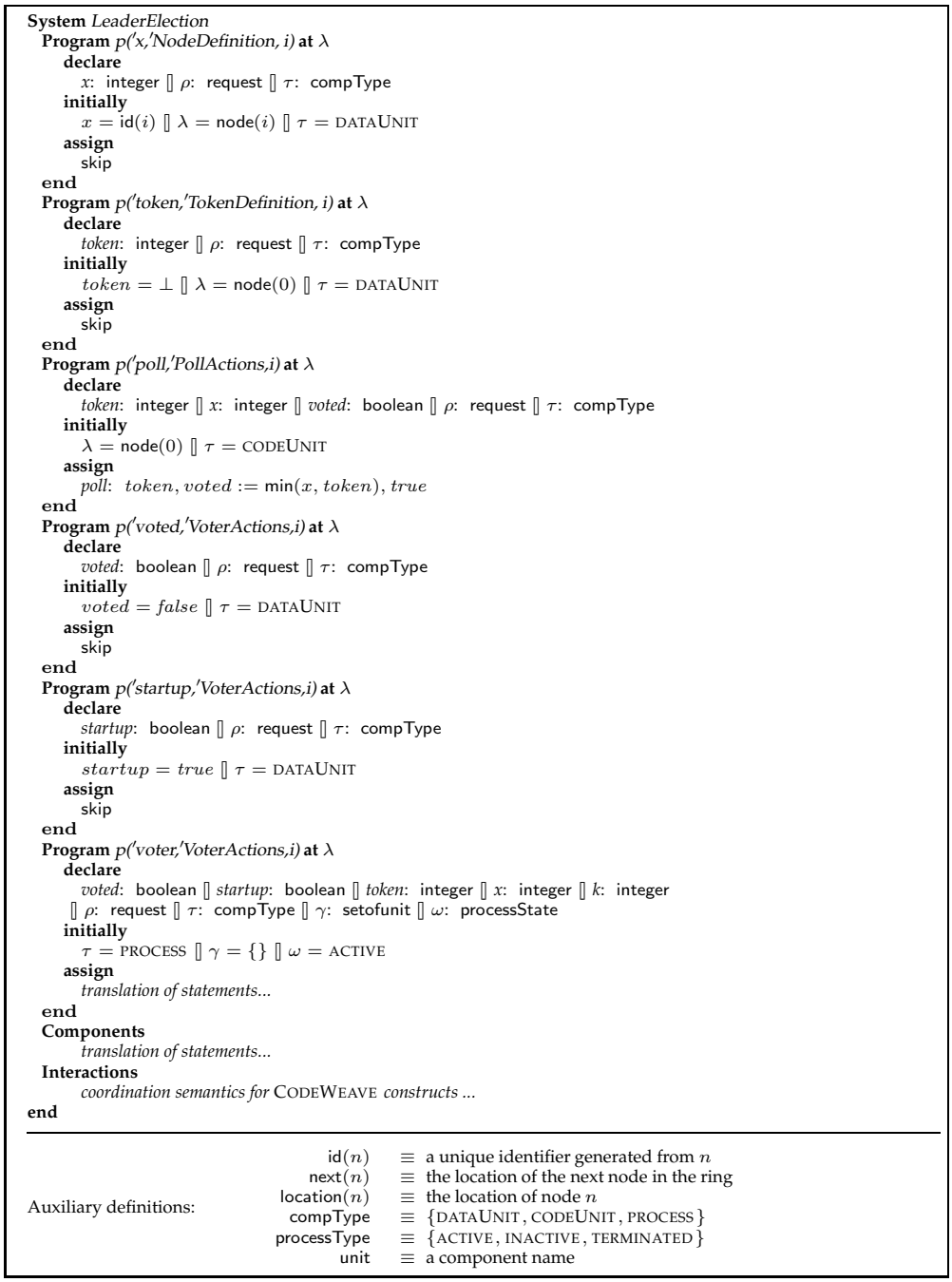

Figure 6: Translating the CODEWEAVE example of Figure 3 into Mobile UNITY.

- Each **Program** contains now also an **initially** section, used to initialize the variables contained in the **Program**, and an **assign** section that is obviously empty, since no statement is associated with a data unit. The initialization of variables, both user-specified and auxiliary ones, is performed by relying on the semantics of **newData**, whose definition is discussed later on.

The third **Program** is the translation of the poll code unit in Figure 3, and it is essentially unchanged with respect to its CODEWEAVE counterpart. Some additional considerations must be made for the translation of process *voter* in Figure 3, instance of the program definition *VoterActions*. The program contains the declaration of two variables, *startup* and *voter*, prepended by the keyword var. As we discussed earlier, this keyword characterizes the variable as

an independent data unit. This syntactic element is then sufficient to decide that the variable should be translated similarly to the other data units, as shown in the fourth and fifth program of Figure 6. However, for the rest of the process, that involves only variables and statements defined in the process and never referenced by any CODEWEAVE construct, two alternative translations are possible. In one case, each variable and statement is translated into a data or code unit, respectively, and the location of these units initialized to place them inside the process, as discussed earlier. In the other case, the process is simply translated as a single Mobile UNITY program containing the aforementioned variable and statements. As it will become clear in the following, the two alternatives are semantically equivalent, since units co-located in a process behave just like variables and statements defined in that process. In Figure 6 we chose the second alternative, as it is more compact and readable.

Figure 6 omits the translation of the CODEWEAVE constructs that appear in the code, the initialization of components, and the definition of interactions. This is because, up to this point, we are mainly concerned with discussing how the *structure* of a CODEWEAVE specification could be translated into Mobile UNITY and are missing some of the elements required to complete the picture. For instance, we will need the semantic definition of the CODEWEAVE constructs, given in the next section, to produce a mechanical translation of the CODEWEAVE code into Mobile UNITY code and to generate the contents of the **Interactions** section. The former entails a line-by-line translation while the latter consists of a coordination program specific to CODEWEAVE, i.e., independent of the particulars of a given CODEWEAVE program and tailored to the choice of constructs being provided and their semantics. Variations in the contents of the **Interactions** section relate only to variations in the model itself. Hence, once the definition of the constructs is given, translation becomes just a tedious and mechanistic process of substituting component names into formulas. Regarding the **Components** section, its role is that of identifying the set of programs that will participate in the computation. In the case of CODEWEAVE, we need to identify the maximal set of components included in the model (because Mobile UNITY does not provide for dynamic program creation) and to discriminate between the components that are in the state of stasis and those that are residing at specific locations in the system. Another way to state this is to think of defining the universe of components and the initial configuration. The information is readily available in the **Components** section of the CODEWEAVE program, when combined with knowledge regarding the upper bound on the maximum number of components ever to be instantiated. Ultimately, the translation process results in a quantified statement that places each program either in the ether or at some location according to the desired initial configuration for the system.

4.2 Scoping Rules and Statement Scheduling

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 we mentioned in Section 2, transient variable sharing across programs. The predicate controlling the sharing simply needs to capture the scoping rules. Figure 7 shows how these rules can be stated as two Mobile UNITY coordination statements. Statement 1 handles sharing between a variable in a data unit and a variable in a code unit, while statement 2 defines the sharing between two variables in data units. Statement 1 states that variables[6] $u_{i,h}.x$ 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 either contained in or referenced by the same process, i.e., the units are in the same scope. In turn, sharing uses the functions containedIn$(v_{j,k}, u_{i,h})$, that determines whether $v_{j,k}$ is a unit contained in $u_{i,h}$, and referencedBy$(v_{j,k}, u_{i,h})$, that determines whether $v_{j,k}$ is referenced by $u_{i,h}$. Note how the latter exploits the auxiliary variable γ associated to each process, as described in Section 4.1.

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. As no **disengage** is specified, the variables retain the values they had before the **when** condition became false. Alternative choices for the sharing rule,

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

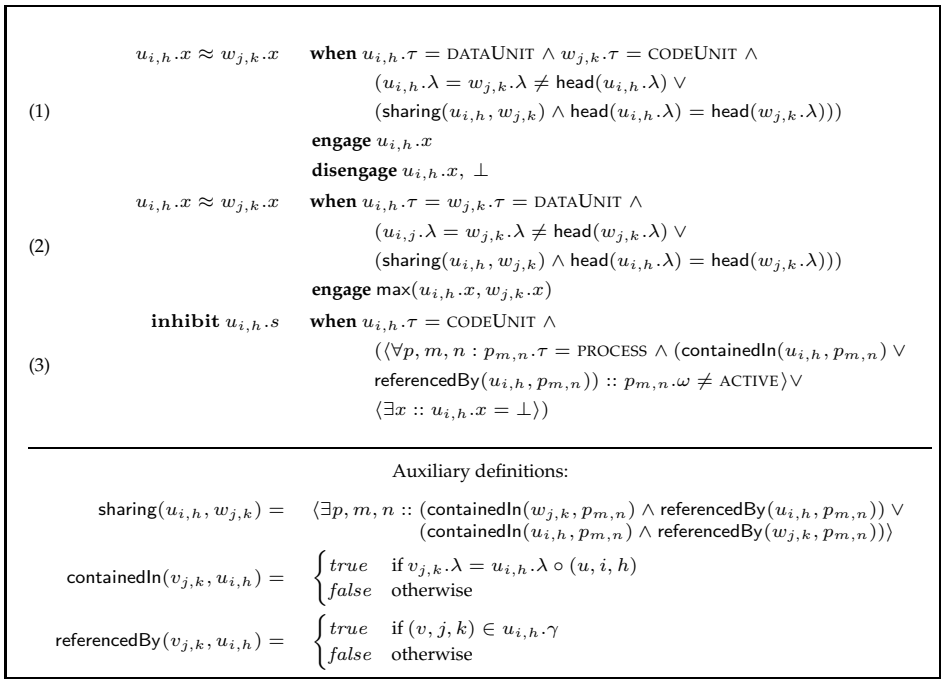

Figure 7: Establishing bindings among units using transient variable sharing and statement inhibition.

hence leading to a different semantics for value reconciliation, can be implemented easily.

Finally, statement 3 takes care of preventing the execution of code units that are not in the scope of an active process or 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). 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. Nevertheless, 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 to inhibit the execution of the statement associated to a code unit when in the aforementioned situations. Note how in Figure 7, and also later in Figure 12, we implicitly restrict the quantification over the variables used in the formulae to a proper range, namely, the set of names appearing in a given unit.

4.3 CODEWEAVE Constructs

In this section we turn our attention to the specification of the CODEWEAVE constructs we illustrated in Section 3.4. First, we give the definition for the functions **find** and **exists**, which provide the ability to locate a component in the system. Then, we present the formalization of the mobility constructs. Finally, we give the definition of the creation predicates **newData**, **newCode**, and **newProcess**.

4.3.1 Finding Components

CODEWEAVE components, i.e., units and processes, are uniquely identified using the triple (c, i, j), where c is the component name, i the program where it was defined, and j the instance discriminator. Nevertheless, as we mentioned in Section 4, the specifier is likely to refer to a component in a specification only through its name, possibly complemented by the program where it was defined, but rarely through the instance discriminator. This is made possible by the functions **find** and **exists**, whose formal definition is shown in Figure 8. These functions range over all the components in the system, and check whether a component matching the indices and location passed as function arguments can be found among them. In the case of **find**, the component identifier is returned. To tackle the case where multiple matching components exist, we arbitrarily chose to pick the one with the minimum value of the instance discriminator.

$$
\begin{array}{rl}
\textbf{find}(c, l) \equiv & \langle \mathsf{min}\ i, j : c_{i,j}.\lambda = l :: (c, i, j) \rangle \\
\textbf{find}(c, i, l) \equiv & \langle \mathsf{min}\ j : c_{i,j}.\lambda = l :: (c, i, j) \rangle \\
\textbf{exists}(c, l) \equiv & \langle \exists i, j :: c_{i,j}.\lambda = l \rangle \\
\textbf{exists}(c, i, l) \equiv & \langle \exists j :: c_{i,j}.\lambda = l \rangle
\end{array}
$$

Figure 8: Definition of the functions **find** and **exists**.

4.3.2 Mobility Constructs

Our formalization of the mobility constructs of CODEWEAVE relies on the following strategy. A CODEWEAVE operation is translated into a Mobile UNITY statement that encodes the operation information (i.e., opcode and arguments) in a tuple that is placed in the distinguished variable ρ declared in every **Program** representing a component, as discussed in Section 4.1. Figure 9 shows the translation for each construct we defined informally in Section 3.4. We then delegate the actual execution of the operation to a series of coordination statements built into the **Interactions** section. The coordination statements possibly propagate the request to the contained units and ultimately carry out the migration of the individual components to the specified location. Atomicity of the actions carried by the coordination statements is guaranteed by the fact that they are defined as Mobile UNITY *reactive statements*. Mobile UNITY is based on a two-phased operational model in which the execution of an ordinary assignment statement is immediately followed by the execution of a set of reactive statements. Logically, the set of reactive statements is executed to fixed point right after each non-reactive statement, and one reactive statement may trigger the execution of other reactive statements. Actually, transient sharing is ultimately defined using reactive statements [14], but this is outside the scope of this paper.

To present the details of our formalization we follow step-by-step the execution of a single construct, the **move** operation. First of all, we should remind the reader that multiple variants of the **move** and of the other operations are defined, to simplify the specifier's task. For instance, the migration of a component is naturally expressed by specifying the source and target location of the component, as in $\textbf{move}(u, l_{cur}, l_{dest})$. Nevertheless, we already showed in Section 3.4 how this form can be reduced to the standard form $\textbf{move}(u, i, j, l_{dest})$ by exploiting the function **find**, e.g., as $\textbf{move}(\textbf{find}(u, l_{cur}), l_{dest})$. Since this latter form is the most general one, we used it in Figure 9, and hereafter it will be the only one used in our formalization.

Hence, let us assume that a component, or more precisely a process $q_{i,j}$, is being relocated to new location l by the CODEWEAVE operation $\textbf{move}(q, i, j, l)$. This invocation is translated, according to Figure 9, into the Mobile UNITY statement

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

which places the tuple on the right in the variable ρ associated to the component that invoked the operation. The first field of the tuple assigned to ρ contains the status of the operation. In this case, REQ indicates that the operation request has been just issued, and awaits processing. The second field is the opcode for the operation, in our case MOVE. The following three fields contain the three indices of the component the operation must be executed on. Finally, the last field contains the operation arguments, encoded in a list. In our case, they consist of the instance discriminator for the component to be moved and of the target location. Note how Figure 9 defines the auxiliary function getid, which returns the identifier of a component located in the ether, i.e., for which $\lambda = \epsilon$. A minimal lexicographical value for the triplet is selected. This definition is useful for defining the cloning operations **put** and **clone**.

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

Auxiliary definitions:

$$
\begin{array}{l}
\mathsf{getid}(name) \equiv \mathsf{tail}(\textbf{find}(name, \epsilon)) \\
\mathsf{getid}(name, i) \equiv \mathsf{tail}(\mathsf{tail}(\textbf{find}(name, i, \epsilon)))
\end{array}
$$

Figure 9: Mapping mobility constructs to Mobile UNITY statements.

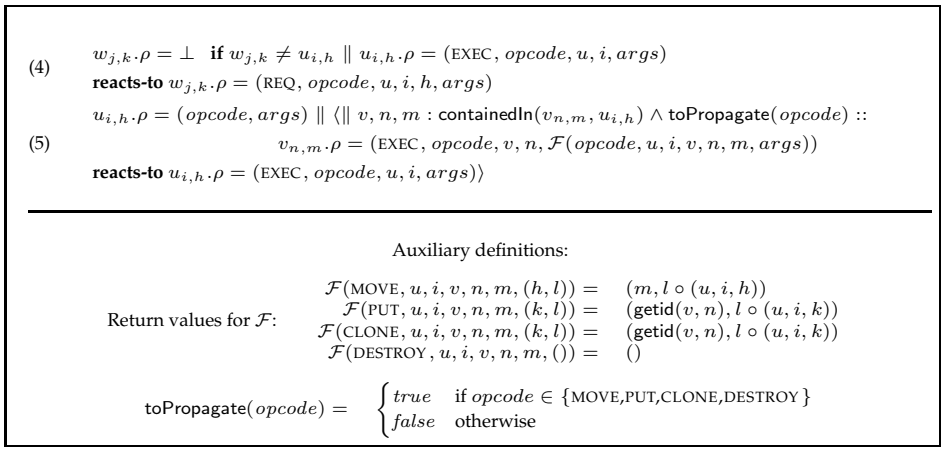

Figure 10: Propagating operation requests.

After the request for processing of an operation has been issued, the actual execution is carried out by a set of coordination statements placed in the **Interactions** section of the system specification. These statements, shown in Figure 10, are Mobile UNITY reactive statements in the form s **reacts-to** p, where s is the statement to be executed reactively, and p is the enabling condition for the reaction—if p is false the reaction is equivalent to a skip operation.

The first reaction, statement 4, is enabled by the presence of an operation request, and takes care of transferring it from the component that invoked the operation to the target component—in our case, process $q_{i,j}$. The operation status is changed from REQ to EXEC. In our case, the right assignment in statement 4 can be rewritten as

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

while the left assignment takes care of clearing the ρ of the component that issued the request. Note how this latter statement is guarded to take into account the case when a component requests an operation on itself.

The second reaction, statement 5, takes care of handling the propagation of an operation to the contained units, if any. For instance, let us assume that process $q_{i,j}$ contains two units, $d_{m,h}$ and $s_{k,n}$. According to what we described in Section 3.4, in this case a **move** operation invoked on $q_{i,j}$ has the effect of relocating the process on the target location as a whole, i.e., together with both its contained units. The left assignment of statement 5 keeps track of propagation by dropping the operation status, and in our case can be rewritten as

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

The other side of the $\parallel$ operator simultaneously propagates the opcode to the contained units, and does so by relying on a few auxiliary definitions. The set of components to which the operations should be propagated is computed by relying on the function containedIn, defined in Figure 7, to determine the set of contained units, and on the function toPropagate, to determine whether the operation at hand requires propagation to the contained units. For each unit in this set, the variable ρ is set to an operation request, whose arguments depend on the particular operation at hand. To make the formalization compact, we rely on a function $\mathcal{F}$, defined in Figure 10, to compute the arguments appropriately. For instance, in our case the statements propagating the request to the components become

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

and the function $\mathcal{F}$ builds the location target for the migration of the two units as the concatenation of the original target location for $q_{i,j}$, and the process identifier, so that the units are eventually relocated within $q_{i,j}$, no matter where it is moved. In our case, after statement 5 is executed for $q_{i,j}$, it is still enabled since the content of the ρ variable of the two contained units matches the enabling condition for the reaction. Nevertheless, in this case no further propagation is possible, since units are indivisible, and hence the statement executes by simply dropping the EXEC status on the operation request, by virtue of the assignment on the left:

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

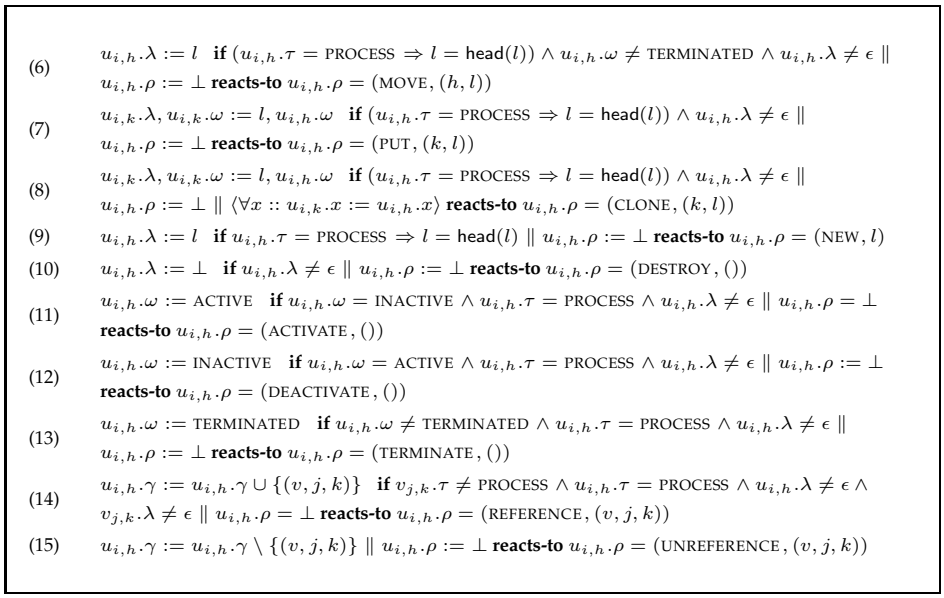

Figure 11: Performing operations on components.

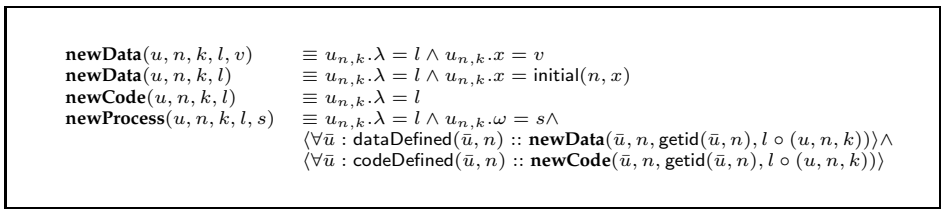

Figure 12: Predicates for the instantiation of components.

At this point, the operation has been propagated to the correct units and processes, and the requested actions (e.g., migration in our case) can be performed on them. This is accomplished by the statements in Figure 11 which, again, are meant to be contained in the **Interactions** section of the specification. The nature of the statements is strongly related to the meaning of the operation, and in the case of **move** it involves a change in the location of the components involved.

Note that each of the statements in Figure 11 is a reactive one. According to the semantics of Mobile UNITY, multiple reactions can be enabled, and they may fire in any order. For instance, in our case three reactions are initially enabled: one for the process $q_{i,j}$ and two for the contained units. Then, the migration of one of the contained units could potentially take place before the migration of the enclosing process, thus placing the unit at a non-existent location. This apparently inconsistent state, however, is "corrected" as soon as all the reactions have executed and the migration is complete. Since the whole set of enabled reactions execute atomically to fixed point (in our case, when the ρ variable is cleared in all three components), the actions representing the CODEWEAVE construct complete always in a consistent state. Hence, in our case the following predicate is guaranteed to hold at the end of the execution of the reactions corresponding to statement 6, independent of the order of their execution:

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

All other CODEWEAVE constructs function in way similar to **move**, except that not all of them are propagated to the contained units. For instance, **reference** only modifies the distinguished variable γ associated to the process.

4.3.3 Creation Predicates

Figure 12 shows the formalization of the creation predicates used in Figure 3 to instantiate units and processes, and initialize their values. Similarly to what we described earlier for **move** and the other mobility constructs, the creation predicates used in Figure 3 are a variant of the more general form shown in Figure 12. For instance, **newData**(u, n, l, v)

can be reduced to the standard form **newData**(u, n, k, l, v) by relying on the function getid defined in Figure 9, as in **newData**$(u, n, \mathsf{getid}(u, n), l, v)$. Hence, in the following we describe only the latter form, since it is the most general.

The first two definitions of Figure 12 define the two variants of **newData**: the first allows the specifier to define an initial value on a per data unit basis, while the second initializes the data unit with the value specified in the **initially** section. Reification of the data unit consists in changing its location to a non-ether value. The third definition deals with **newCode**, and contains only the location assignment representing instantiation, since no data value needs to be handled at instantiation time for a code unit. Finally, the last definition in Figure 12 contains the instantiation of a process through **newProcess**. This is accomplished by placing the process at location l and with the status s specified by the parameters. Moreover, 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 (whose definition is not shown for the sake of brevity) check whether the unit $\bar{u}$ is defined in a given program n of the specification.

5 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 [22] provide a flat model where agents (i.e., the execution units) move from host to host. A relevant exception is Telescript [20] 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, 19], for instance, model nested scope with a process algebra based formalism (more details and comparisons can be found in Section 7) 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 site 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 enclosing blocks. However, the inner blocks can access the content of the outer blocks. The binding mechanism presented in Figure 7 is readily extended to accommodate the new hierarchical process structure. Figure 13 shows an example. 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 u becomes bound to the 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 scoping tree (i.e., the site).

The access to referenced units must also be adjusted for use with nested scopes. 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 site) 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 specifier 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 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 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

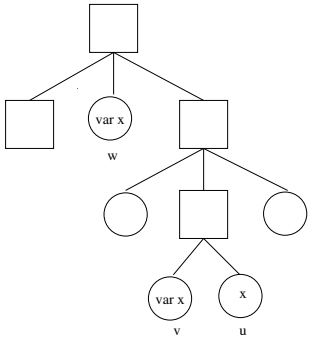

Figure 13: Scoping in the enhanced model: units containing variables with declaration are data units. Units containing variables without declaration are code units.

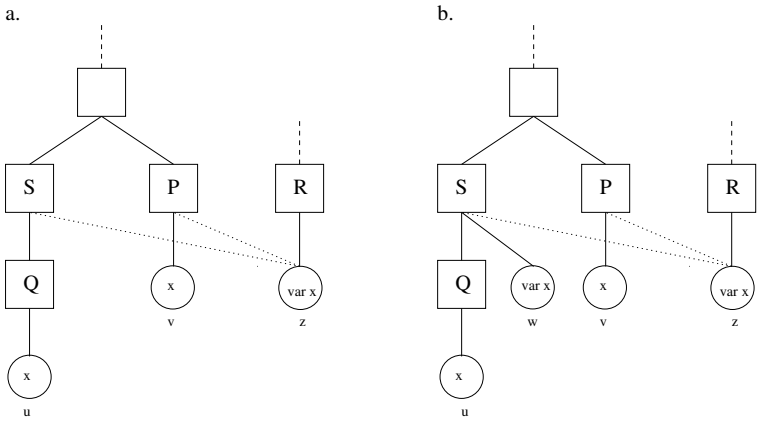

Figure 14: Referencing in the hierarchical model.

the accessibility of a unit only to peer units, for instance to hide the content of a unit from lower levels. To this end, 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.

Since the hierarchical model we described affects mostly the notion of location, the changes required to the semantics presented in the previous section are minimal. The operations **move**, **put**, **clone** are now more flexible as they can place a process inside another process, not only on sites. Let us consider, for instance, the **move** operation (statement 6 in Figure 11). The guard

$$u_{i,h}.\tau = \text{PROCESS} \Rightarrow l = \mathsf{head}(l)$$

ensures that a process can only be moved to a site, and not inside another process (i.e., l is a site). In the **move** modified for the hierarchical model we just described, this check disappears and the new formalization of **move** becomes:

$$u_{i,h}.\lambda := l \;\; \textbf{if } (u_{i,h}.\omega \neq \text{TERMINATED} \wedge u_{i,h}.\lambda \neq \epsilon) \parallel u_{i,h}.\rho := \bot \textbf{ reacts-to } u_{i,h}.\rho = (\text{MOVE}, (l))$$

In addition, the operations **activate**, **deactivate**, and **terminate** must now be propagated to all the child processes of the input process. Hence, the function $\mathcal{F}$ must be defined also for these constructs: in this case, it simply needs to return the () value.

6 Discussion

In this section we challenge CODEWEAVE by informally 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.

6.1 Reality Check

The goal of our model is to offer a formal tool that can be used to specify the semantics of operations and mechanims 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:

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

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

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

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 [11].

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 [13], 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 [20], 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 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 [13]), to completely dynamic and under the control of the programmer (like in μCODE [17]). 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 to 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.

6.2 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 [21]) 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.

Verification brings additional opportunities. 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 [18]. The fine-grained perspective adopted in this paper, however, enables not only to reason about the location of mobile programs, as in [18], 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 on the target site. 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.

Finally, other opportunities for a fruitful application of CODEWEAVE may come from research areas that are rather far from mobility. For instance, the ability to dynamically "weave" application code into a running application is fundamental in aspect-oriented programming. Hence, CODEWEAVE could be used to provide a formal specification of aspects, and of the run-time support necessary to their weaving into applications.

7 Related Work

Logical mobility has been the subject of multiple formal treatments. Many of them rely on specializations of process algebras, in sharp contrast with Mobile UNITY, which is a state-based transition system. As a result, they offer very different notions of mobility and treatments of space. Most often, mobility is reduced to a dynamic change in the system structure. For instance, π-calculus [15] is based on the notion of process communication via *channels*. Because channels names can travel along other channels, the movement of computations is represented as the movement of channels that refer to them. π-calculus has 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 a related development, Klaim [16] combines π-calculus with a tuple-space based language à la Linda [7] that provides the framework for formalizing the concept of location. Other models such as Ambient calculus [3] and Seal calculus [19]

go one step further and introduce an explicit notion of *environment* (*ambients* in Ambient calculus and *seal* in Seal calculus). Environments define the computation scope and movement is constrained by the structure of the defined system. In the remainder of this section we contrast the manner in which CODEWEAVE and some of the models listed above address issues fundamental to mobile computing.

The *granularity and nature of movement* was a paramount concern in the design of CODEWEAVE. The notions of statement and variable are clearly fundamental to all programming languages and were selected as units of mobility because we cannot imagine anything of a still finer granularity. As such, the choice is less dependent on the notation system of Mobile UNITY than one might believe at first sight. Similarly, in process algebra everything is a process, which naturally becomes the unit of mobility as well. Since processes can be of arbitrary complexity, the Ambient calculus can readily make the transition to using the environment as unit of computation and mobility. Also, by focusing on the confines of the immediately accessible context, mobility becomes a local restructuring of the environment itself. By contrast, π-calculus enforces a sense of locality by simply restricting the passage of channel names only to existing connections but lacks the concept of unit of mobility and, therefore, one cannot even consider the notion of granularity.

In retrospect, the idea of *decoupling the unit of execution and mobility* seems natural and desirable even though none of the earlier models provided any hints in this direction, maybe with the exception of channel passing. Programs in Mobile UNITY and processes in process algebras are units of execution, which were readily transformed into units of mobility as well. By considering the notion of moving variables we encountered the case of a unit of mobility that was not a candidate for execution. The realization that statements should execute only in the right context offered additional impetus to draw the distinction between the units of execution and mobility. The CODEWEAVE process was introduced to capture the notion of unit of execution as distinct from the unit of mobility. A process can function as both but statements and variables can move as well. Actually, we provide specific operations (**activate**, **deactivate**, **terminate**) to act on the execution state of processes.

Support for *objective and subjective* mobility is important for a model that seeks generality rather than support for a specific design methodology. 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. However, such constraints can be readily formalized in CODEWEAVE by restricting its use to certain schemas or by redefining the semantics of the constructs being used, if really needed. By contrast, in the Ambient calculus every environment can decide to move with its content whenever it wants to subject to the limits of its immediate context. This is built into the model in order to enforce a notion of operating within the confines of nested administrative domains. At the same time, the Seal calculus prohibits this behavior for security reasons. In the Seal calculus the environment decides on the movement of the contained entities.

Developing an understanding for what are the *basic operations for code mobility* was another goal of our investigation. All the formalisms considered provide more or less explicit mobility constructs, π-calculus to a lesser degree. A novel construct available in Ambient calculus is the *open* operation, which is 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, open can be formalized as a movement (i.e., **move**) of all the contained entities of a process followed by a **destroy** of the process itself. We specify the same 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 process algebraic models exploit the replication construct to formalize cloning and, by their very nature, provide no notion of initialization values.

As components move from one locale to another, the entities being visible to the unit change even though the definition of the unit does not. The manner in which a model handles the changing relation among names and the entities they denote, i.e., the *binding rules*, impacts the expressive power of the model and its style of computing. π-calculus, for instance, derives much of its elegance from the ability to create new names, to pass them among components, and to allow communications over channels known by components within the scope of the channel name. Ambient and Seal calculi rely on the notion of scoping defined by the environment hierarchy and common names within the same scope continue to play a role in binding names to resources. CODEWEAVE offers a wider range of binding constructs also constrained by scope, i.e., location. Location variables may be bound to a particular scope provided by the host or process. Variable names that are assumed to be free are bound to other variables that denote storage units and storage units at the same level within a process are bound to common values. The CODEWEAVE concept of references provides a mechanism for binding to resources across process boundaries. Scope-based binding is implicit and automatic. Reference-based binding is explicit and programmatic. In all these cased, names are bound

to resources. The other form of binding relates code units to contexts, This richer set of bindings seems to be required in CODEWEAVE because we distinguish among multiple types of units.

Security considerations are important when dealing with highly dynamic systems involving mobile code. In CODEWEAVE, the proof system allows one to verify that certain properties are preserved by the execution of the system and schemas can be employed to offer certain security guarantees by construction. Nevertheless, CODEWEAVE offers no specialized support features. In some models, security is based on a constrained mobility mechanism, e.g., in Ambient calculus and Seal calculus components move from one domain to another crossing one boundary at a time. Klaim [16] 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 see future work in this direction especially in terms of resource access constraints: a first step could be adding operators constraining the visibility of units (as we discussed in Section 5).

8 Conclusions

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 2^{nd} 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. 22^{nd} ACM Symp. on Principles of Programming Languages (POPL)*, 1995.

[3] L. Cardelli, G. Ghelli, and A.D. Gordon. Types for the Ambient Calculus. *Journal of Information and Computation*, 2002.

[4] W. Emmerich, C. Mascolo, and A. Finkelstein. Implementing Incremental Code Migration with XML. In *Proc. of the 22^{nd} Int. Conf. on Software Engineering (ICSE2000)*, pages 397–406, 2000.

[5] C. Fournet, G. Gonthier, J.J. Levy, L. Maranget, and D. Remy. A Calculus of Mobile Agents. In *Proc. 7^{th} Int. Conf. on Concurrency Theory (CONCUR '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 *3^{rd} 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] R.S. Gray, G. Cybenko, D. Kotz, R.A. Peterson, and D. Rus. D'Agents: Applications and Performance of a Mobile-Agent System. *Software: Practice and Experience*, 32(6):543–573, May 2002.

[11] Javasoft. *Java Remote Method Invocation Specification*. Available at `www.javasoft.com/products/jdk/rmi`.

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

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

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

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

[16] 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.

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

[18] G.P. Picco, G.-C. Roman, and P.J. McCann. Reasoning about Code Mobility with Mobile UNITY. *ACM Trans. on Software Engineering and Methodology (TOSEM)*, 10(3):338–395, July 2001.

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

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

[21] 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.

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