Mobile UNITY Schemas for Agent Coordination

Gruia-Catalin Roman and Jamie Payton

Department of Computer Science and Engineering
Washington University in St. Louis
Campus Box 1045, One Brookings Drive
St. Louis, MO 63130-4899, USA
{roman, payton}@cse.wustl.edu

Mobile UNITY refers to a notation system and proof logic initially designed to accommodate the special needs of the emerging field of mobile computing. The model allows one to define units of computation and mobility and the formal rules for coordination among them in a highly decoupled manner. In this paper, we reexamine the expressive power of the Mobile UNITY coordination constructs from a new perspective rooted in the notion that disciplined usage of a powerful formal model must rely on formally defined schemas. Several coordination schemas are introduced and formalized. They examine the relationship between Mobile UNITY and other computing models and illustrate the mechanics of employing Mobile UNITY as the basis for a formal semantic characterization of coordination models.

1 Introduction

Mobile UNITY [11, 19] is a formal model developed with mobility in mind and out of the desire to accommodate a new generation of systems that exhibit physical mobility of hosts and logical mobility of code (e.g., code on demand, remote evaluation, and mobile agent paradigms). Mobile UNITY inherits the simplicity of the original UNITY [4] and specializes its parent model in two important ways. First, it imposes a program structure that emphasizes decoupling and modularity. Systems (multiple cooperating programs in the parlance of UNITY) are structured in terms of program descriptions (types), program instances, and interactions. The last part defines the rules by which information is exchanged among programs which otherwise cannot communicate with each other—in a departure from UNITY, all variable names are local and only the interaction policy can refer to variables across program boundaries. Second, Mobile UNITY introduces several new constructs, including the reactive statement, which represents the boldest departure from its UNITY origins. While the term might be suggestive of event processing constructs in other models, the semantics of Mobile UNITY reactive statements are unique. The reactive statements are triggered not by events but by system state and have the power to alter the current state until a new configuration is reached in which no reactive statements are enabled.

The new model of concurrency inherits much of the notation and the proof logic directly from UNITY but makes the transition to mobility straightforward,

with no impact on program structure or proof logic. The typical application of Mobile UNITY to mobility is actually only a schema definition, i.e., a syntactic restriction on the general model. In this schema, each program is required to have a distinguished variable that denotes the current location of the program in some logical or physical space, and all interactions are conditional on the relative locations of the programs making up the system.

The Mobile UNITY approach to mobility is indeed different from that of Mobile Ambients [3], π-calculus [14], or mobile agent systems (e.g., LIME [16], MARS [2], D'Agents [7], etc.). Mobile Ambients structures space hierarchically and limits movement to conform to this hierarchical organization; the structure of the space is actually the structure of the system being described and can change with it. The π-calculus is a process algebra that allows for the creation of new unique channel names and for passing such names among processes for the purpose of establishing private communication channels. As a result, mobility is reduced to a restructuring of the communication structure, i.e., space and movement do not have a direct representation in the model. Mobile agent systems vary greatly but typically address the mobility of agents across connected hosts and the mechanics of coordination among them; the setting is generally that of logical mobility (LIME [16] is an exception in this respect, as it accommodates both logical and physical mobility), and space is assumed to be predefined. In Mobile UNITY space can be arbitrary, location is part of a program's state, and movement is reduced to value assignment to the location variable. Interactions among programs can be specified to meet the precise needs of each application domain.

The goal of this paper is to explore the richness of the model by examining a range of schemas that can adapt the basic Mobile UNITY model for a varied set of purposes. The expressive power of Mobile UNITY has been previously tested as part of a series of studies that demonstrated its ability to construct novel coordination constructs (e.g., transient and transitive variable sharing) [19, 11], to express clock-based synchronization [12], to specify and verify communication protocols (e.g., Mobile IP) [12], to specify and reason about code mobility paradigms [18], and to function as a semantic model for coordination constructs offered by new middleware for mobility. While building on earlier experience with Mobile UNITY, this paper seeks to distill the knowledge we gained and present its essence by reduction to a set of simple schemas that might assist in future efforts to apply Mobile UNITY to novel settings.

The Mobile UNITY model is presented in Section 2. The importance of schemas is highlighted in Section 3, where it is demonstrated that the treatment of mobility in Mobile UNITY can be defined as a schema. Sections 4 through 7 introduce new schemas for mobile computing and coordination. While the coordination mechanisms discussed in these sections relate directly to specific formal models or systems known in the literature on coordination or mobility, no effort is made to detail how a model or system might be simulated in Mobile UNITY, as our interest is in defining interesting coordination schemas rather than putting forth claims of universality. In each case, the origins of the schema

will be acknowledged, but its formalization will be abstract and minimalist. Finally, conclusions are presented in Section 8.

2 The Essence of Mobile UNITY

In this section, we give a brief introduction to the Mobile UNITY [11, 19] model, first describing the notation and associated proof logic, then applying the model to a simple example.

Notation. As in UNITY, the key elements of program specification are variables and assignments. Programs are simply sets of conditional assignment statements, separated by the symbol $[\!]$. Each statement is executed atomically and is selected for execution in a weakly fair manner—in an infinite computation, each statement is scheduled for execution infinitely often. A program description begins by introducing the variables being used in the **declare** section. Abstract variable types such as sets and sequences can be used freely. The **initially** section defines the allowed initial conditions for the program. If a variable is not referenced in this section, its initial value is constrained only by its type. The heart of any Mobile UNITY program is the **assign** section consisting of a set of labeled conditional assignment statements of the form:

$$label :: var_1, var_2, \ldots, var_n := expr_1, expr_2, \ldots, expr_n \textbf{ if } cond$$

where the optional *label* associates a unique identifier with a statement. The guard *cond*, when false, reduces the statement execution to a skip.

As in UNITY, Mobile UNITY also provides quantified assignments, specified using a three-part notation:

$$label :: \langle \| \; vars : condition :: assignment \rangle$$

where *vars* is a list of variables, *condition* is a boolean expression that defines a range of values, and *assignment* is an assignment statement. For every instance of the variables in *vars* satisfying the *condition*, an *assignment* statement is generated. All generated assignments are performed in parallel. (This three part notation is used for other operations besides quantified assignment. For example, the $\|$ can be replaced with a '+', and all generated expressions are added together and a value is returned). Though not provided in the original model, the nondeterministic assignment [1] proved to be useful in many formalizations. A nondeterministic assignment statement such as $x := x'.Q$, assigns to x a value x' nondeterministically selected from among the set of values satisfying the predicate Q.

In addition to the types of assignment statements described above, Mobile UNITY also provides a *transaction* (not present in UNITY) for use in the **assign** section. Transactions capture a form of sequential execution whose net effect is a large-grained atomic state change (in the absence of reactive statements, which are explained later in this section). A transaction consists of a sequence

of assignment statements which must be scheduled in the specified order with no other statements interleaved in between. The notation for transactions is:

$$label :: \langle s_1; s_2; \ldots; s_n \rangle$$

The term *normal statement*, or simply *statement*, will be used to denote both transactions and the stand-alone assignments discussed earlier to constrast them with *reactive statements* introduced later in this section. As previously stated, normal statements are selected for execution in a weakly fair manner and executed one at a time. The guards of all normal statements can be strengthened without actually modifying the statement itself by employing *inhibit statements* of the form:

inhibit *label* **when** *cond*

where *label* refers to some statement in the program and *cond* is a predicate. The net effect is conjoining the guard of the named statement with the negation of *cond* at runtime, thus inhibiting execution of the statement when *cond* is true.

A construct unique to Mobile UNITY is the *reactive statement*:

s **reacts-to** *cond*

where *s* is an assignment statement (not a transaction) and *cond* is a predicate. The basic idea is that reactions are triggered by any assignment establishing the reactive condition *cond*. The semantics are a bit more complex since a program (or a *system*, which will be defined later) can contain many reactive statements. Operationally, one can think of each assignment (appearing alone or as part of a transaction) as being extended with the execution of all defined reactions up to such a point that no further state changes are possible by executing reactive statements alone. More formally, the set of all reactive statements forms a program $\Re$ that is executed to fixed point after each atomic state change by assignments appearing alone or within a transaction. Clearly, $\Re$ must be a terminating progrm. The result is a very powerful construct that can easily capture the effects of interrupts, dynamic system reconfigurations, etc.

Illustration. A sample Mobile UNITY program is shown in Figure 1. The program text specifies the actions of a baggage cart that moves along a track, loading at one end of the track and unloading at the other end. The program *Cart* defines variables y and λ of type **integer** in the **declare** section; y represents the size of the cart's current load, and λ can be thought of as the cart's location on the track. The **initially** section states that the cart is empty at the start of execution. Note that λ is not given a value in the initially section; λ can take on any value of type integer at the beginning of program execution. The **assign** section of *Cart* illustrates the use of several Mobile UNITY constructs.

Statement *load* is a simple conditional non-deterministic assignment statement that places a load in the cart (represented by the non-deterministic choice of a positive integer) if the cart is located at position 0 and is empty. The statements *go_right* and *go_left* are simple assignment statements that update the

cart's location on the track. The first inhibit statement prevents the execution of the *go_right* statement when the cart is empty. Similarly, the next inhibit statement prevents the cart from moving left when the cart is not empty. The next statement, *unload*, assigns to y a value of 0 if the cart is not empty and the cart is located at position N, effectively emptying the cart. The two statements following the *unload* statement are reactive statements. In the first, the reactive statement is enabled when the cart is at a position less than 0. If after the execution of a normal statement in the program, this statement becomes enabled, the cart's position is updated to a legal position (position 0) on the track. Similarly, the second reactive statement, when enabled, will force the cart in a legal position on the track, position N.

program *Cart*
  declare
    y, λ : integer
  initially
    $y = 0$
  assign
    load $:: y := y'.(y' > 0)$ **if** $\lambda = 0 \wedge y = 0$
  [] *go_right* $:: \lambda := \lambda + 1$
  [] *go_left* $:: \lambda := \lambda - 1$
  [] **inhibit** *go_right* **when** $y = 0$
  [] **inhibit** *go_left* **when** $y \neq 0$
  [] *unload* $:: y := 0$ **if** $\lambda = N \wedge y \neq 0$
  [] $\lambda := 0$ **reacts-to** $\lambda < 0$
  [] $\lambda := N$ **reacts-to** $\lambda > N$
end

Fig. 1. An example Mobile UNITY program

Proof Logic. Mobile UNITY has an associated proof logic by and large inherited directly from UNITY. Program properties are expressed using a small set of predicate relations whose validity can be derived directly from the program text or from other properties through the application of inference rules.

Basic safety is expressed using the **unless** relation. For two state predicates p and q, the expression p **unless** q means that for any state satisfying p and not q, the next state in the execution sequence must satisfy either p or q. There is no requirement for the program to reach a state that satisfies q, i.e., p may hold forever. Progress is expressed using the **ensures** relation. The relation p **ensures** q means that for any state satisfying p and not q, the next state must satisfy p or q. In addition, there is some statement that guarantees the establishment of q if executed in a state satisfying p and not q. Note that the **ensures** relation is not itself a pure liveness property, but is a conjunction of a safety and a liveness property. The safety part of the **ensures** relation can be expressed as an **unless** property, and the existence of an establishing statement

can be proven with standard techniques. In UNITY, the two predicate relations are defined by:

$$p \textbf{ unless } q \equiv \langle \forall s : s \textbf{ in } P :: \{p \wedge \neg q\} s \{p \vee q\} \rangle$$

$$p \textbf{ ensures } q \equiv (p \textbf{ unless } q) \wedge \langle \exists s : s \textbf{ in } P :: \{p \wedge \neg q\} s \{q\} \rangle$$

where s is a statement in the program P.

The distinction between UNITY and Mobile UNITY becomes apparent only when we consider the manner in which we prove Hoare triples, due to the introduction of transactions and reactive statements. For instance, in UNITY a property such as:

$$\{p\} s \{q\} \textbf{ where } s \textbf{ in } P$$

refers to a standard conditional multiple assignment statement s exactly as it appears in the text of the program P. By contrast, in a Mobile UNITY program we will need to use:

$$\{p\} s^* \{q\} \textbf{ where } s \in \aleph,$$

and $\aleph$ denotes the normal statements of P while s^* denotes a statement s modified to reflect the guard strengthing caused by inhibit statements and the extended behavior resulting from the execution of the reactive statements in the reactive program $\Re$ consisting of all reactive statements in P. The following inference rule captures the proof obligations associated with verifying a Hoare triple in Mobile UNITY under the assumption that s is not a transaction:

$$\frac{p \wedge \iota(s) \Rightarrow q, \{p \wedge \neg \iota(s)\} s \{H\}, H \mapsto (FP(\Re) \wedge q) \text{ in } \Re}{\{p\} s^* \{q\}}$$

For each non-reactive statement s, $\iota(s)$ is defined to be the disjunction of all **when** predicates of inhibit clauses that name statement s. Thus, the first part of the hypothesis states that if s is inhibited in a state satisfying p, then q must be true of that state also. $\{p \wedge \neg \iota(s)\} s \{H\}$ (from the hypothesis) is taken to be a standard Hoare triple for the non-augmented statement s. H is a predicate that holds after execution of s in a state where s is not inhibited. It is required that H leads to fixed-point and q in the reactive program $\Re$.

For transactions of the form $\langle s_1; s_2; \ldots s_n \rangle$ the following inference rule can be used before application of the one above:

$$\frac{\{a\} \langle s_1; s_2; \ldots s_{n-1} \rangle^* \{c\}, \{c\} {s_n}^* \{b\}}{\{a\} \langle s_1; s_2; \ldots s_n \rangle^* \{b\}}$$

where c may be guessed at or derived from b as appropriate. This represents sequential composition of a reactively-augmented prefix of the transaction with its last sub-action. This rule can be used recursively until we have reduced the transaction to a single sub-action. This rule may seem complicated, but it represents standard axiomatic reasoning for ordinary sequential programs, where each sub-statement is a predicate transformer that is functionally composed with others.

System specification. So far, the notation and logic of Mobile UNITY have been discussed in terms of a single program. However, Mobile UNITY structures computations in systems consisting of multiple components and coordination rules that govern their interactions. Each component is a program with unique variable names. Programs are defined as instantiations of program types. Program type definitions are followed by a **Components** section that establishes the overall system configuration and some initialization parameters and by an **Interactions** section consisting of coordination constructs used to capture the nature of the data transfers among the decoupled component programs.

A **System** description begins by providing paramaterized type definitions for the programs to be composed. A type definition of a program is simply program text that has a parameter used only to identify an instantiation of a program. Type definitions are similar to macros in that the textual type definition of a program can be substituted for a program instantiation anywhere within the **System**.

In the **Components** section of a system, component programs are instantiated using the name of a type definition and a parameter value to identify the instantiated program. The **Components** section assumes a form such as:

$$programA(1) [] programA(2) [] programB(1)$$

where *programA(i)* and *programB(j)* are type definitions in the system, and *programA(1)*, *programA(2)*, and *programB(1)* are the desired program instantiations.

Instantiated programs making up a **System** in Mobile UNITY have disjoint namespaces. The separate namespaces for programs hide variables and treat them as internal by default, instead of being universally visible to all other components as is the case in UNITY program composition. Formally, uniqueness of variable names in Mobile UNITY systems is achieved by implicitly prepending the name of the component to that of each variable, e.g., *programA(1).x*, *programB(1).x*. This facilitates modular system specification and impacts the way program interactions are specified for those situations where programs must communicate. Coordination among programs in Mobile UNITY is facilitated by defining rules for program interaction in the **Interactions** section of a system.

The **Interactions** section of the system specification defines inter-process communication. As mentioned previously, programs in Mobile UNITY cannot interact with each other in the same style as in UNITY (by sharing identically named variables) because they have distinct namespaces. Instead, special constructs must be provided to facilitate program interaction. These rules must be explicitly defined in the **Interactions** section of a system, using fully-qualified variable names. Since in mobile computing systems, interaction between components is transient and location-dependent, the **Interactions** section often restricts communication based on variables representing location information. Reactive statements, inhibit statements, and assignment statements can appear in the **Interactions** section. Here, however, references to variables that cross program boundaries are permitted.

Illustration. Figure 2 shows a system called *BaggageTransfer*. It is based upon a restructuring of the earlier *Cart* program designed to separate the cart, loading, and unloading actions. Three types of components are used: *Cart(k)*, *Loader(i)*, and *Unloader(j)*. Each program type is parmeterized so as to allow for the creation of multiple instances of the same type.

```
System BaggageTransfer
  program Cart(k)
    declare
      y, λ : integer
    initially
      y = 0
    assign
      go_right   :: λ := λ + 1
    [] go_left    :: λ := λ − 1
    [] inhibit go_right when y = 0
    [] inhibit go_left when y ≠ 0
    [] λ := 0 reacts-to λ < 0
    [] λ := N reacts-to λ > N
  end

  program Loader(i)
    declare
      x : integer
    initially
      x = 0
    assign
      load :: x := x'.(x' > 0)
  end

  program Unloader(j)
    declare
      z : integer
    initially
      z = 0
    assign
      unload :: z := 0
  end

  Components
    Cart(1) [] Cart(2)
  [] Loader(1) [] Unloader(1)

Interactions [1]
  Cart(k).y, Loader(i).x := Loader(i).x, 0
      when Cart(k).y = 0
           ∧ Loader(i).x ≠ 0
           ∧ Cart(k).λ = 0
[] Cart(k).y, Unloader(j).z := 0, Cart(k).y
      when Cart(k).y ≠ 0
           ∧ Unloader(j).z = 0
           ∧ Cart(k).λ = N

end BaggageTransfer
```

Fig. 2. An example Mobile UNITY system

Cart(k) defines a program in which a baggage cart is moved along a track. As before, the movement of the cart depends on the value of the program variable y, which represents the weight of the current baggage in the cart. Notice that the program type definition contains no statement in which y is explicitly assigned. *Loader(i)* defines a program in which a variable x is non-deterministically assigned a value, presumably defining a baggage weight to be loaded. *Unloader(j)* defines a program in which a variable is assigned a value of 0.

The **Components** section instantiates the component programs in the *BaggageTransfer* System. Two carts (*Cart(1)* and *Cart(2)*) are created along with a single loader and unloader. The two carts are distinguished by the distinct values given to parameter k.

[1] Though its semantics are identical to those of the **if** keyword, the **when** keyword is used for emphasis in the **Interactions** section of Mobile UNITY systems.

The **Interactions** section allows the carts, loader, and unloader program instantiations to work together to transport baggage. The first statement is an asynchronous value transfer conditional on the location of the cart and the status of the loader. Since all free variables are assumed to be universally quantified by convention, the statement describes the relationship between a typical loader and a typical cart, and so it applies to both carts. The load stored in *Loader(i).x* is transferred to the cart and stored in *Cart(k).y*. This will enable the cart to start its movement towards the unloader. In a similar fashion, the arrival of a cart at the right hand side of the track makes it possible for the load to be transferred from *Cart(k).y* to *Unloader(j).z*, later to be discarded as apparent in the code of the unloader.

As shown elsewhere [11], many different coordination constructs can be built out of the basic constructs presented so far. Among them, one of particular interest is transient and transitive variable sharing, denoted by $\approx$. For instance, the code below describes an interaction between a cart and an inspector where the cart and the inspector share variables y and w as if they denoted the same variable, when co-located. At the point when the cart and inspector become co-located, the shared variable is given the value of the cart's y variable as specified by the **engage** clause. When the cart and inspector are no longer co-located, the cart's y variable retains the value of the shared variable and the inspector's w variable is set to 0, as stated in the **disengage** clause.

$$\begin{array}{ll} \mathit{Cart(k).y} \approx \mathit{Inspector(q).w} & \textbf{when } \mathit{Cart(k).\lambda} = \mathit{Inspector(q).\lambda} \\ & \textbf{engage } Cart(k).y \\ & \textbf{disengage } Cart(k).y,\ 0 \end{array}$$

Proof Logic Revisited. The entire system can be reasoned about using the logic previously presented because it can easily be re-written as an unstructured program with the name of each variable and statement expanded according to the program in which it appears, and with all statements merged into a single **assign** section. In other words, the system structuring represents solely a notational convenience with no deep semantic implications.

3 Mobile UNITY as a General Schema for Mobility

The features that differentiate Mobile UNITY [11] from the original UNITY [4] model have been the result of a careful analysis of what is necessary to model mobile systems. Yet, the presentation so far deliberately avoided showing these constructs in the context of mobility. This is because the new features, while needed to capture mobility, can be used in a wide range of settings and for a variety of purposes. In this section, we specialize the general notation to one form of mobility and coordination representative of much of the published work on Mobile UNITY. We accomplish this by defining a schema for mobility. Other schemas specialized for other situations and related to existing models of coordination will be described in following sections.

A schema is typically defined as a syntactic (sometimes semantic) pattern which restricts the use of language constructs to specific forms that have desirable properties. Efficient implementation or accurate modeling of key application concerns are properties one may seek in defining a particular schema. In this section and in the remainder of this paper, we focus on the latter. More precisely, in this section we desire to restrict the model in a manner that allows one to directly capture systems whose components can move freely through a space and interact with each other in a location-dependent manner. Programs are defined to be the basic unit of mobility, modularity, and execution. This is a natural choice in Mobile UNITY and, fortunately, places no undue burden on the modeling process because programs can be of arbitrary complexity. Both fine-grained mobility (e.g., single statements) and coarse-grained mobility (e.g., whole components) may be expressed simply by varying the size of the programs being used. The use of program types facilitates compact representation of systems consisting of large numbers of mobile components. For now, we impose no restrictions on the size of the program code, the functions it performs, or the number of components that are being instantiated. In a given application setting, however, such restrictions may prove to be highly profitable, e.g., when one considers the case of very small devices such as sensors dedicated to evaluating one single local environmental condition, such as temperature.

System *BaggageTransfer*
  program *Cart(k)* **at** λ
    ...
  end

  program *Loader(i)* **at** λ
    ...
  end

  program *Unloader(j)* **at** λ
    ...
  end

 Components
  Cart(1) $[\!]$ *Cart(2)* $[\!]$ *Loader(1)* at 0 $[\!]$ *Loader(2)* at $N/2$
 $[\!]$ *Unloader(1)* at N $[\!]$ *Unloader(2)* at $3N/4$

 Interactions
  $Cart(k).y, Loader(i).x := Loader(i).x, 0$
    when $Cart(k).y = 0 \wedge Loader(i).x \neq 0 \wedge Cart(k).\lambda = Loader(i)\lambda$
 $[\!]$ $Cart(k).y, Unloader(j).z := 0, Cart(k).y$
    reacts-to $Cart(k).y \neq 0 \wedge Unloader(j).z = 0 \wedge Cart(k).\lambda = Unloader(j).\lambda$

end *BaggageTransfer*

Fig. 3. Sample application of the general mobility schema

Because programs are expected to be mobile, a mechanism must be introduced to capture the notion that a given component is present at a specific location and that it can move from one location to another. We choose to model location as a distinguished variable which is required to appear in all programs. Conventionally, this variable is named λ. Figure 3 depicts a new version of the *BaggageTransfer* system presented earlier, slightly modified to conform with the mobility schema presented in this section. The first thing to observe is the slight change in notation. The distinguished variable λ is pulled into the program type declaration. The resulting notation is merely a mechanical transformation designed to enforce the distinguished nature of the variable λ. Furthermore, the initialization of λ, if necessary, is relegated to the **Components** section.

By having an explicit representation of the program location as part of its state, mobility is reduced to changes in the value of λ. The type of λ is determined by the specific way in which space is modeled. In the example shown in Figure 3, we assume a discrete linear space over the range 0 to N. Other notions of space can be used with equal ease. When modeling physical movement, a latitude and longitude pair may be appropriate in defining a point in space. Logical mobility may entail the use of host identifiers. Spaces may be uniform and bounded, may be undefined in certain regions, or may extend to infinity. The operations permitted for use in changing λ are specified implicitly in the definition of the space. As before, in Figure 3 the location of the cart is incremented or decremented one unit at a time, thus faithfully representing the nature of discrete but contiguous linear movement. When the space being modeled has a specific structure, mobility requires appropriate constraints. For instance, if the space is defined as a graph, it is reasonable to expect that movement takes place along edges in the graph. In other cases, we may prefer to allow a program to change location by simply moving to any reachable node in the graph if the passage through intermediary nodes results in no local interactions. In the *BaggageTransfer* example presented in Figure 3, changes to the cart's λ variable are restricted so that the cart can only move along positions on the track. It is important to note that, by reducing movement to value assignment, the proof logic naturally covers mobility without necessity for extensions.

The manner in which programs use the location variable, in turn, may induce several subschemas. A program may be location-oblivious in the sense that it never refers to λ anywhere in its code (except for its compulsory declaration). In such cases, mobility is external to the program, however, its behavior may be affected by mobility indirectly, i.e., by the coordination rules defined in the **Interactions** section. Location-aware programs refer to the value of λ and alter their behavior based on their current location. Finally, location-controlling programs actually modify their location explicitly in their code. In Figure 3, the loader and unloader programs are location-oblivious, and the cart programs are location-controlling.

Finally, we turn our attention to the **Interactions** section. It is intended to serve as a repository for the inter-program coordination rules. In general, statements present in the **Interactions** section can be of an arbitrary nature and

are permitted to refer to variables owned by the individual programs contained within the system.

In the case of a mobile system, it is reasonable to assume that interactions are local. When mobile code is involved, interactions among programs take place whenever the components are co-located. In the presence of physical mobility, connectivity relies on wireless communication that is limited in range. Given such considerations, a reasonable restriction might be to require all statements appearing in the **Interactions** section to limit their effects only to pairs of variables in co-located programs. The co-location requirement can be a reasonable abstraction for situations in which all components within some limited region of the space can communicate with each other. Pairwise interactions may be imposed due to the nature of the underlying communication protocols (e.g., message passing). In Figure 3, both interactions are guarded by the requirement that the affected components are at the same location. In one case, whenever either cart is co-located with a loader, loading is possible but not required. In the other case, by using a reactive statement, a cart co-located with an empty unloader is guaranteed to unload.

At this point, all the essential features of this mobility schema are represented in Figure 3, but a slight generalization could result in a richer model. We achieve this by permitting the equality relation used among locations to be replaced by any arbitrary binary relation among points in space. For instance,

$$within_range(A.\lambda,\ B.\lambda) \text{ or } reachable(A.\lambda,\ B.\lambda)$$

would allow one to accommodate the notion of limited wireless transmission range or connectivity in wired networks, respectively.

In this paper, we are concerned with a particular class of mobile systems, one that emphasizes spatial and temporal decoupling among components. In coordination models, such decoupling facilitates component interactions (exhibiting various degrees of decoupling) that are more abstract than those provided by the typical communication layers. In the remainder of this paper, we will explore styles of coordination and Mobile UNITY's ability to capture their essential features in schemas.

4 Agent Mobility in Wired Networks

Agent systems represent a popular new mode of computing specifically designed to take advantage of the global space offered by the Internet. An agent is a code fragment that can move from site to site in purposeful pursuit of some task defined at its point of origin. The underlying space is a graph whose vertices denote servers willing and able to accept agents. Since Internet connectivity may be perceived to be reliable and universal, the edges in the graph represent accessibility to other sites. Each agent carries with it not only the code to be executed, but also data and control state that affect its actions at each site. The movement from one site to the next may be autonomous (subjective) or initiated by the server (objective). Agents belonging to the same community of

applications may interact with each other. In D'Agents [7], for instance, message passing, streams, and events are used to enable agents to communicate among themselves. Agent systems that stress coordination rather than communication tend to rely on tuple spaces, in the spirit of the orginal coordination modality proposed in Linda [6]. TuCSoN [17], MARS [2], and Limbo [5] are just a small sample of agent sytems that employ tuple based coordination. They provide the basis for the schema proposed below.

In examining such systems, the following features capture their essence:

- agent mobility among servers
- admission control
- coordination by means of tuple spaces located on the server
- traditional tuple space operations, e.g., **out**(*tuple*), **in**(*pattern*), **rd**(*pattern*)
- augmentation of tuple space operations with reactions that extend the effects of the basic operations to include arbitrary atomic state transitions.

A coordination schema that enforces this design style will need to distinguish between agents and servers. Syntactically this can be done by substituting **Server** or **Agent** for the keyword **Program**, as needed. For instance, one can do this by means of a macro definition of the form:

$$\begin{array}{l} \textit{Agent } X \equiv \\ \quad \textit{Program Agent_X} \end{array}$$

With this distinction, we can examine the different requirements for agent and server programs. The agent location is the location of one of the servers and the change in location can be accomplished by modifying λ to hold the value of some other server location, including the agent's home location. For reasons having to do with admission control, it is best to think of λ as holding a pair of values:

$$\begin{array}{l} \lambda \equiv \\ \quad \textit{(current_location, desired_server_relocation)} \end{array}$$

and provide the agent with a single move operation:

$$\begin{array}{l} \textit{goto}(\mathcal{S}) \equiv \\ \quad \lambda := (\lambda \uparrow 1, S) \end{array}$$

We use $var \uparrow n$ to denote the nth element held by the record-like variable var. Since checking that the new location is valid requires comparing the agent's location with the location of a server, the actual move (i.e., the assignment to λ) is relegated to the **Interactions** section.

While the agent is present at a particular server, all interactions with the server and other agents take place by sharing a single tuple space owned by the server. The variable T could be used to represent such a tuple space (assumed here to be a set of tuples) inside the agent with the **Interactions** section establishing the tuple sharing rules. Access to T is restricted to tuple space operations. The **out** operation simply adds a tuple to the set if the guard g is true:

$\mathbf{out}(t)$ **if** $g \equiv$
$\quad T := T \cup \{t\}$ **if** g.

The **in** operation is blocking and removes a tuple matching some pattern p:

$z = \mathbf{in}(p)$ **if** $g \equiv$
$\quad \langle\, \theta : \theta = \theta'.(\theta' \in T \wedge match(\theta', p)) \wedge g :: z := \theta \| T := T - \{\theta\}\rangle$

where we use the nondeterministic value selection expression $x'.Q$ to indentify one suitable tuple. If none exists, the operation is reduced to a skip. Busy waiting is the proper modeling solution for blocking operations in the Mobile UNITY context. The **rd** operation is similar to an **in**, the only difference being that the returned tuple is not removed from the tuple space.

The server also has a location λ and a variable T, but its location cannot change. For the sake of uniformity, the server's location variable must hold a pair like the agent's location variable λ, but the two fields hold identical values. Since the server is stationary, it cannot change its λ variable, and the *goto* operation is not available. However, the server needs to be aware of the presence of agents at its location, either in order to refuse admission by sending an agent back before it can have any local effects or by forcing an agent to move elsewhere when conditions demand it. The presence of an agent could be made known to the server by introducing a new variable Q in both agents and servers. On the agent, the variable contains a tuple i that identifies that agent but no operations are available to access it. The **Interactions** section will ensure that all the variables Q of all the agents are shared with the server forming a set of all agents present at that location. (The server need not store its own identity in Q.) The server can discover the presence of agents by reading the contents of Q without being able to modify it. This can be accomplished by hiding Q inside an operation such as:

AG:=*LocalAgents()* $\equiv$
$\quad AG := Q$

Finally, the server may request an agent to move to some other location by employing an operation such as:

Move A to $S \equiv$
$\quad M := (A, S)$

which places in the hidden variable M a request to move agent A to server S. The actual move is encoded in the **Interactions** section, which will determine the location of S, will use it to change the location of A, and clear the request from M.

The syntactic restrictions on agent and server code are complemented by coordination patterns built into the **Interactions** section. First, we must specify

the sharing rules governing the variables T and Q. Using the transient and transitive variable sharing of Mobile UNITY, the sharing rules become:

$$\begin{array}{l} S.T \approx A.T \ \textbf{when}\ S.\lambda \uparrow 1 = A.\lambda \uparrow 1 \\ \qquad\qquad\quad \textbf{engage}\ S.T \\ \qquad\qquad\quad \textbf{disengage}\ S.T., \emptyset \end{array}$$

$$\begin{array}{l} S.Q \approx A.Q \ \textbf{when}\ S.\lambda \uparrow 1 = A.\lambda \uparrow 1 \\ \qquad\qquad\quad \textbf{engage}\ S.Q \cup A.Q \\ \qquad\qquad\quad \textbf{disengage}\ S.Q - \{A.\iota\}, \{A.\iota\} \end{array}$$

where we assume that the initial value of $A.Q$ is permanently saved in $A.\iota$, another hidden variable.

Mobility requests are handled by introducing reactive statements designed to extend the request (a local operation) with its actual processing (a global coordination action). For instance, the objective move operation requested by the server is transformed into an equivalent hidden subjective request:

$$\begin{array}{l} A.\lambda := (A.\lambda \uparrow 1, S.\lambda \uparrow 1) \| S'.M := nil \\ \quad \textbf{reacts-to}\ A.\lambda \uparrow 1 = S'.\lambda \uparrow 1 \wedge S'.M = (A, S). \end{array}$$

This, in turn, results into two cases to consider: when A is accepted by the destination $\mathcal{S}$ and the move is carried out:

$$A.\lambda := (S.\lambda \uparrow 1, S.\lambda \uparrow 1) \ \ \textbf{reacts-to}\ A.\lambda \uparrow 2 = S.\lambda \uparrow 1 \wedge \mathit{admitted}(A.Q, S)$$

and when the move is rejected and the agent is reactivated at its current location:

$$A.\lambda := (A.\lambda \uparrow 1, A.\lambda \uparrow 1) \ \ \textbf{reacts-to}\ A.\lambda \uparrow 2 = S.\lambda \uparrow 1 \wedge \neg \mathit{admitted}(A.Q, S).$$

As an example, consider an inspector agent that moves among unloader service sites and computes the total number of packages that pass through the system. Each unloader is assumed to hold a local counter of packages. The inspector adds the local counter to its own and resets the local one. Once all sites are visited, the inspector agent returns home. Each site will reject any inspector agent that is not authorized to collect the data. By employing the schema presented in this section, the agent code for this example becomes:

program *Inspector(k)* **at** λ

always

$home_again = (\lambda = (home(k), home(k)))$

declare

...

initially

$\iota = (inspector.k, password(k))$

$[\!] \ Q = \{\iota\} \ [\!] \ T = \{\} \ [\!] \ N = 0 \ [\!] \ \lambda = (home(k), home(k))$

assign

$\langle goto(next_server(\lambda)); t := in(\langle counter, int : m\rangle);$
$N := N + t \uparrow 2;\ out(\langle counter, 0\rangle)\rangle$

$[\!] \ N := 0$ **reacts-to** $home_again$

end

One element still missing from the schema definition is the augmentation of tuple space operations with arbitrary extra behaviors. This can be accomplished by separating the initiation of an operation from its execution. An **in** operation, for instance, can be redefined as a request RQ which, in turn, can enable a programmer specified reaction on the server:

t:=**in**(p) **if** $g \equiv$

$\langle RQ := (id, in, p)$ **if** $g; t, T, tt := tt, T - \{tt\}, nil$ **if** $tt \neq nil\rangle$

$\langle \| \theta : \theta = \theta'.(\theta' \in T \wedge\ match(\theta', p)) :: tt := \theta\rangle$ **reacts-to** $RQ \uparrow 3 = p$

action **extends**$(\rho, \omega, \pi) \equiv$

action **reacts-to** $RQ \neq nil \wedge tt \neq nil \wedge \rho(RQ \uparrow 1) \wedge \omega(RQ \uparrow 2) \wedge \pi(RQ \uparrow 3)$

$\|\ RQ := nil$ **reacts-to** $RQ \neq nil \wedge tt \neq nil$

where (ρ, ω, π) specifies the criteria under which the **in** operation is extended. This illustration assumes one extension only, but it could be rewritten to accommodate multiple extensions to be applied in a nondeterministic order.

Since systems consist of components controlling local actions and interactions that extend their effects to other components, it is not suprising that the schema definition also seems to be structured along these lines: mostly syntactic restrictions of the component code (further refined by component type) and coordination patterns of a behavioral nature, restricted in scope solely to variables involved in the process of information sharing. It is this structuring of the schema definition that qualifies it as a *coordination schema*.

5 Agent Mobility in Ad Hoc Networks

In this section we explore the implication of extending the mobile agent paradigm to ad hoc networks. Ad hoc networks are formed when hosts come into wireless

contact with each other and communicate as peers in the absence of base stations and any wired infrastructure. In such settings, one can envision systems consisting of hosts that move through physical space and agents that reside on such hosts. Agents can coordinate application activities with other agents within reach, and also have the ability to move from one host to another when connectivity is available. One of the very few systems to offer these capabilities is LIME [16], which will be used as a model for the schema we explore in this section. The key features of LIME are as follows:

- each agent may create an arbitrary number of local tuple spaces, each bearing a locally distinct name
- agents coordinate by sharing identically-named tuple spaces belonging to agents on connected hosts, i.e., each agent has access to all the tuples in such combined tuple spaces (called federated tuple spaces).

In Mobile UNITY, it is convenient to represent each tuple space as a pair of variables, one holding a name and the other storing the set of locally-owned tuples that are part of that tuple space, tuples the agent is willing to share with other agents. Consequently, the tuplespace sharing rule can be easily expressed as follows:

$$
\begin{array}{l}
A.X_T \approx B.Y_T \ \textbf{when} \ \mathit{connected}(A,B) \wedge A.X_N = B.Y_N \\
\qquad \textbf{engage} \ A.X_T \cup B.Y_T \\
\qquad \textbf{disengage} \\
\qquad \langle \text{set } t, C, Z : \mathit{connected}(A,C) \wedge A.X_N = C.Z_N \\
\qquad\quad \wedge\ t \in A.X_T \wedge\ t \ \mathit{owned\ by}\ C :: t \rangle, \\
\qquad \langle \text{set } t, C, Z : \mathit{connected}(B,C) \wedge B.Y_N = C.Z_N \\
\qquad\quad \wedge\ t \in B.Y_T \wedge\ t \ \mathit{owned\ by}\ C :: t \rangle
\end{array}
$$

where *connected* is defined in terms of reachability in the ad hoc network and the extensions $_N$ and $_T$ refer to names and sets of tuples, respectively. Upon connection, the engagement value is the union of all the connected identically-named tuple spaces, and, upon disconnection, the set of tuples is repartitioned according to the new connectivity pattern. However, in order to accomplish this, the concept of tuple ownership needs a representation; we assume that each tuple includes a current location field (an agent id, ι) which allows us to define:

$$
\begin{array}{l}
t \ \mathit{owned\ by}\ C \equiv \\
\quad t.\mathit{loc} = C.\iota
\end{array}
$$

In the above, we take the liberty to assume that fields in a tuple could be referenced by name. It is interesting to note the kind of hierarchical spatial organization emerging from this schema: hosts have locations in the physical space and their wireless communication capabilities can be abstracted by a reachability predicate, not shown but implied in the definition of *connected*; agents reside on hosts or servers in a manner similar to that shown in the previous section (for this reason, we do not repeat the details of agent movement even though now it

is conditional on the availability of connectivity); tuples reside on agents, a new logical space defined by the name of the tuple space combined with that of the agent.

Since tuples have a logical location, it becomes reasonable to consider the possibility of restricting operations on tuples to particular subspaces, and to entertain the notion of tuple movement. Actually, LIME offers both capabilities. For instance, **in** and **out** operations can be restricted to a specific agent location. More interestingly, **out** operations can be targeted to a particular location, i.e., the named tuple space of a particular agent. Since the agent may not be connected at the time, the tuple is augmented with a second location field that stores the desired destination. This is reminiscent of the agent mobility treatment from the previous section but with one important difference—the tuple will continue to reside locally until such time that migration becomes possible. Migration, immediate or upon the establishment of a new connection, is captured by an interaction of the form:

$$
\begin{array}{l}
A.X_T := \\
\quad \langle \text{set } t, B, Y : t \in A.X_T \wedge \ t.dest = B \wedge \ connected(A, B) \\
\quad\quad \wedge\ A.X_N = B.Y_N :: t[loc : B; dest : B] \rangle \\
\quad \cup \\
\quad \langle \text{set } t, B, Y : t \in A.X_T \wedge \ t.dest = B \wedge \ \neg(connected(A, B) \\
\quad\quad \wedge\ A.X_N = B.Y_N) :: t \rangle \\
\quad\quad \textbf{reacts-to } true
\end{array}
$$

where we use the notation $t[field_name : newvalue]$ to denote a modification of a particularly named field in tuple t.

Since the purpose of this paper is to explore coordination schemas, we refrain from including here all features of LIME and limit ourselves to noting that a complete formalization of LIME in terms of Mobile UNITY has been performed [15]. The features that were discussed in this section demonstrate the applicability of the model to an area of computing of growing importance, one that presents new challenges to the software engineering community.

6 Mobility in Malleable Program Structures

In some systems, the definition of space is the program itself. In Mobile Ambients [3], for instance, the program consists of environments called ambients that can be embedded within each other. Mobility takes place by altering the relation among ambients, which, for mobility purposes, are treated as single units. An ambient can exit its parent and become a peer with the parent; an ambient can enter a peer ambient; and an ambient can dissolve the domain boundary of a peer ambient. All these can be done only if the name of the relevant ambient is known. This is a way to model security capabilities. Other systems, such as MobiS [9], are more restrictive in terms of the range of operations provided for mobility while others, such as CodeWeave [10], may approach mobility at a much

finer level of granularity—in CodeWeave, single statements and single variables can be moved anywhere in the program structure where the latter is distributed across hosts and is hierarchical along the lines of block-structured programming languages.

The schema we describe in this section is directly inspired by Mobile Ambients. Key points of distinction will be related to fundamental differences between a process algebra and a programming notation that does not support dynamic process creation or scope restriction. To avoid possible confusion, we will use the term spatial domain, or simply domain, to refer to the analog of an ambient. The defining features of the resulting schema are:

- hierarchical structuring of the space in terms of embedded domains that reflect directly the overall structure of the system
- protection enforcement via capabilities that rely on secret unique names
- mobility in the form of localized restructuring of the system structure.

In Mobile UNITY, a system is simply a collection of programs. One way to organize it hierarchically and still allow for dynamic reconfiguration is to impose a partial order over the set of programs in a manner that corresponds to a tree having an imaginary root. A domain is defined in terms of all the programs that share a common parent, and the name of the parent can be used to uniquely designate that domain. This can be encoded by simply setting λ to refer to the $(program, parent)$ pair of names. An assignment of location values in the **Components** section defines the initial program structure. At the start, each program is given a unique name which, as explained later, can change over time. The program instance parameter can be used for this purpose. Below is an example of a well-formed **Components** section:

$A(1)$ **at** (1, 0)
$B(1)$ **at** (1.1, 1)
$C(1)$ **at** (1.2, 1)

where A, B, and C receive hidden distinct names 1, 1.1, and 1.2, respectively. The above establishes four domains: domain 0, which contains $A(1)$; domain 1, which contains the peer components $B(1)$ and $C(1)$; domain 1.1, which is empty; and domain 1.2, which is also empty. References to domain names will be needed in the programs. For this reason we assume that a distinguished variable ι provides each program with its own name, assumed to be unique. We assume, however, that λ (the pair consisting of ι and its parent, i.e., the domain name) is not directly accessible to the individual programs, i.e., the schema rules out statements that refer to λ in any way.

To enforce some sort of scoping constraints, we simply require that program to program communication be restricted only among peers. The type of communication is not important for the remainder of this presentation, but the reader should assume that it is available in the form of tuple space coordination or synchronous message exchange. One thing that is important is the fact that program/domain names can be passed among programs.

In the spirit of Mobile Ambients, we treat naming as the critical element of any security enforcement policy. Without exception, all operations entailing mobility involve a domain name reference, and such names must be acquired via some communication channel and cannot be guessed. For instance, the exit and enter operations allow a component to move up in the structure at the level of the current parent program and to move down in the structure inside the domain associated with one of its peers, respectively. In both cases, the correct name of the parent or the sibling must be known in order for the operation to succeed. This will become apparent only when we discuss the coordination semantics expressed in the **Interactions** section since both operations reduce simply to appropriate requests for action:

$x :=$ **exit** n **if** $g \equiv$
$\quad \langle OP := (exit, n)$ **if** g; $x := true$ **if** $OP \uparrow 1 = pass$; $OP := nil\rangle$
$y :=$ **enter** n **if** $g \equiv$
$\quad \langle OP := (enter, n)$ **if** g; $y := true$ **if** $OP \uparrow 1 = pass$; $OP := nil\rangle$

The variables x and y are used to communicate back to the program that the operation succeeded. We use a transaction to set the variables x and y to the correct values after the coordination is completed.

In Mobile Ambients, **open** n dissolves the boundaries of a peer level ambient n. In our case, this is equivalent to bringing the subordinate programs to the same level as the parent. The domain does not disappear, but it does become empty. Locally, the operation is encoded again simply as a request which may or may not be satisfied:

$x :=$ **open** n **if** $g \equiv$
$\quad \langle OP := (open, n)$ **if** g; $x := true$ **if** $OP \uparrow 1 = pass$; $OP := nil\rangle$

The most subtle aspect of our attempt to create a structured navigation schema along the lines defined by Mobile Ambients is the management of domain names. In process algebras, the name restriction operator provides a powerful mechanism for generating new, unique names and for using them to enforce private communication among components. The operational approach associated with a programming notation such as Mobile UNITY forces us to consider an operational alternative that can offer comparable capabilities. Our solution is to permit domain (i.e., program) renaming. A renamed program cannot be referenced by anyone else unless the new unique name is communicated first—this is the analog of scope extension in process algebras.

The renaming operation assumes the form:

$d :=$ **rename** n **if** $g \equiv$
$\quad \langle OP := (rename, n, new())$ **if** g; $d := nil$;
$\qquad d := OP \uparrow 3$ **if** $OP \uparrow 1 = pass$; $OP := nil\rangle$

When renaming is successful, the new domain name is returned in d in order to facilitate it being communicated to other components. In principle, a component

may be able to rename itself, its domain (i.e., its parent), and its peers—as long as it has their correct names. This can be restricted further if necessary.

The **Interactions** section needs to encode the coordination rules associated with the operations above. The general pattern is to verify that the referenced name is correct, record in the variable OP this fact, and complete all necessary changes to the domain structure. We illustrate this by considering the case when a request is made to rename the current domain, i.e., the parent name:

$P.OP := (pass, n, m)$ **reacts-to** $P.OP = (rename, n, m) \wedge P.\lambda \uparrow 2 = n$
$Q.\lambda := (m, Q.\lambda \uparrow 2)$ **reacts-to** $P.OP = (pass, n, m) \wedge Q.\lambda \uparrow 1 = n$
$R.\lambda := (R.\lambda \uparrow 1, m)$ **reacts-to** $P.OP = (pass, n, m) \wedge R.\lambda \uparrow 2 = n$

The first reactive statement records the success of the renaming for the case when n is indeed the domain name containing P, the initiator of the operation. The second reactive statement changes the domain name while the third changes the domain reference in all the components associated with the renamed domain. Similar code can be used to process exit, enter, and all other open requests.

As an illustration let us consider two programs P and Q which desire to share private information in a protected domain, and let us assume the existence of a third program S. Initially P, Q, and S are assumed to be part of some domain U, as shown in Figure 4a.

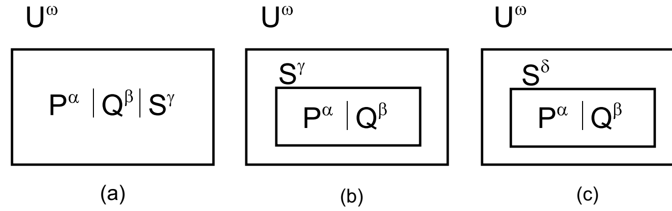

Fig. 4. Domain configurations

We use superscripts to denote the domain names. Assuming that P and Q know the name γ of S, they both can issue the operation **enter** γ changing the configuration to that shown in Figure 4b.

At this point, P can rename S with a new unique name δ and communicate the name δ to Q. The resulting configuration is shown in Figure 4c. Now, both P and Q can exit and enter S at will with no risk that any other program might be able to enter their private domain.

One problem this example ignores is the situation that some other program R may have entered S prior to P and Q. While R is trapped forever (R cannot perform any operations on S because the name of S is changed), R could interfere with the data exchanges between P and Q. There are several ways to avoid this

situation. One interesting solution is to allow P to know the cardinality of its domain, i.e., the number of components in S.

7 Location-Sensitive Synchronous Communication

The study of synchronous communication has its origins in CSP [8] and was later refined with the introduction of an entire family of process algebras, including CCS [13] and π-calculus [14]. Most often, events are identified by naming a communication channel and are differentiated as being send ($\overline{c}$) and receive (c) events associated with distinct processes. Pairs of matching events are executed simulataneously. If data is actually being exchanged, the typical notation is $c!x$ for sending a value and $c?x$ for receiving a value. In principle, many pairs of processes could be synchronized using the same channel name with matching pairs being selected nondeterministically. In π-calculus it becomes possible to protect access to a specific channel by creating new channel names and communicating them to other specific processes. In this section, we explore a schema that has these kinds of features and we show how communication can be constrained based on the relative location among processes. Let us consider first a generic event model, á la CSP [8], in which a process can send ($c!x$) or receive ($c?x$) values on a specified channel. Since Mobile UNITY programs are not sequential in nature, blocking will be interpreted as no additional operations being permitted to take place on the respective channel. Finally, we allow a process to use the same channel in both directions, but not at the same time.

Under these assumptions, an output operation may assume the following syntax and semantics:

$$\begin{aligned} & c!x \textbf{ if } g \equiv \\ & \quad c := (out, x) \textbf{ if } g \wedge c = nil \end{aligned}$$

The local view of the channel stores a request for output, if not already in use. If the guard is passable and the channel is not in use, a request for an input operation works similarly, but requires the transfer of the channel value to a specified variable:

$$\begin{aligned} & c?x \textbf{ if } g \equiv \\ & \quad c := (in, nil) \textbf{ if } g \wedge c = nil \ [\!] \ x, c := c \uparrow 2, nil \ \textbf{ reacts-to } c \uparrow 2 \neq nil \end{aligned}$$

The reaction guarantees the immediate data transfer and the resetting of the channel state.

As before, the actual coordination takes place in the **Interactions** section. The standard solution is to simply match pairs of pending input/output operations present in different processes and involving the same channel:

$$P.c, Q.c := nil, (in, P.c \uparrow 2) \textbf{ when } P.c \uparrow 1 = out \wedge Q.c \uparrow 1 = in$$

where process names P and Q and channel name c are universally quantified.

Another alternative is to carry out communications as soon as they are feasible. This can be accomplished simply by replacing the asynchronous transfer above by a corresponding reactive statement:

$$P.c, Q.c := nil, (in, P.c \uparrow 2) \ \textbf{reacts-to}\ P.c \uparrow 1 = out \wedge Q.c \uparrow 2 = in$$

The final variation on this theme is to restrict such communications to situations in which P and Q are co-located, as in:

$$\begin{array}{l} P.c, Q.c := nil, (in, P.c \uparrow 2) \\ \quad \textbf{reacts-to}\ P.c \uparrow 1 = out \wedge Q.c \uparrow 2 = in \wedge P.\lambda = Q.\lambda. \end{array}$$

So far we assumed that the channel names were fixed and the lack of any protection against unauthorized usage. In order to capture the unique ability of π-calculus to create new channel names that can be passed around among processes, we need to distinguish between the variable used to refer to a channel and the channel name. By storing the channel name, it becomes possible for it to be changed and shared. Surprisingly, the changes in the schema are relatively straightforward. First, we assume the existence of a function that returns a unique system-wide name that can be stored in a local program variable and cannot be forged:

$$\eta := new().$$

Second, we alter the structure of the local channel to accept a new name, but only when not in use:

$$\begin{array}{l} c\ \textbf{named}\ \eta\ \textbf{if}\ g \equiv \\ \quad c := (\eta, nil)\ \textbf{if}\ g \wedge c \uparrow 2 = nil \end{array}$$

The send and receive operations are altered so as to not impact the channel name. The requests are stored in the second field associated with the local view of the channel:

$$\begin{array}{l} c!x\ \textbf{if}\ g \equiv \\ \quad c := (c \uparrow 1, (out, x))\ \textbf{if}\ g \wedge c \uparrow 2 = nil \\ c?x\ \textbf{if}\ g \equiv \\ \quad c := (c \uparrow 1, (in, nil))\ \textbf{if}\ g \wedge c \uparrow 2 = nil \\ \ [\!]\ x, c := c \uparrow 2 \uparrow 2, (c \uparrow 1, nil)\ \ \textbf{reacts-to}\ c \uparrow 2 \uparrow 2 \neq nil \end{array}$$

Finally, input/output commands are matched based on names associated with the individual channels, and, if desired, constrained to co-location:

$$\begin{array}{l} P.a, Q.b := (P.a \uparrow 1, nil), (Q.b \uparrow 1, (in, P.a \uparrow 2 \uparrow 2)) \\ \quad \textbf{where}\ P.a \uparrow 1 = Q.b \uparrow 1 \wedge P.a \uparrow 2 \uparrow 1 = out \\ \qquad \wedge\ Q.b \uparrow 2 \uparrow 1 = in \wedge P.\lambda = Q.\lambda. \end{array}$$

The result is an interesting combination of a mobility schema with a dynamic reconfiguration schema.

8 Conclusion

The premise of this position paper has been the notion that effective use of powerful models in practical settings must rely on disciplined use of the model. One formal strategy for establishing such a discipline of thought and design is to impose an appropriate programming schema, i.e., mostly syntactic restrictions over the model. We explored the schema-based approach in the particular domain of coordination models and languages for mobility and produced evidence that Mobile UNITY can be readily customized to accommodate a diverse set of coordination models in a manner that maintains the clean separation between fully decoupled component actions and the coordination semantics that tie them together. We view the examples we provided mostly as exercises designed to illustrate ideas rather than solve specific problems or define novel formal strategies. Nevertheless, some of the illustrations offer practical guidelines for the use of a formal notation such as Mobile UNITY as the basis for precise definitions of coordination-based middleware constructs. At the same time, some of the exercises suggest interesting new investigative paths deserving of a more careful formal analysis.

Acknowledgements
This research was supported in part by the National Science Foundation under Grant No. CCR-9970939 and the Office of Naval Research under ONR MURI research contract N00014-02-1-0715. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

1. R. J. R. Back and K. Sere. Stepwise refinement of parallel algorithms. *Science of Computer Programming*, 13(2–3):133–180, 1990.
2. G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A programmable coordination architecture for mobile agents. *IEEE Internet Computing*, 4(4):26–35, 2000.
3. L. Cardelli and A. Gordon. Mobile ambients. *Theoretical Computer Science, Special Issue on Coordination*, 240(1):177–213, June 2000.
4. K. M. Chandy and J. Misra. *Parallel Program Design: A Foundation*. Addison-Wesley, New York, NY, 1988.
5. N. Davies, S. Wade, A. Friday, and G. Blair. Limbo: A tuple space based platform for adaptive mobile applications. In *Proceedings of the International Conference on Open Distributed Processing/Distributed Platforms (ICODP/ICDP '97)*, pages 291–302, May 1997.
6. D. Gelernter. Generative communication in Linda. *ACM Computing Surveys*, 7(1):80–112, Jan. 1985.

7. R. Gray, D. Kotz, G. Cybenko, and D. Rus. D'agents: Security in a multiple-language, mobile agent system. In G. Vigna, editor, *Mobile Agents and Security*, volume 1419 of *Lecture Notes in Computer Science*, pages 154–187. 1998.
8. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
9. C. Mascolo. MobiS: A specification language for mobile systems. In *Proceedings of 3rd Int. Conf. on Coordination Models and Languages*, volume 1594, pages 37–52. Springer-Verlag, 1999.
10. C. Mascolo, G.P. Picco, and G.-C. Roman. A fine-grained model for code mobility. In *Proceedings of the Seveth European Software Engineering Conference ESEC*, volume 1687 of *Lecture Notes in Computer Science*, pages 39–56. Springer-Verlag, September 1999.
11. P.J. McCann and G.-C. Roman. Compositional programming abstractions for mobile computing. *IEEE Transactions on Software Engineering*, 24(2):97–110, 1998.
12. P.J. McCann and G.-C. Roman. Modeling Mobile IP in Mobile UNITY. *ACM Transactions on Software Engineering and Methodology*, 8(2):115–146, April 1999.
13. R. Milner. *Communication and Concurrency*. Prentice Hall, London, 1989.
14. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992.
15. A.L. Murphy. *Enabling the Rapid Development of Dependable Applications in the Mobile Environment*. PhD thesis, Washington University in St. Louis, August 2000.
16. A.L. Murphy, G.P Picco, and G.-C. Roman. Lime: A middleware for physical and logical mobility. In *Proceedings of the 21st International Conference on Distributed Systems*, pages 524–533. IEEE Computer Society Press, April 2001.
17. A. Omicini and F. Zambonelli. The TuCSoN coordination model for mobile information agents. In *Proceedings of the 1st Workshop on Innovative Internet Information Systems*, June 1998.
18. G.P. Picco, G.-C. Roman, and P.J. McCann. Reasoning about code mobility in Mobile UNITY. *ACM Transactions on Software Engineering and Methodology*, 10(3):338–395, 2001.
19. G.-C. Roman and P. J. McCann. A notation and logic for mobile computing. *Formal Methods in System Design*, 20:47–68, 2002.