Report Number: WUCSE-2004-19

2004-04-22

A Principled Exploration of Coordination Models

Gruia-Catalin Roman and Jamie Payton

Coordination is a style of interaction in which information exchange among independent system components is accomplished by means of high-level constructs designed to enhance the degree of decoupling among participants. A de-coupled mode of computation is particularly important in the design of mobile systems which emerge dynamically through the composition of independently developed components meeting under unpredictable circumstances and thrust into achieving purposeful cooperative behaviors. This paper examines a range of coordination models tailored for use in mobile computing and shows that the constructs they provide are reducible to simple schema definitions in Mobile UNITY. Intellectually, this exercise contributes... **Read complete abstract on page 2.**

A Principled Exploration of Coordination Models

Gruia-Catalin Roman and Jamie Payton

Complete Abstract:

Coordination is a style of interaction in which information exchange among independent system components is accomplished by means of high-level constructs designed to enhance the degree of decoupling among participants. A de-coupled mode of computation is particularly important in the design of mobile systems which emerge dynamically through the composition of independently developed components meeting under unpredictable circumstances and thrust into achieving purposeful cooperative behaviors. This paper examines a range of coordination models tailored for use in mobile computing and shows that the constructs they provide are reducible to simple schema definitions in Mobile UNITY. Intellectually, this exercise contributes to achieving a better operational-level understanding of the relation among several important classes of models of mobility. Pragmatically, this work demonstrates the immediate applicability of Mobile UNITY to the formal specification of coordination constructs supporting mobile computing. Moreover, the resulting schemas are shown to be helpful in reducing the complexity of the formal verification effort.

Department of Computer Science & Engineering

Washington University in St. Louis
School of Engineering & Applied Science

2004-19

A Principled Exploration of Coordination Models

Authors: Roman, Gruia-Catalin; Payton, Jamie

April 22, 2004

Abstract: Coordination is a style of interaction in which information exchange among independent system components is accomplished by means of high-level constructs designed to enhance the degree of decoupling among participants. A decoupled mode of computation is particularly important in the design of mobile systems which emerge dynamically through the composition of independently developed components meeting under unpredictable circumstances and thrust into achieving purposeful cooperative behaviors. This paper examines a range of coordination models tailored for use in mobile computing and shows that the constructs they provide are reducible to simple schema definitions in Mobile UNITY. Intellectually, this exercise contributes to achieving a better operational-level understanding of the relation among several important classes of models of mobility. Pragmatically, this work demonstrates the immediate applicability of Mobile UNITY to the formal specification of coordination constructs supporting mobile computing. Moreover, the resulting schemas are shown to be helpful in reducing the complexity of the formal verification effort.

A Principled Exploration of Coordination Models

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}@wustl.edu

Coordination is a style of interaction in which information exchange among independent system components is accomplished by means of high-level constructs designed to enhance the degree of decoupling among participants. A decoupled mode of computation is particularly important in the design of mobile systems which emerge dynamically through the composition of independently developed components meeting under unpredictable circumstances and thrust into achieving purposeful cooperative behaviors. This paper examines a range of coordination models tailored for use in mobile computing and shows that the constructs they provide are reducible to simple schema definitions in Mobile UNITY. Intellectually, this exercise contributes to achieving a better operational-level understanding of the relation among several important classes of models of mobility. Pragmatically, this work demonstrates the immediate applicability of Mobile UNITY to the formal specification of coordination constructs supporting mobile computing. Moreover, the resulting schemas are shown to be helpful in reducing the complexity of the formal verification effort.

1 Introduction

Mobile devices equipped with wireless communication capabilities have become essential tools in everyday life. As society becomes increasingly familiar with and discovers new uses for mobile devices, we can expect heightened demand for new applications designed with mobility in mind. Logical mobility allows for software components to migrate from one host to another. Physical mobility entails changes in location and connectivity. Many systems entail both forms of mobility. Consider, for instance, the case of a tourist carrying a miniature PDA in his shirt pocket. Upon entering a museum, a mobile agent may migrate to the PDA to help the tourist navigate through the exhibits. The agent, in turn, communicates the tourist's location and preferences to the display units on the wall, thus making it possible for custom presentations to accompany the visitor from one hall to another. The development of such applications entails not only the introduction of new concepts but also new levels of complexity, which is compounded as the system becomes increasingly dynamic. A museum guide, while employing both physical and logical mobility, is relatively simple to design because of the highly constrained and structured environment in which it

functions. By comparison, an application that executes in a fully ad hoc network formed by vehicles that travel at high speeds over the highway is significantly more difficult to construct. Environments that are open and subject to rapid evolution are likely to pose the greatest challenges for the software engineer.

A design strategy that promises to faciliate the development of mobile applications is the use of coordination middleware. The idea is to simplify the development effort by offering the software engineer powerful high level constructs for component interaction within the confines of a known programming language. The programming gains are the result of a careful formulation of an appropriate conceptual model and its packaging as a set of coordination primitives accessible in the form of a standard API (Application Program Interface). Precise semantics are critical to effective use of such middleware. Unfortunately, the semantic treatment of coordination middleware has been highly polarized. Formalists found the intellectual challenges of developing new classes of models exciting and concentrated their attention on applying sophisticated skills to formalizing a range of new concepts and constructs—almost solely focusing on formal exercises without paying much attention to the software engineering process. Pragmatists delivered new kinds of middleware motivated by the needs of various application domains, rarely being concerned with a proper formalization of the constructs they provide. Our goal is to bridge the gap between formal models of mobility and the pragmatics of dependable software development, with a focus on precise semantic definitions of coordination constructs. By necessity, our treatment will be less formal than is the custom in the theoretical computer science community and more formal than most software engineering approaches. This is because we pursue a very pragmatic strategy for bridging formal thinking and engineering practice. We do this in a very narrow band, focusing on the precise semantic definition of coordination constructs, which is fundamental to achieving correct usage of the middleware and supporting informal verification of the resulting applications.

The starting point for any precise semantic definition is always a formal model. Many such models have been proposed with the most visible among them being Mobile Ambients [3], π-calculus [15], and Mobile UNITY [13, 20]. Their individual perspectives on mobility are quite distinct and entail only limited overlap in concepts and methods. Mobile Ambients structures space hierarchically and limits movement to conform to this hierarchical organization; the (possibly changing) structure of the space is actually the structure of the dynamic system being described. In π-calculus, mobility is reduced to the reorganization of the communication structure by allowing the creation of new unique channel names and the passing of such names among processes for the purpose of establishing private communication channels. Both models fall in the general category of process algebras and lend themselves naturally to a treatment of mobility that entails mostly evolutionary change rather than a more explicit notion of space. Because of their popularity, many variants of these models have been proposed. Included among them are the join calculus [5], which restricts interactions to a particular sites, and the nomadic π-calculus (related to the Nomadic Pict[21]

language) which covers a variety of low-level and high-level communication primitives. Mobile UNITY is fundamentally different from all these models: it is a state transition model that employs an explicit (albeit abstract) notion of space, and relies on an assertional proof logic for program verification.

Our strategy to relating these different models to each other centers on representing their key concepts within the Mobile UNITY framework. We have been successful in doing so in other settings by exploiting the decoupling Mobile UNITY provides between programs (as units of execution, mobility and modularity) and the interactions among them (the rules by which information is exchanged between programs which otherwise appear unable to communicate with each other). Our past experience with CodeWeave [11] and LIME [16] are particularly relevant in this respect. CodeWeave is a model developed to support fine-grained code mobility, i.e., the unit of mobility is allowed to be as small as a single variable or an individual program statement. LIME [16] is a model and middleware that adapts the Linda coordination model to support communication in ad hoc networks; its primitives have been formally specified in Mobile UNITY. In this paper we build directly on these past experiences, but focus our attention on exploring a broader range of interaction styles. The specific modes of interaction we consider are inspired by existing middleware for agent coordination, formal models of mobility, and models of communication.

In each case, we capture the essential features of a coordination model by providing a schema that consists of a set of macros that appear in the code for agents making up the system and an operational specification of the interaction rules. The former capture local actions while the latter defines coordination activities that span multiple agents. We provide schemas that encompass processes that employ location-sensitive synchronous communication (á la CSP [9]), processes that create and pass private channel names (á la π-calculus), mobile agent systems for wired networks (á la TuCSoN [18], MARS [2], and Limbo [4]), mobile agent systems for mobile ad hoc networks (á la LIME), and programs with malleable structures (á la Mobile Ambients).

At first sight, the process of building the various schemas appears to be merely an exercise in elegant coding of a series of coordination constructs into a notation system specifically designed to provide support for compact expression of coordination processes. In actuality, much more is accomplished along the way. First, this is the very first attempt to examine formally the relation among three important models of mobility. The question regarding whether Mobile UNITY is able to capture the essential features of π-calculus and Mobile Ambients has been finally resolved in the affirmative. While other methods for comparison could have been considered, we view the operational approach employed in this paper as being more in tune with software engineering practice and more likely to lead to the development of middleware inspired by some of these models. Second, the manner in which tuple space coordination constructs are given formal operational semantics in terms of Mobile UNITY offers a practical illustration of how to generate precise semantic specifications for mobility middleware. The net result is that of offering the software engineering community a valuable intellectual

tool for exploring coordination alternatives and their semantics. It is precisely the operational style of our investigation that makes this possible.

While a purely operational approach could be construed as being less abstract and a possible impediment to formal analysis, Mobile UNITY includes an associated assertional style proof logic, and models captured in terms of Mobile UNITY can be subject to formal verification. Actually, when a particular schema is used, it becomes possible to carry out the formal verification of the overall agent system without having to consider the details of the coordination mechanics. In this paper we show how an abstract semantic definition of the coordination constructs can be built, verified, and reused in proofs for systems that employ the same schema. In turn, this leads to a simplification of the verification process.

An introduction to Mobile UNITY, the model used in this paper as a foundation for the development of coordination schemas, is given in Section 2. In each of Sections 3 through 7, a Mobile UNITY formalization of a coordination model is given. Section 8 provides an overview of how formal verification is employed when schemas are used. Finally, conclusions are presented in Section 9.

2 The Essence of Mobile UNITY

In this section, we give a brief introduction to the Mobile UNITY [13, 20] model. We describe the concept of a Mobile UNITY system, which encapsulates a set of programs and governs their interactions. We begin by introducing the foundational element of all systems—the program. The notation used to specify a Mobile UNITY program is presented and applied to a simple example, a program that specifies the actions of a mobile baggage cart. The proof logic associated with Mobile UNITY programs is discussed. We then describe how a Mobile UNITY system captures program descriptions, their instantiations, and the interactions between instantiated programs. We describe the notation used to specify a system, and illustrate its application, expanding upon the baggage cart example to create a baggage transport system. Finally, we discuss the associated proof logic for a Mobile UNITY system.

2.1 Program Specification

Programs are defined to be the basic units 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., the movement of single statements) and coarse-grained mobility (e.g., the movement of whole components) may be expressed simply by varying the size of the programs being used. 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 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.

Notation. As in UNITY, the key elements of a Mobile UNITY 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 with a **declare** section that introduces the variables used. 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.

Like 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 also 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 UNITY model, the nondeterministic assignment [1] proved to be useful in many formalizations, and is sometimes included in presentations of the UNITY and Mobile UNITY models. 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 aforementioned types of assignment statements provided in the original UNITY model, Mobile UNITY also provides a *transaction* for use in the **assign** section. Transactions capture a form of sequential execution whose net effect is an atomic state change, on a scale larger than that of a simple assignment. A transaction consists of a sequence of assignment statements which must be scheduled in the specified order with no other statements interleaved. The notation for transactions is:

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

where s_i for $i = 1$ to n is a stand-alone assignment, as discussed earlier. The term *normal statement*, or simply *statement*, will be used to denote both transactions and the stand-alone assignments.

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 another Mobile UNITY construct, the *inhibit statement*:

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 powerful 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 more complex, since a program (or a *system*, which will be defined in the next subsection) 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 program. The result is a powerful construct that can easily capture the effects of interrupts, dynamic system reconfiguration, etc.

The above constructs have resulted from a careful analysis of what is necessary to model mobile systems. However, because programs are expected to be mobile, a mechanism is still needed to capture the notion that a given component is present at a specific location and that it can move from one location to another. To address this need, location is modeled in Mobile UNITY as a distinguished variable which is required to appear in all programs. Conventionally, this variable is named λ.

By having an explicit representation of the program location as part of a program's 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. For example, 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. 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.

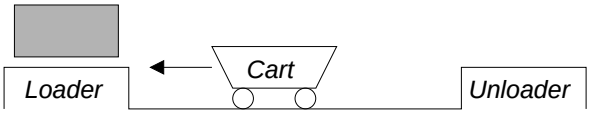

Fig. 1. A baggage transfer example

Illustration. A sample Mobile UNITY program, *Cart*, is illustrated in Figure 1. The purpose of the cart is to transport baggage from one place to another. The program text in Figure 2 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 space in which the cart moves, i.e., along the track, is assumed to be a discrete linear space over the range 0 to N.

The program *Cart* defines a variable y of type **integer** in the **declare** section; y represents the size of the cart's current load. The **initially** section states that the cart is empty at the start of execution. Note that the distinguished variable λ is not given a value in the initially section of Mobile UNITY programs; in this example, λ can take any integer value at the beginning of program execution. (Alternatively, λ can be initialized in the **Components** section of a system description, as discussed in Section 2.2.)

The **assign** section of *Cart* illustrates the use of several Mobile UNITY constructs. The 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 location of the cart is incremented or decremented one unit at a time, faithfully representing the nature of discrete but contiguous linear movement. 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 is located at position N, effectively emptying the cart. The two statements following the *unload* statement are reactive statements. The first 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.

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. It is important to note that, by reducing movement to value assignment, the proof logic naturally covers mobility without the necessity for extensions.

program *Cart* **at** λ
  declare y : 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. 2. An example Mobile UNITY program

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 exists a 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, expressed in Hoare triple notation [8], 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:

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

where $\aleph$ denotes the normal statements of P while s^* denotes a statement s modified to reflect the guard strengthening 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. $\{p \wedge \neg\iota(s)\}s\{H\}$ is 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 both 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 rule represents the proof obligation for transactions as the sequential composition of two elements. The first element is a subsequence of the normal statements (augmented with reactive behaviors) in the transaction. We call this subsequence a *sub-action* of the transaction. The second element in the sequential composition is the last sub-action of the transaction. This proof 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 other sub-statements.

2.2 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, each having uniquely named variables. 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.

Notation. A **System** description begins by providing parameterized 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. An initial location can optionally be provided to the program instantiation as well. The **Components** section assumes a form such as:

$$\textit{programA(1)}\, [] \, \textit{programA(2)}\, [] \, \textit{programB(1)}\ \textbf{at}\ (1,2)$$

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. Notice in this example the extension to the last program instantiation. This provides *programB(1)* with an initial location, giving its distinguished λ variable a value of $(1, 2)$ at the start of its execution.

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 allowing them to be 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 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 captures 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 defined explicitly in the **Interactions** section, 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. When mobile code is involved, interactions among programs take place whenever the components are co-located. In the presence of physical mobility, interactions are allowed when components are within wireless communication range. Reactive statements, inhibit statements, and assignment statements can appear in the **Interactions** section. In contrast to the **assign** section of a program, however, references to variables that cross program boundaries are permitted.

Illustration. Figure 3 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 parameterized to allow for the creation of multiple instances of the same type.

Cart(k) defines a program in which a baggage cart is moved along a track, which ranges from position 0 to position N. As before, the movement of the cart

[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.

System *BaggageTransfer*

program *Cart(k)* **at** λ
declare y : integer
initially $y = 0$
assign
go_right $:: \lambda := \lambda + 1$
[] *go_left* $:: \lambda := \lambda - 1$
[] **inhibit** *go_right* **when** $y = 0$
[] **inhibit** *go_left* **when** $y \neq 0$
[] $\lambda := 0$ **reacts-to** $\lambda < 0$
[] $\lambda := N$ **reacts-to** $\lambda > N$
end

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

program *Unloader(j)* **at** λ
declare z : integer
initially $z = 0$
assign
unload $:: z := 0$
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 [1] *Cart(k).y* $= 0 \wedge$ *Loader(i).x* $\neq 0 \wedge$ *Cart(k).*λ = *Loader(i)*λ
[] *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).*λ = *Unloader(j).*λ

end *BaggageTransfer*

Fig. 3. An example Mobile UNITY system

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 z is assigned a value of 0.

The **Components** section instantiates the component programs in the *BaggageTransfer* System. To illustrate the ease with which our original baggage example can be extended to include multiple components of the same type, two carts, *Cart(1)* and *Cart(2)*, are created along with two loaders and two unloaders. The two carts are distinguished by the values given to parameter k. The loaders and unloaders are similarly distinguished.

The **Interactions** section allows the cart, 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 an empty unloader 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 [13], 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.

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

Proof Logic. 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 Location-Sensitive Synchronous Communication

The study of synchronous communication has its origins in CSP [9] and was later refined with the introduction of an entire family of process algebras, including CCS [14] and π-calculus [15]. Most often, events are identified by naming a communication channel and are differentiated as being send and receive events associated with distinct processes. Pairs of matching events are executed simultaneously; if no match exists, the communication channel is blocked until a match is available. In principle, many pairs of processes could be synchronized using the same channel name with matching pairs being selected nondeterministically.

The typical notation used for communication between processes can be easily explained. The notation $c!x$ is used for sending a value stored in x on channel c. Similarly, $c?y$ is used to express the desire to receive a value on channel c to be stored in y. In this section, we capture key features of these models, and we show how communication can be constrained based on the relative locations of processes.

Let us first consider a generic event model similar to CSP [9], in which a process can send ($c!x$) or receive ($c?y$) values on a specified channel. Figure 4 illustrates such a model. 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. To simplify the presentation of the schema, we assume that only one sender and one receiver can communicate on a particular channel. Throughout the section, we refer to the writer as process A and the reader as process B.

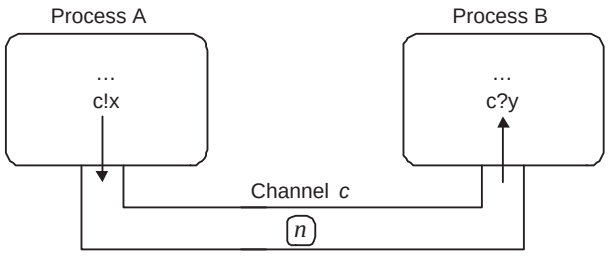

Fig. 4. An overview of the basic CSP model. In the figure, each process has access to the named channel c. Process A writes the value n contained in variable x to channel c using the syntax $c!x$. Process B reads channel c using the syntax $c?y$ and places the value of n in the variable y.

A process represents its local view of the channel as a variable, c. The value of the local channel variable is a tuple. The first field of the channel variable indicates the channel's status. The channel is either *idle*, which means that a value can be sent on the channel, or *ready*, which means that a value has been written to the channel and a process is ready to participate in the transfer of a value. The second field of the channel variable indicates the value currently available on the channel. At times, the channel is *idle* and there is no value available on the channel; this is indicated by the value $\perp$. The valid states of the channel are as follows:

- $(idle, \perp)$: indicates that the channel is available
- $(ready, v)$: indicates that the channel is busy and a value v is available for reading
- $(ready, \perp)$: indicates that the channel is busy, that the channel value has been read, and that the channel is being cleared for further use

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

$$c!x \textbf{ if } g \equiv$$
$$\quad c := (ready, x) \textbf{ if } g \wedge c \uparrow 1 = idle \qquad (s_1)$$

where the notation $c \uparrow 1$ is used to refer to the first field of the variable c. The local view of the channel stores a request for output, if the channel is not already

in use (indicated by *idle* channel status) and the guard is passable. A request for an input operation works similarly when a value is available on the channel, but requires the transfer of the channel value to a specified variable:

$$c?y \textbf{ if } g \equiv$$
$$y, c := c \uparrow 2, (ready, \bot) \textbf{ if } g \wedge\ c \uparrow 1 = ready \wedge c \uparrow 2 \neq \bot \qquad (s_2)$$

Notice that the read operation can only place a value in the specified local variable if the channel is *ready* and has a value available. The channel can only be put in such a state by the execution of a write operation on the channel. Since each process has a local representation of the channel and these channel values are not automatically shared across Mobile UNITY processes, a process executing a read must become aware somehow that a write request has been placed on the other end of the channel. This form of coordination between the two processes takes place in the **Interactions** section. A simple solution is to match pairs of pending input/output operations present in different connected processes and involving the same channel, and to transfer values accordingly:

$$B.c := A.c \textbf{ reacts-to } \textit{connected(A,B)} \wedge B.c \uparrow 1 = idle \qquad (s_3)$$
$$\wedge\ A.c \uparrow 1 = ready \wedge\ A.c \uparrow 2 \neq \bot$$

$$A.c, B.c := (idle, \bot), (idle, \bot) \textbf{ reacts-to } \textit{connected(A,B)} \qquad (s_4)$$
$$\wedge\ B.c = (ready, \bot)$$

where process names A and B and channel name c are universally quantified. The relation *connected(A,B)* is used to determine if processes are allowed to communicate. The basic *connected* relation may be defined in terms of physical connectivity, i.e., *connected(A,B)* holds if and only if processes A and B are within communication range.

Given this model of interaction, we use the **Interactions** section to address the appropriate action to take when two processes become disconnected before a channel value that is read is written locally:

$$B.c := (idle, \bot) \textbf{ reacts-to } \neg \textit{connected(A,B)} \wedge B.c \uparrow 2 \neq \bot \quad (s_5)$$

This allows our model to clear the channel on a potential reader process's side in the event of disconnection. When using connectivity as a condition for triggering reactions or in defining conditional assignment statements, we must rely on the assumption that disconnections between processes are detectable. More sophisticated connectivity functions can be defined which use additional properties of processes, such as access control rights and policies, as parameters to the function.

4 Private Communication in a Mobile Setting

The π-calculus is a process algebra that builds on the CSP [9] and CCS [14] models of synchronous communication to encompass mobility. In the π-calculus,

mobility is represented as the dynamic reconfiguration of the communication structure through the creation, communication, and adoption of new channel names. An interesting aspect of this model is that it becomes possible to protect access to a specific channel by creating new channel names and communicating them to other specific processes.

Typically, in the π-calculus, a protected channel is created for communication between processes using the scoping operator ν to ensure uniqueness of the created name and to restrict the use of the channel name to only the specified processes. For example, the notation $(\nu c)(P|Q)$ means that a new name c is created whose scope is the processes P and Q. Processes output new channel names using send events. A notation similar to that of CSP is used to represent send events in the π-calculus, e.g., $c!x$ where c is the name of the channel used for communication and x is a variable holding the new channel name to be communicated. A process notes that it expects to receive a name on channel c by $c?y$, where y is a variable that will hold the input received along channel c. We capture the key elements of the π-calculus model—the creation, output, and input of channel names—in the coordination schema below. Differences from the original π-calculus are a direct reflection that our schema defines an operational model while the π-calculus is an algebra.

In other synchronous coordination models like CSP, we assumed that channel names are fixed and that there is no protection against unauthorized usage. To capture the unique ability of π-calculus to create new channel names that can be passed 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:

c **named** η **if** $g \equiv$
$\quad c := (\eta, nil)$ **if** $g \wedge c \uparrow 2 = \perp$

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:

$c!x$ **if** $g \equiv$
$\quad c := (c \uparrow 1, (ready, x))$ **if** $g \wedge c \uparrow 2 = (idle, \perp)$
$c?y$ **if** $g \equiv$
$\quad y, c := c \uparrow 2 \uparrow 2, (c \uparrow 1, (ready, \perp))$ **if** $g \wedge\ c \uparrow 2 \uparrow 1 = ready \wedge\ c \uparrow 2 \uparrow 2 \neq \perp$

Finally, input/output commands are matched based on channel names:

$B.c := A.c$ **reacts-to** *connected(A,B)* $\wedge\ B.c \uparrow 1 = A.c \uparrow 1 \wedge\ B.c \uparrow 2 \uparrow 1 = idle$
$\wedge\ A.c \uparrow 2 \uparrow 1 = ready \wedge\ A.c \uparrow 2 \uparrow 2 \neq \bot$

$A.c, B.c := (A.c \uparrow 1, (idle, \bot)), (B.c \uparrow 1, (idle, \bot))$ **reacts-to** *connected(A,B)*
$\wedge\ B.c \uparrow 1 = A.c \uparrow 1 \wedge\ B.c \uparrow 2 = (ready, \bot)$

We address disconnection of a communicating pair of processes by resetting the potential reader's channel:

$B.c := (B.c \uparrow 1, (idle, \bot))$ **reacts-to** $\neg$*connected(A,B)*
$\wedge\ B.c \uparrow 1 = A.c \uparrow 1 \wedge B.c \uparrow 2 \uparrow 2 \neq \bot$

The resulting schema allows us to capture an interesting combination of dynamicity and mobility.

5 Agent Mobility in Wired Networks

Agent systems represent a popular new style of computing specifically designed to take advantage of the global space offered by the Internet. In these systems, an agent is a code fragment that can move from site to site in 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 original coordination modality proposed in Linda [6]. TuCSoN [18], MARS [2], and Limbo [4] are just a small sample of agent systems that employ tuple based coordination.

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

- agent mobility among servers
- mechanisms for controlling the admission of migrating agents
- 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.

The basic priniciples of these models are illustrated in Figure 5, and provide the basis for the schema proposed below.

A coordination schema that enforces the design style discussed above will need to distinguish between agents and servers. Syntactically this can be done

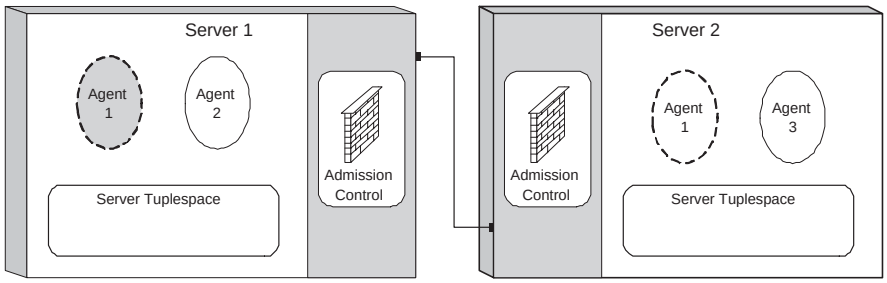

Fig. 5. The basic tuple space coordination model for wired networks. Agents 1 and 2 initially reside on Server 1 and can issue traditional tuple space operations on the server's tuple space. Server 1 is connected to Server 2, on which Agent 3 resides. At some point in time, Agent 1 meets the admission conditions for migration and becomes resident on Server 2.

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:

$Agent\ X \equiv$
$\quad Program\ Agent\_X$

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:

$\lambda \equiv$
$\quad$ *(current_location, desired_server_relocation)*

and to provide the agent with a single move operation:

$goto(\mathcal{S}) \equiv$
$\quad \lambda := (\lambda \uparrow 1, S)$

As before, we use $var \uparrow n$ to denote the n^{th} field of the value stored in variable var. This subjective move operation simply changes the value of the agent's location variable λ to reflect the fact that new server location S is desired. The agent is restricted to using only this operation to foster a change in its location.

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. A variable T could be used to represent such a tuple space, where T is a set of tuples. 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:

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

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

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

where we use the nondeterministic value selection expression $x'.Q$ to identify 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 variable T, intended to hold the contents of the tuple space used by co-located agents, and a location λ. Unlike agents, a server's location cannot change (we address systems that encompass both host and agent mobility in the next section). 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 Q contains a tuple i that identifies that agent but no operations are available to access it. The server need not store its own identity in Q. The server can discover the presence of agents by reading the shared tuple space Q without being able to modify it. Restricting access in such a way can be accomplished by hiding Q inside an operation such as:

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

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

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

which places in the hidden variable M a request to move agent A to server S.

The schema elements presented so far restrict the representation of location in agents and servers to a particular form, prevent altering of the location variable except through the provided subjective and objective move constructs, restrict an agent's method of accessing the tuple space to the defined tuple space operations, and restrict servers to the given construct for agent discovery to pre-

vent tampering of agent identifiers. These syntactic restrictions on agent and server code are complemented by coordination patterns built into the **Interactions** section. In this particular schema, the coordination patterns define the rules by which agents and servers share tuple spaces and how agents' subjective and objective move requests are completed.

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:

$$S.T \approx A.T \textbf{ when } S.\lambda \uparrow 1 = A.\lambda \uparrow 1$$
$$\textbf{engage } S.T$$
$$\textbf{disengage } S.T., \emptyset$$

$$S.Q \approx A.Q \textbf{ when } S.\lambda \uparrow 1 = A.\lambda \uparrow 1$$
$$\textbf{engage } S.Q \cup A.Q$$
$$\textbf{disengage } S.Q - \{A.\iota\}, \{A.\iota\}$$

where we assume that the initial value of $A.Q$ is permanently saved in $A.\iota$, another hidden variable. The first sharing rule simply states that when an agent becomes located at a server, the contents of the server's tuple space are shared with the agent. When the agent moves away from the server, the tuple space is no longer shared; the server retains the contents of the formerly shared tuple space, but the agent's tuple space becomes empty. The second sharing rule similarly defines sharing of the tuple space that holds agent identifiers, in order to support the discovery of agents at a server and admission control functions.

Second, we must specify the rules for handling agent migration requests. 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 (stored in M) by the server S' is transformed into an equivalent hidden subjective request:

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

Notice that neither the subjective move specified above nor the subjective move specified by the *goto* operation actually moves the agent to a new location; rather, they change the agent's location to reflect that a new location is desired. To complete the subjective move, i.e., to change the agent's current location to be the desired location, we must consider two cases. First, when the agent A is accepted by the destination $\mathcal{S}$ and the move is carried out:

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

and second, when the move is rejected and the agent move request is cleared:

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

where $admitted(A.Q, S)$ is a user-defined function that captures the admission control policy for accepting or rejecting migration requests.

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
$\cdots$

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

The **assign** section of the Inspector program has two statements. The first is a transaction that moves the Inspector to the next service site, removes the package counter at the site via an **in** operation, adds the counter to its own, and resets and replaces the counter via an **out** operation. The second reactively resets the Inspector's counter when it reaches its home site.

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** $RQ \neq nil \wedge tt = nil$

where ρ, ω, and π are user-defined functions that specify the criteria under which the **in** operation is extended, and id is simply an auxiliary variable used to

represent the issuing agent's unique identifier. Notice that the above definition uses the fact that reactive statements are interleaved after the execution of an assignment statement (even in transactions) to find a tuple tt matching the requested pattern p, to accomplish the execution of the extended behavior *action*, and to clear the request RQ. 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*.

6 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 essence of the LIME model is illustrated in Figure 6, and the key features of LIME can be described 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).

The above features are captured in the schema proposed below.

In Mobile UNITY, it is convenient to represent each tuple space as a pair of variables, with the first element 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 tuple space sharing rule can be easily expressed as follows:

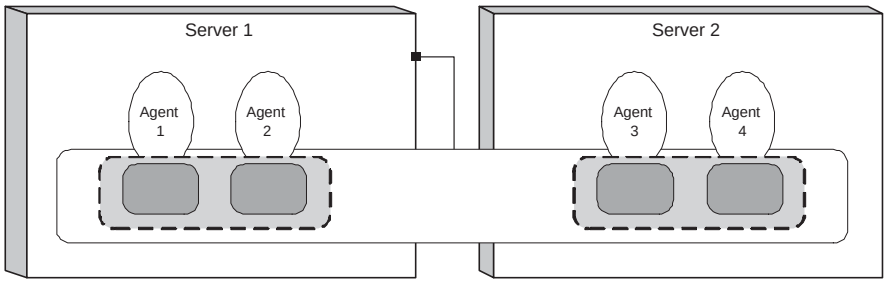

Fig. 6. The essence of LIME coordination model. Hosts, represented as shadowed boxes, serve as containers for agents. Each agent, represented by an oval, is permanently associated with a tuple space, represented as a dark gray box. Agents share tuple spaces when on the same host, creating a logical host-level tuple space (represented by the medium gray box). Agents residing on connected hosts also share tuple spaces, creating a logically federated tuple space represented by the lightest gray box.

$$
\begin{aligned}
A.X[2] \approx B.Y[2] \ &\textbf{when}\ \mathit{connected}(A,B) \wedge A.X[1] = B.Y[1] \\
&\textbf{engage}\ A.X[2] \cup B.Y[2] \\
&\textbf{disengage} \\
&\langle \text{set}\ t, C, Z : \mathit{connected}(A,C) \wedge A.X[1] = C.Z[1] \\
&\quad \wedge\ t \in A.X[2] \wedge\ t\ \mathit{owned\ by}\ C :: t \rangle, \\
&\langle \text{set}\ t, C, Z : \mathit{connected}(B,C) \wedge B.Y[1] = C.Z[1] \\
&\quad \wedge\ t \in B.Y[2] \wedge\ t\ \mathit{owned\ by}\ C :: t \rangle
\end{aligned}
$$

where A and B are agents, *connected* is defined in terms of reachability in the ad hoc network and the array elements $X[1]$ and $X[2]$ 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:

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

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. 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:

$$
\begin{array}{l}
A.X[2] := \\
\quad \langle \text{set } t, B, Y : t \in A.X[2] \wedge \ t.dest = B \wedge \ connected(A, B) \\
\qquad \wedge \ A.X[1] = B.Y[1] :: t[loc : B; dest : B] \rangle \\
\quad \cup \\
\quad \langle \text{set } t, B, Y : t \in A.X[2] \wedge \ t.dest = B \wedge \ \neg(connected(A, B) \\
\qquad \wedge \ A.X[1] = B.Y[1]) :: t \rangle \\
\qquad \textbf{reacts-to } true
\end{array}
$$

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 the features of LIME. The interested reader can find a complete formalization of LIME in terms of Mobile UNITY in [17]. 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.

7 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 [10], are more restrictive in terms of the range of operations provided for mobility while others, such as CodeWeave [12], 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 directly reflect the overall structure of the system
- protection enforcement via capabilities that rely on unique secret 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 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 to communication 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** $OP \uparrow 1 = pass$; $OP := nil \rangle$

where n is the domain name to be operated upon. 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 creation of 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** $g; d := nil;$
$\quad\quad d := OP \uparrow 3$ **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 for encoding any of these operations is to first verify that the referenced name is correct; second, to record this fact in the variable OP; and finally, to complete all necessary changes to the domain structure. We illustrate the use of this pattern 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 7a.

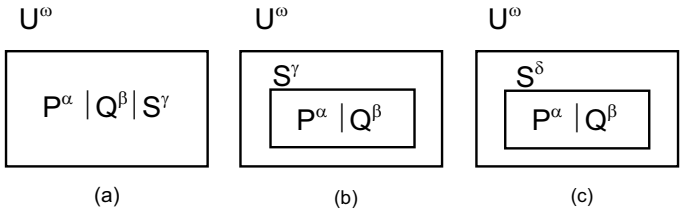

Fig. 7. Domain configurations

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 7b.

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 7c. 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.

8 Formal Verification

In this section we take up the issue of formal verification. Systems specified using Mobile UNITY may be formally checked by employing the Mobile UNITY proof logic. Since all coordination constructs we considered so far are ultimately expressed in terms of basic Mobile UNITY statements, one can simply expand the macros defining the semantic interpretation of each construct and, by also taking into consideration the contents of the **Interactions** section, one can carryout the desired proof from first principles. We show how this can be accomplished in the next subsection, but we do this strictly in order to illustrate the essence of the verification process. We also take advantage of this opportunity to emphasize the critical role the Hoare triple plays in enabling us to reason about individual statements, whether or not they are augmented with reactions.

Translating everything to Mobile UNITY for verification purposes is clearly not a very convenient way to carry out the proofs. For this reason we propose an approach that is more abstract and less cumbersome. The general idea is to provide an abstract semantic definition for each coordination construct. We do so by employing the concept of a *global virtual data structure* (GVDS) [19], an abstract representation of the global state of the coordination process. Local coordination-related actions are given semantic meaning in terms of their logical effect on the GVDS. By employing the notion of GVDS, coordination actions are reduced (logically speaking) to value assignments to the GVDS, which acts as a single global variable. The basic idea is illustrated later in this section for three different coordination models, and can be summarized as entailing the following key steps. Given a specific coordination model whose primitives have been expressed in Mobile UNITY as macros and interactions, a specific GVDS is formulated and all primitive operations are recast as atomic transitions over the GVDS. This abstract version of the operations is shown to be indistinguishable from the concrete realization of the same set of operations, under an appropriate state-to-state mapping. At this point, any proof about the overall system can use the abstract version of the statements without any need to refer to the actions included in the **Interactions** section of the system description, i.e., by consulting only the program text and previously established properties of the coordination constructs.

The notion of GVDS emerged from our own earlier work with LIME and was envisioned as a way to allow software engineers to design systems in terms of local actions and to reason about their effects on a global scale. Up to now, this idea has been leveraged strictly as a design concept. This is the first instance that

demonstrates its potential for simplifying program verification, whether carried formally or informally. Before demonstrating this new application of the GVDS concept on several coordination models, we review some of the basic notions of formal verification in Mobile UNITY. Synchronous message passing is used as an example.

8.1 Verification from First Principles

In this section, we use the Mobile UNITY implementation of the $c!x$ and $c?y$ synchronous communication constructs as a vehicle for explaining how to employ standard Mobile UNITY proof techniques. For the reader's reference, the Mobile UNITY encoding of the CSP-like coordination constructs discussed in this section can be found in Section 3. Before we begin a discussion of how to apply verification techniques to this encoding, we take this opportunity to remind the reader that in this example we continue to assume that the channel is defined strictly between only one writer process (process A) and one reader process (process B).

The desired system behavior between a writing process and a reading process using the Mobile UNITY implementation of the $c!x$ and $c?y$ synchronous communication constructs is summarized by the state transition diagram shown in Figure 8. Reads and writes are issued by the communicating processes on the channel. The execution of each read or write operation results in a state change on the local channel variable. Disconnections and reconnections caused by the relative movement of components also impacts the state of local channel variables. These local state changes and their associated reactions cause the state transitions seen in the diagram. The global configuration is constrained such that only these transitions occur in the execution of the communication protocol.

The correctness conditions captured in Figure 8 can be formally expressed by capturing each depicted transition in terms of properties of the form p **unless** q. For instance, when the system is in a state in which the channel variables of two connected processes both have a value of $(ready, v)$, only two transitions are possible. One takes the system to a state in which both channel variables have a value of $(idle, \perp)$, while in the other, the channel variables have values of $(ready, v)$ and $(idle, \perp)$. To prove an **unless** property such as this, we must consider its validity over every statement in the system. Most statements have no impact, and can be ignored. Only those statements that impact the state of the channel variable or the distinguished location variable λ should be considered. Two statements in our schema that could violate the **unless** property under consideration are those that implement the $c!x$ and $c?y$ constructs, i.e., statements s_1 and s_2 in Section 3, respectively. We must also consider the movement of processes. When we consider each of these statements and the impact of mobility when the system is in the state $((ready, v), (ready, v))$, we can show that all transitions agree with those depicted in Figure 8.

statement s_1: If the statement $s_1 \equiv c!x$ is selected for execution in the sender process A, the statement reduces to a skip and no state change occurs. This is

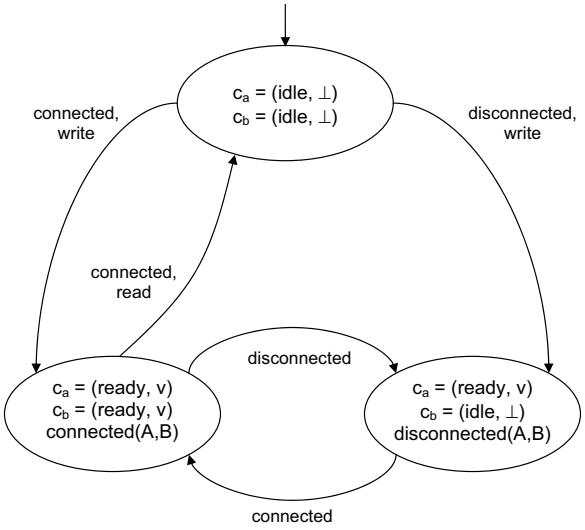

Fig. 8. An overview of the desired system behavior. In the figure, the local channel states of two processes A and B are represented by variables c_a and c_b, respectively.

because the guard of s_1 is not satisifed; the channel variable has a state that is not of $(idle, \perp)$.

statement s_2: If the statement $s_2 \equiv c?y$ is selected for execution in the receiving process B, the guard for statement s_2 is satisfied. The execution of the statement results in a state in which the local channel value of the process issuing the read becomes $(ready, \perp)$. This triggers a reaction (statement s_4) which resets the channel variables to $(idle, \perp)$ for both A and B, and the reactive program reaches fixed point in a single step. The combination of the execution of s_2 and the reactive statement s_4 force an atomic state change in the system. This transition, then, is a valid transition identified by the **unless** property that we are considering.

mobility: At any point in the execution of the protocol, the partipating parties may move. If either of the communication partners move, then the result of *connected* relation between the communication partners is updated to reflect the change in location. If the *connected* relation changes to reflect that the sender becomes disconnected from the receiver while the system is in a state in which both processes' local channel variables have values of $(ready, v)$, then the reaction s_5 fires. The reactive program resets the reader's channel variable to the value $(idle, \perp)$ and reaches fixed point. Again, this is a single atomic state change that

is identified as a valid transition in the **unless** property that we are interested in.

From the above, we can conclude that neither the execution of statements s_1 and s_2 nor the mobility of processes causes a violation of the **unless** property that we are interested in.

To illustrate the process in a more precise and formal manner, let us consider statement s_2 that implements the $c?y$ construct with respect to one of the state transitions described above. Let us consider the starting state in which two processes are connected and have values available on their local channels. The safety property under discussion is captured by:

P **unless** $(Q \vee W)$
where
$P \equiv \mathit{connected}(A, B) \wedge\ c_a = (\mathit{ready}, v) \wedge\ c_b = (\mathit{ready}, v)$
$Q \equiv \mathit{connected}(A, B) \wedge\ c_a = (\mathit{idle}, \perp) \wedge\ c_b = (\mathit{idle}, \perp)$
$W \equiv \neg\mathit{connected}(A, B) \wedge\ c_a = (\mathit{ready}, v) \wedge\ c_b = (\mathit{idle}, \perp)$

With respect to s_2, the proof obligation assumes the form:

$$\{P \wedge\ \neg(Q \vee W)\}\ s_2\ \{P \vee (Q \vee W)\}$$

This reduces to showing that

$\{P \wedge \neg(Q \vee W\}\ s_2\ \{H\}$
$H \mapsto \mathit{fixpoint}(\Re)$
$\{\mathit{fixpoint}(\Re)\} \Rightarrow P \vee (Q \vee W)$

where H is an intermediate state that holds after the execution of the normal statement s_2 but before the reactive program W begins execution. Given a state in which P holds, the execution of statement s_2 takes the system to a state H in which $c_a = (\mathit{ready}, v)$ and $c_b = (\mathit{ready}, \perp)$ when the guard of s_2 is met. A single reactive statement, s_4, is enabled in the reactive program $\Re$. This reactive statement assigns c_a and c_b the value of $(\mathit{idle}, \perp)$. The reaction is disabled and no other reactions are enabled. Thus, fixed point is reached, and Q holds.

If we were to consider a movement that causes disconnection, a single reactive statement, s_5, is enabled in the reactive program $\Re$. This reactive statement assigns to the potential reader, c_b, the value of $(\mathit{idle}, \perp)$. The reaction s_5 disables itself, and the new state does not enable any other reactions. Thus, fixed point is reached, and W holds.

In a similar manner, we can consider each transition with respect to every statement in the program and the mobility of processes. We could then conclude that the **unless** properties under consideration hold throughout the execution of the communication protocol.

8.2 Abstract Treatment of Coordination Constructs

The coordination schemas presented in this paper can be used to simplify programming in Mobile UNITY by allowing programmers to rely on familiar coordination constructs. Given a schema that captures the essence of a coordination model, one can simply use a high-level coordination construct in a Mobile UNITY program without having to consider how the schema is implemented. As seen in the previous section, verifying that the program is correct, however, requires one to delve into the details of the schema implementation. Requiring a style of verification that utilizes low level mechanics of a construct's implementation seems to defy our philosophy of providing minimal and elegant coordination mechanisms in Mobile UNITY. As noted at the start of Section 8.2, we propose to exploit the notion of global virtual data structures (GVDS) [19]. A GVDS is a reflection of the global system state captured in a standard data structure that is accessed locally through the use of a familiar API. The term virtual is used to describe the representation because the entirety of the structure is not simultaneously available; local access via the data structure's API is constrained according to properties of the environment. The key contribution of this approach is that in creating the appropriate abstraction, an operation that requires interaction among multiple components is reduced to an atomic change of the global representation of system state, the GVDS.

In this section, we illustrate this style of verification for three of the coordination models considered in this paper. We begin by showing how reasoning about the synchronous communication implementation can be simplified by using an abstract shared channel. We move on to show how a similar approach can be employed in LIME by relying on an abstract shared tuple space. Finally, we show how spatial relations can ease verification in settings such as the Mobile Ambients model.

An Abstract Shared Channel in CSP. We have illustrated the mechanics of how to prove the correctness of the Mobile UNITY implementations of the $c!x$ and $c?y$ constructs. In the style of verification outlined in the previous subsection, it was necessary to utilize details of the implementation in order to prove system properties. In this section, we show how we can reduce the expression of synchronous communication between two programs with distinct variables representing a channel to a single abstract shared variable representing a channel. By doing so, we provide a programmer with the ability to prove properties about a program utilizing the Mobile UNITY $c!x$ and $c?y$ constructs by reasoning about a high-level abstraction, a shared channel, without needing to know the lower-level mechanics present in the **Interactions** section of the actual implementation.

We can represent synchronous communication through the use of an abstract shared variable $\mathcal{C}$ representing a shared channel. The channel $\mathcal{C}$ can be in one of three states: $\perp$, v, or $\hat{v}$. The state $\perp$ indicates that the channel is idle, v indicates that a value is available on the channel, and $\hat{v}$ indicates that a process wanting to send the value v is waiting for connectivity to the reading process. Table 1

shows how the states of the local program variables c_a and c_b map to the abstract variable $\mathcal{C}$'s state. Notice that the local channel states are not mapped to the abstract shared channel state for the last three rows. This is because the system state is a *shadow state*, i.e., the state exists locally but is invisible globally since a reaction is immediately enabled which changes the channel state in a single atomic step.

Process A's state	Process B's state	Connectivity	Abstract Shared Channel State
$(idle, \perp)$	$(idle, \perp)$	$connected(A, B)$	$\perp$
$(idle, \perp)$	$(idle, \perp)$	$\neg connected(A, B)$	$\perp$
$(ready, v)$	$(ready, v)$	$connected(A, B)$	v
$(ready, v)$	$(idle, \perp)$	$\neg connected(A, B)$	$\hat{v}$
$(ready, v)$	$(idle, \perp)$	$conected(A, B)$	invisible
$(ready, v)$	$(ready, \perp)$	$connected(A, B)$	invisible
$(ready, v)$	$(ready, \perp)$	$\neg connected(A, B)$	invisible

Table 1. Local channel states and corresponding global states

Given the correspondence between the states of the local channel variables and the abstract shared variable $\mathcal{C}$, we can give the following definitions of the shared channel:

$\mathcal{C} = \perp$ **iff** $c_a = c_b = (idle, \perp)$
$\mathcal{C} = v$ **iff** $c_a = c_b = (ready, v)$
$\mathcal{C} = \hat{v}$ **iff** $c_a = (ready, v) \wedge\ c_b = (idle, \perp)$

We can now define the $c!x$ and $c?y$ constructs in terms of their effects on the abstract shared channel:

$c!x$ **if** $g \equiv$
    $\mathcal{C} := x$ **if** $g \wedge \mathcal{C} = \perp \wedge$ *connected(A,B)*
        $\parallel\ \mathcal{C} := \hat{x}$ **if** $g \wedge \mathcal{C} = \perp \wedge \neg$ *connected(A,B)*
$c?y$ **if** $g \equiv$
    $y, \mathcal{C} := \mathcal{C}, \perp$ **if** $g \wedge\ \mathcal{C} \neq \perp \wedge$ *connected(A,B)*

Typically, proving safety and liveness properties about a protocol whose implementation spans across programs requires a programmer to rely on the details of the **Interactions** section. However, the concept of an abstract shared channel in this case provides a global perspective on the communication protocol using local state information, which allows the programmer to transparently reason about the protocol without knowing the Mobile UNITY implementation details. The need to consult the **Interactions** section for verification purposes is eliminated, which can greatly reduce the complexity of the proof process. Given the concept of the abstract shared channel as described, a programmer can prove

properties about the program simply by using the standard assignment axiom in Hoare proof logic. For example, proving properties about the construct $c!x$ reduces to:

$$\{P\}\ \mathcal{C} := x \textbf{ if } g \wedge\ \mathcal{C} = \bot \wedge \textit{ connected(A,B)}$$
$$\|\ \mathcal{C} := \hat{x} \textbf{ if } g \wedge\ \mathcal{C} = \bot \wedge\ \neg \textit{ connected(A,B)}\ \{Q\}$$

where the proof obligation is satisfied simply by showing

$$P \Rightarrow (\textit{connected(A,B)} \wedge g \wedge\ \mathcal{C} = \bot \Rightarrow Q[x/\mathcal{C}])$$
$$\wedge\ (\neg\ \textit{connected(A,B)} \wedge\ g \wedge\ \mathcal{C} = \bot \Rightarrow Q[\hat{x}/\mathcal{C}])$$

It should be noted that the channel state is affected not only by the communication constructs but also by mobility and its impact on connectivity. As such, the abstract semantics of a value assignment to λ must also include changes to the channel state, e.g., v to $\hat{v}$.

An Abstract Shared Tuple Space in LIME. The LIME system makes coordination possible in ad hoc networks by distributing the tuple space over multiple agents. Agents coordinate via tuple space operations on the logically merged tuple space formed by the individual tuple spaces of connected agents. Central to the Mobile UNITY implementation of the LIME approach in Section 6 is a local tuple space variable defined using the transitive and transient sharing operation, $\approx$, subject to agent connectivity constraints. Given this encoding, the Mobile UNITY implementations of the **out**, **in**, and **rd** constructs and their extensions are elegantly captured as simple assignment statements on the local tuple space variable. This style of encoding, however, requires one to examine the tuple space variables of several programs in order to verify the implementation of the construct. To aid the programmer using the Mobile UNITY schema in the verification process, we turn our attention to providing a more abstract representation of the tuple space sharing mechanism.

In this case, the GVDS has a structure that is controlled by the physical connectivity of hosts. We represent the set of all tuples in the universe as a single virtual global tuple space, $\mathcal{T}$, which is accessible by all agents in the system. Since tuples are shared only between connected agents, we represent a snapshot of agent connectivity at the current moment in time using a connectivity relation, *connected*, defined over all agents in the system. Specifically, this connectivity relation is defined in terms of the transitive closure of physical network communication links that exist at the current instant in time. The combination of the set $\mathcal{T}$ and the connectivity relation form our GVDS. Given this representation, it is possible to extract a snapshot of a particular agent's logically shared tuple space. Moreover, we can interpret the Mobile UNITY implementations of LIME tuple space coordination constructs as operations that access and modify the shared tuple space $\mathcal{T}$.

In Section 6, the Mobile UNITY implementations of tuple space operations were captured as assignments to the local tuple space variable T which was

transitively and transiently shared across connected agent programs. Given the abstract treatment described in the previous paragraph, we can redefine the traditional content-based based retrieval operations performed on the tuple space in terms of $\mathcal{T}$:

$$z = \mathbf{in}(p) \textbf{ if } g \equiv$$
$$\langle\ \theta : \theta = \theta'.(\theta' \in \mathcal{T} \wedge match(\theta', p) \wedge\ connected(A, \theta'.loc) \wedge g ::$$
$$z := \theta \parallel \mathcal{T} := \mathcal{T} - \{\theta\}\rangle$$

where A is the requesting agent. Notice that the assignment to the global tuple space $\mathcal{T}$ is dependent on the connectivity relation with respect to the tuple θ's location, i.e., the agent owning the tuple. It is possible to model operations this way because, as discussed in Section 6, tuples have a hidden "owner" field that holds the tuple owner's location. The **rd** construct can be redefined in an similar fashion to take advantage of the use of an abstract shared tuple space, except that the returned tuple is not removed from the variable $\mathcal{T}$. Tuple generation is likewise redefined with respect to $\mathcal{T}$, but does not require constraints using the connectivity relation. Notice that these statements simply assign a value to the universal tuple space abstraction $\mathcal{T}$, much like the Mobile UNITY definition that updates the local tuple space variable T. The difference lies in the representation: the abstract treatment of the LIME tuple space coordination mechanism allows us to reason about the effects of assignment on a global virtual data structure using the standard assignment axiom, rather than examining the impact of the assignment across the variables of multiple programs.

As a final note on tuple space coordination constructs, defining the location-aware tuple generation operation **out** in terms of the tuple space abstraction $\mathcal{T}$ requires additional consideration to capture the correct "delivery" of tuples to particular agent or host in our abstract treatment of tuple space coordination. As discussed in Section 6, the location-aware **out** implementation places a tuple in the creator's local tuple space, and the intent to migrate the tuple is indicated by setting the destination field of the tuple to the desired location. If the creating agent is connected to the destination, the location fields of the tuple are changed to reflect migration of the tuple. In cases where the generated tuple is intended for an agent that is not currently connected to the creator of the tuple, the tuple is stored locally until a connection to the intended recipient becomes available. We can simply treat the migration of tuples as an extension to the movement of agents in order to capture the definition of the globabl virtual data structure. With respect to our abstract treatment of tuple space coordination, as an agent moves, the value of the *connected* relation changes in response. The location fields of the tuple of interest are changed to reflect the new state of connectivity. Given this representation, proofs of the encodings of these constructs must consider location changes as well.

The result of this section is a reduction of operations in the coordination process to standard assignment statements on the global virtual data structure. In this case, tuple space operations can be represented as assignments on a data structure that encapsulates a set $\mathcal{T}$ of tuples and an associated connectivity

relation. Verification, then, is achieved through straightforward application of the assignment axiom.

Spatial Reasoning in Mobile Ambients. Actions available to an ambient to induce mobility among domains, such as **exit**, **enter**, and **open**, are realized in a distributed fashion through the interaction of multiple cooperating ambients, and the result is a restructuring of the domain organization. The implementation of the Mobile Ambient constructs in the Mobile UNITY schema presented in Section 7 reflects the distributed nature of ambient interactions. Typically, the effect of a Mobile UNITY operation implemented in a distributed fashion is studied by analyzing the details of ambient interactions. Here again, the key to simplifying verfication of such programs is to abstract away the details of the system. Though it is not immediately apparent that we can apply the same approach set forth in the previous section for achieving abstract reasoning, our encoding of Mobile Ambient constructs in Mobile UNITY allows us to employ a similar modeling and verification technique.

In section 7, we captured the actions used to manipulate an ambient's location in the domain hierarchy as a Mobile UNITY schema. In general, these actions were expressed using local Mobile UNITY assignment statements and global reactions. For example, the **exit** n operation issued by a Mobile UNITY ambient program was expressed in the program text as a series of sequential assignment statements:

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

Reactive statements in the **Interactions** section are interleaved with the execution of the individual normal statements in the transaction above. The following reactions act to service the request:

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

An **exit** request is submitted locally by an ambient via the first normal statement in the transaction. As a result, the first reactive statement is triggered. This statement executes to indicate that the operation was a success and disables itself, reaching fixed point. After execution of the second normal statement in the locally executed transaction above, the second reactive statement is enabled. This statement changes the parent references of the ambient and disables itself. No other reactions are enabled, and fixed point is reached. Finally, the third normal statement in the locally executed transaction is executed. This statement clears the **exit** request from the system. The net effect of executing the Mobile UNITY implementation of the **exit** operation is a single atomic state change that manipulates the domain structure formed by the collection of ambients in the system. This feature of the **exit** encoding in Mobile UNITY allows us to

deal with ambient actions in a more abstract fashion by interpreting them as atomic operations that operate over a global representation. Therefore, we can represent the domain organization formed by a collection of ambients in Mobile UNITY as a global virtual data structure; given that the domains are organized hierarchically, the natural choice is to represent the collection of ambients as a tree $\mathcal{A}$.

Given the tree representation of the domain structure, the **exit** n action can be viewed as an atomic operation that removes a subtree in $\mathcal{A}$ rooted at node n with parent p and inserts the subtree in $\mathcal{A}$ as a child of p's parent. The result is a tree in which the ambient n is now a sibling of its former parent. This atomic operation on the global virtual tree can be reduced to a simple Mobile UNITY value assignment. Given that we represent the tree $\mathcal{A}$ as a set of *(node, set of children)* pairs, the **exit** n action executed by an ambient m reduces to the following assignment:

$$\begin{aligned}&\langle p, Y, Z\colon (n, Z) \in \mathcal{A} \wedge\ m \in Z \wedge (p, Y) \in \mathcal{A} \wedge\ n \in Y\\ &\quad :: \mathcal{A} := \mathcal{A} - \{(n, Z)\} + \{(n, Z - \{m\})\} - \{(p, Y)\} + \{(p, Y + \{m\})\}\rangle\end{aligned}$$

The implementations of the **enter** and **open** constructs are achieved in a fashion similar to that of the **exit** operation discussed above: a local ambient assignment statement is used to issue and clear requests, and a group of globally applicable reactive statements carry out the resulting domain restructuring. For either action, the end result of execution is a single large-grained state change for the system. As such, we can also capture their effects as simple assignment statements on the global representation of the domain structure, the tree $\mathcal{A}$. Using such an abstraction allows the programmer to reason about the global effect of distributed, interacting ambient programs implemented in Mobile UNITY using standard proof techniques for value assignment on the tree $\mathcal{A}$.

Case after case, a simplification of the verification process is achieved by considering abstract representations over a global structure. The question is, why not rely solely on these abstractions? The answer rests with the distinction between examining the coordination process versus reasoning about the overall result. We are interested in presenting a method of formalizing high level coordination constructs that appeals to both formalists and software engineers. As such, we must precisely define the constructs with respect to the actions each performs in the coordination process, while providing an acessible method of verification that focuses on the results of the executed coordination constructs.

9 Conclusions

The theme of this paper is the formalization of coordination models, particularly in settings that entail mobility. The essential traits of a variety of coordination styles recognized in the literature have been captured using Mobile UNITY. In all cases, the formalization has been partitioned between a set of local actions and a set of global interactions that abstract the coordination and communication activities associated with each specific model. It is our hope that these

exercises will be used by others to specify the coordination semantics associated with various coordination languages and constructs. Such exercises are particularly important in situations where mobility is present. Precise specifications assume critical importance if dependability requirements are to be met. The simplicity of the formalizations discussed in the paper suggests that Mobile UNITY is an appropriate vehicle for exploring the semantics of coordination. The similarity among the resulting formal treatments reflects the decoupled style of computing being promoted by coordination models in general and facilitates direct comparisons among competing techniques and constructs. Extensions of the original models (e.g., the inclusion of location sensitive interactions) highlight the opportunities that exist to formally explore new modes of coordination before incorporation into a coordination infrastructure or middleware. Finally, a style of verification based on the global virtual data structures concept is offered which provides a simplified and pragmatic approach to formal analysis of programs utilizing Mobile UNITY encodings of coordination constructs by reducing the task to reasoning about atomic changes to a global representation of system state.

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. 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.
5. C. Fournet and G. Gonthier. The reflexive CHAM and the join-calculus. In *Proceedings of the 23rd ACM Symposium on Principles of Programming Languages*, pages 372–385, 1996.
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. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

9. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.
10. C. Mascolo. MobiS: A specification language for mobile systems. In *Proceedings of 3rd International Conference on Coordination Models and Languages*, volume 1594, pages 37–52. Springer-Verlag, 1999.
11. C. Mascolo, G.P. Picco, and G. Roman. CodeWeave: Exploring fine-grained mobility of code. *Automated Software Engineering Journal (to appear)*.
12. C. Mascolo, G.P. Picco, and G.-C. Roman. A fine-grained model for code mobility. In *Proceedings of the Seventh European Software Engineering Conference (ESEC)*, volume 1687 of *Lecture Notes in Computer Science*, pages 39–56. Springer-Verlag, September 1999.
13. P.J. McCann and G.-C. Roman. Compositional programming abstractions for mobile computing. *IEEE Transactions on Software Engineering*, 24(2):97–110, 1998.
14. R. Milner. *Communication and Concurrency*. Prentice Hall, 1980.
15. R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, parts I and II. *Information and Computation*, 100(1):1–77, 1992.
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.L. Murphy, G.P. Picco, and G.-C. Roman. LIME: A coordination middleware supporting mobility of hosts and agents. Technical Report WUCSE-03-21, Washington University, Department of Computer Science and Engineering, St. Louis, Missouri, 2003.
18. A. Omicini and F. Zambonelli. The TuCSoN coordination model for mobile information agents. In *Proceedings of the First Workshop on Innovative Internet Information Systems*, June 1998.
19. G. P. Picco, A. L. Murphy, and G.-C. Roman. *Process Coordination and Ubiquitous Computing*, chapter Global Virtual Data Structures, pages 11–29. CRC Press, 2002.
20. G.-C. Roman and P. J. McCann. A notation and logic for mobile computing. *Formal Methods in System Design*, 20:47–68, 2002.
21. P. Sewell, P. Wojciechowski, and B. Pierce. Location-independent communication for mobile agents: a two-level architecture. In H. Bal, B. Belkhouche, and L. Cardelli, editors, *Internet Programming Languages*, volume 1686 of *Lecture Notes in Computer Science*. 1999.