Report Number: WUCS-97-44

1997-01-01

An Introduction to Mobile UNITY

Gruia-Catalin Roman and Peter J. McCann

Traditionally, a distributed system has been viewed as a collection of fixed computational elements connected by a static network. Prompted by recent advances in wireless communications rechnology, the emerging field of mobile computing is challenging these assumptions by providing mobile hosts with connectivity that may change over time, raising the possibility that hosts may be called upon to operate while only weakly connected to or while completely disconnected from other hosts. We define a concurrent mobile system as one where independently executing coponents may migrate through some space during the course of the computation, and where the pattern of... **Read complete abstract on page 2.**

An Introduction to Mobile UNITY

Gruia-Catalin Roman and Peter J. McCann

Complete Abstract:

Traditionally, a distributed system has been viewed as a collection of fixed computational elements connected by a static network. Prompted by recent advances in wireless communications rechnology, the emerging field of mobile computing is challenging these assumptions by providing mobile hosts with connectivity that may change over time, raising the possibility that hosts may be called upon to operate while only weakly connected to or while completely disconnected from other hosts. We define a concurrent mobile system as one where independently executing coponents may migrate through some space during the course of the computation, and where the pattern of connectivity among the components changes as they move in and out of proximity. Note that this definition is general enough to encompass a system of mobile hosts moving in physical space as well as a system of migrating software agents implemented on a set of possibly non-mobile hosts. In this paper, we present Mobile UNITY, which is a notation for expressing such systems and a logic for reasoning about their temporal properties. Based on the UNITY language of Chandy and Misra, our goal is to find a minimalist model of mobile computation that will allow us to express mobile components in a modular fashion and to reason formally about the possible behaviors of a system composed from mobile components. A simplified serial communication protocol among components which can move in space serves as an illustration for the notation.

School of Engineering & Applied Science

An Introduction to Mobile UNITY

Gruia-Catalin Roman
Peter J. McCann

WUCS-97-44

9 October 1997

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

An Introduction to Mobile UNITY

Gruia-Catalin Roman[1] and Peter J. McCann[2]

[1]Department of Computer Science
Washington University
One Brookings Drive
St. Louis, MO 63130 U.S.A.
roman@cs.wustl.edu

[2]Bell Laboratories
Lucent Technologies
1000 E. Warrenville Road
Naperville, IL 60566 U.S.A.
mccap@research.bell-labs.com

Abstract. Traditionally, a distributed system has been viewed as a collection of fixed computational elements connected by a static network. Prompted by recent advances in wireless communications technology, the emerging field of mobile computing is challenging these assumptions by providing mobile hosts with connectivity that may change over time, raising the possibility that hosts may be called upon to operate while only weakly connected to or while completely disconnected from other hosts. We define a *concurrent mobile system* as one where independently executing components may migrate through some space during the course of the computation, and where the pattern of connectivity among the components changes as they move in and out of proximity. Note that this definition is general enough to encompass a system of mobile hosts moving in physical space as well as a system of migrating software agents implemented on a set of possibly non-mobile hosts. In this paper, we present Mobile UNITY, which is a notation for expressing such systems and a logic for reasoning about their temporal properties. Based on the UNITY language of Chandy and Misra, our goal is to find a minimalist model of mobile computation that will allow us to express mobile components in a modular fashion and to reason formally about the possible behaviors of a system composed from mobile components. A simplified serial communication protocol among components which can move in space serves as an illustration for the notation.

Keywords: formal methods, mobile computing, UNITY, Mobile UNITY, shared variables, synchronization, transient interactions

Acknowledgment: This paper is based upon work supported in part by the National Science Foundation under Grants No. CCR-9217751 and CCR-9624815. 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 National Science Foundation.

1. Introduction

The emergence of mobile communications technology is bringing a new perspective to the study of distributed systems. Viewed simply, this technology bestows network connectivity on computers that are mobile, allowing these hosts to be treated as nodes in a traditional distributed computation. However, this view ignores many important issues surrounding mobile computing. These issues stem from both the characteristics of the wireless connection and the nature of applications and services that will be demanded by users of the new technologies.

The low bandwidth, frequent disconnection, and high latency of a wireless connection lead to a *decoupled* style of system architecture. Disconnections may be unavoidable as when a host moves to a new location, or they may be intentional as when a laptop is powered off to conserve battery life. Also, wireless technologies will always lag behind wired ones in terms of bandwidth due to the added technical difficulties [5, 8]. Systems designed to work in this environment must be decoupled and opportunistic. By "decoupled," we mean that applications must be able to run while disconnected from or weakly connected to servers. "Opportunistic" means that interaction can be accomplished only when connectivity is available. These aspects are already apparent in working systems such as Coda [16], a filesystem supporting disconnected operation, and Bayou [18], a replicated database where updates are propagated by pairwise interaction among servers, without involving any global synchronization. Both systems relax the degree of consistency offered to the application programmer in favor of higher availability. In the case of Coda, this tradeoff is justified due to the low degree of write-sharing in the typical filesystem environment. In the case of Bayou, update conflicts are handled with application-specific detection and resolution procedures. Neither system takes the traditional view that distribution should be hidden from the application programmer; both yield to the reality of frequent disconnection and deal with the consequences of update conflicts.

In addition to being weakly connected, mobile computers change location frequently, which leads to demand for *context dependent services*. A simple example is the location dependent World Wide Web browser of Voelker et al [19]. This system allows the user to specify location-dependent queries for information about the current surroundings and the services available. A more general point of view is evidenced in [17], which notes that application behavior might depend on the totality of the current context, including the current location and the nearness of other components, like the identity of the nearest printer or the group of individuals present in a room. The dynamic nature of interaction among components brings with it unprecedented challenges analogous to those of open software systems. Components must function correctly in any of the myriad configurations that might occur. They must also continue to function as components are reconfigured. It is important that we begin to investigate methods for specifying and reasoning about such behavior.

While some systems will be mobile-aware and require explicit reasoning about location and context, the vast majority of existing distributed applications make use of *location transparent* abstractions. Not every distributed algorithm should be re-written from scratch for the mobile setting, and support for location transparent messaging services is desirable. Mobile IP [13] attempts to provide this in the context of the Internet. Even if the goal is transparent mobility, the designers of such a protocol must face the issues brought on by mobility. Explicit reasoning about location and location changes are required to argue that a given protocol properly implements location transparency.

It is important to note that mobile communications technology is not essential for these issues to be made manifest; they were already present in the wide area networks of today. In the current Internet, links to distant nodes are typically of low-bandwidth and are not very reliable. Tightly coupled algorithms do not perform well in this kind of environment, and as in the mobile setting it is appropriate in some cases to sacrifice consistency for better availability. Reconfigurable systems are closely related to notions of executable content and mobile agents, which are motivated by reasons other than host mobility [6]. A mobile software agent might have explicit knowledge and control over its location (which may be specified as a host address), and must interact with components with which it is co-located to achieve some goal. Open software systems that must interoperate under unanticipated circumstances are another example of situations where a broad range of configurations must be considered during system design and implementation in order to guarantee correct behavior [12].

The kinds of weakly connected, context-dependent systems inspired by mobile computing will require new ways of thinking about distributed system design. Important to this task are models and techniques for specification and verification. This paper proposes a new notation and underlying formal model supporting specification of and reasoning about decoupled, location-aware systems. The approach is based on the UNITY [3] model of concurrent computation. This work extends the UNITY notation with constructs for expressing both component location and transient interaction among components. In Section 2, we review UNITY and provide the motivation for our later extensions. Section 3 is a succinct introduction to our new notation, called Mobile UNITY. A formal axiomatic definition of each construct is included. This section treats Mobile UNITY as a mere technical modification to UNITY independent of any notions of mobility. In Section 4, we discuss the mobility and modularity aspects of the Mobile UNITY notation and show how the composition of mobile units reduces to a form of program union. The new notation is illustrated via a simple example, a serial communication protocol which assumes unidirectional transmission from senders to receivers. The former group is assumed to be stationary while members of the latter are allowed to move around. Section 5 discusses the potential use of the new constructs for the semantic definition of novel high-level abstractions of communication and coordination in mobile systems. Conclusions are presented in Section 6.

2. UNITY Review and Critique

Chandy and Misra put forth the UNITY model [3] as a vehicle for the study of distributed computing. A minimal set of concepts, a simple notation and a restricted form of temporal logic were evaluated against a broad range of traditional distributed computations and software development activities including specification, design, coding, and verification. UNITY's success as a research tool rests with its ability to focus attention on the essence of the problem being studied rather than notational artifacts. This is a direct result of its minimalist philosophy which we are about to put to the test in a challenging new arena, mobile computing. In this section we provide a very brief overview of the UNITY notation and proof logic and discuss its strengths and weaknesses with respect to specifying and reasoning about mobile computations.

The key elements of the UNITY model are the concepts of variable and assignment, actually the conditional multiple assignment statement. Programs are simply sets of assignment statements which execute atomically and are selected for execution in a weakly fair manner—in an infinite computation each statement is scheduled for execution infinitely often. An example program called *sender* is shown below. It starts off by introducing the variables used by the program in the **declare** section. Abstract variable types such as sets and sequences can be used freely. The **initially** section defines the allowed initial conditions for the program. If a variable is

not referenced in this section, its initial value is constrained only by its type. The heart of any UNITY program is the **assign** section consisting of a set of assignment statements. The program below has two assignment statements. Each is given a label for ease of reference.

```
program sender
  declare
    bit : boolean
  [] word : array[0..N-1] of boolean
  [] csend, crecv : integer
  initially
    bit = 0
  [] csend = N
  assign
    transmit :: bit, csend := word[csend], csend+1  if csend < N ∧ csend = crecv
  [] new      :: word, csend := NewWord(), 0        if csend ≥ N
end
```

The program *sender* is a model of the sender side of an asynchronous serial communications link. It declares four variables. The first, *bit*, is the shared medium used to transmit, one bit at a time, the value in *word*. The variables *csend* and *crecv* are counters used to keep track of the progress of the sender and receiver, respectively. The statement *transmit* copies the next bit of *word* to *bit* and increments *csend*, if the sender and receiver counters have the same value. When the *csend* counter reaches *N*, the statement *new* is enabled which writes a new value to *word* and resets *csend*. Both *transmit* and *new* are assumed to be atomic operations. In the above program, the guards are mutually exclusive and only one statement can be effectively executed at any point, although this is not required by the model; in general, more than one statement may be effectively enabled. Concurrency is modeled by interleaved execution of these atomic operations. At each computation step one statement is selected for execution and the program state is atomically modified according to that statement. Fairness assumptions require that no statement be excluded from selection forever.

The very simple notation illustrated by the above example has been used successfully to construct abstract operational specifications of some of the best known problems in distributed computing. More importantly, Chandy and Misra have been able to show that an equally parsimonious proof logic can be employed in the formal derivation (through specification refinement) and verification of such programs. In the UNITY proof logic, program properties are expressed using a small set of predicate relations whose validity can be derived directly from the program or from other properties through the application of inference rules. These predicate relations are expressions of allowed sequences of system states, and can be thought of as specifications for correct behavior. A proof of correctness is a demonstration that the text of a program meets a certain specification, i.e., the sequence of states encountered in any possible execution is one of those allowed by the specification. We distinguish two basic kinds of system properties, *safety* and *liveness* properties. Intuitively, a safety property states that some undesirable circumstance does not occur. A liveness property requires that some desirable circumstance eventually does occur. A pure safety property is satisfied by a behavior if and only if it is satisfied by every finite prefix of that behavior. A pure liveness property is one that can always be satisfied by some infinite extension of any finite execution. Any property (set of allowed behaviors) can be expressed as the intersection of a pure safety and a pure liveness property [2].

Standard UNITY [3] provides proof rules for very basic safety and liveness properties that make direct use of the program text. We choose to express basic safety using the **constrains** relation of [11], abbreviated as "**co**." This is a predicate relation developed in the context of generic action systems and is not specific to UNITY, but has a particularly simple form. For two state predicates p and q the expression p **co** q means that for any state satisfying p, the next state in the execution sequence must satisfy q. If this expression is part of a correctness specification, it rules out all those behaviors for which a state satisfying p is followed by a state that does not satisfy q. By using this relation one can state formally that the value of *csend* does not decrease unless it becomes zero, no matter which statement is executed:

$$\text{csend} = \text{k} \;\; \textbf{co} \;\; \text{csend} \geq \text{k} \;\vee\; \text{csend} = 0$$

By convention, all free variables are assumed to be universally quantified, e.g., the above property holds regardless of the current value k assumed by *csend*. To prove that the program *sender* meets the above specification, we need to show that if any statement is selected for execution in a state satisfying $csend = k$, it terminates in a state satisfying $csend \geq k \;\vee\; csend = 0$, for all values of k. We can use well known techniques from sequential programming [4] to carry out this proof for each statement. Formally, **co** can be defined as

$$p \;\textbf{co}\; q \;\equiv\; \langle \forall\, \text{s} :: \{\text{p}\}\ \text{s}\ \{\text{q}\} \rangle$$

using *Hoare triple* [7] notation where s is any statement from the program, p is a precondition, and q is a postcondition. Properties expressed with **co** should be *stuttering invariant*, that is, inserting repeated elements into an execution sequence should not change the value of a **co** relation applied to that execution. This is equivalent to assuming that every program includes a do-nothing *skip* statement or requiring that $p \Rightarrow q$.

More complex safety properties can be defined in terms of the **co** relation. For instance, verification of a program invariant such as

$$\textbf{invariant}\ 0 \leq \text{csend} \leq \text{N}$$

requires one to show that *csend* is initially in the range *0* to *N* and remains so throughout the execution of the program. The former proof obligation is verified by using the information in the **initially** section. The latter proof obligation is a **co** property which has to be checked against each statement of the program.

Progress or liveness properties can also be proven from the text of a program. These properties use UNITY's built-in fairness assumptions to guarantee that a certain predicate is eventually established. Progress is expressed in standard UNITY using the **ensures** relation. The relation p **ensures** q means that for any state satisfying p and not q, the next state must satisfy p or q. In addition, there is some statement s that guarantees the establishment of q if executed in a state satisfying p and not q. Because fairness guarantees that this statement will eventually be selected for execution, the **ensures** relation rules out execution sequences containing states satisfying p unless the last state in any maximal subsequence of p states itself satisfies q or is immediately followed by a state satisfying 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 a **co** property, and the existence of an establishing statement can be proven with standard techniques:

$$p \textbf{ ensures } q \equiv (p \wedge \neg q \textbf{ co } p \vee q) \wedge \langle \exists s :: \{p \wedge \neg q\}\ s\ \{q\} \rangle$$

We take **ensures** as a fundamental element of progress specifications, rather than the newer pure liveness *p* **transient** operator [10] due to **ensure**'s linguistic clarity and our familiarity with it.

A progress property that the *sender* program should satisfy is that the counter *csend* eventually should increase or be reset to zero. This can be expressed as

$$\text{csend} = k \textbf{ ensures } \text{csend} > k \vee \text{csend} = 0$$

This relation states that if the variable *csend* has value *k*, it retains this value until it is set to a greater one or to zero, and that some statement will eventually perform this task. Another desirable progress property is that if *csend* equals zero, it should eventually be set to 1.

$$\text{csend} = 0 \textbf{ ensures } \text{csend} = 1$$

It is straightforward to prove the safety part of each of these **ensures** relations. However, we run into a problem when we try to prove that some statement will eventually establish the right hand side of each of these **ensures**. The *transmit* statement that increments *csend* is only enabled when *csend* = *crecv*. Because no statement in *sender* changes *crecv*, we can easily prove

$$\textbf{stable } \text{crecv} = k$$

which is a formal expression of the fact that the *crecv* variable retains its initial value. Thus no statement can increase *csend* when *csend* ≠ *crecv*. If we had initially constrained the value of *crecv* to be zero in the **initially** section, then we could prove the latter **ensures**, but not the former. However, no such assumption appears in the program text, and *crecv* is initially constrained only to be an integer.

The problem with these proofs arises because the *sender* program expects to be composed with the *receiver* program, shown below. The receiver declares three variables, *bit*, *csend*, and *crecv*, that also appeared in *sender*, and one new variable, *buffer*. The receiver action *receive* copies the variable *bit* into the array *buffer* and increments *crecv*. The *reset* action resets the counter *crecv* to -1.

```
program receiver
  declare
    bit : boolean
  [] buffer : array[0..N-1] of boolean
  [] csend, crecv : integer
  initially
    bit = 0
  [] crecv = N
  assign
  [] receive  :: buffer[crecv+1], crecv := bit, crecv+1 if crecv < N ∧ crecv ≠ csend
  [] reset    :: crecv := -1                            if crecv ≥ N ∧ csend = 0
end
```

We use the UNITY *union* operator, [] to construct a new system, denoted by *sender* []*receiver*. Operationally, the new system consists of the union of all the program variables, i.e., variables with the same name refer to the same memory, the union of all the assignment statements, which are executed in a fair atomic interleaving, and the intersection of the initial conditions. Note that program *receiver* constrains the initial value of *crecv*, but not that of *csend*. If *csend* is initialized

according to the *sender* program, then neither *receive* nor *reset* is initially enabled. The only action that can execute is *new*. This enables *reset*, which in turn enables *transmit*, which in turn enables *receive*. From that point *transmit* and *receive* execute alternately until the entire word has been transmitted. Then another cycle with a different word can begin.

Neither of the above programs is able to make progress without the presence of the other. This is a very tightly coupled system where the two counter values are used to implement a turn-taking scheme for the *transmit* and *receive* statements. In an actual serial communication channel, this turn taking would be the result of the real-time behavior of the two components. The system as presented above is not a good abstraction of such a physical system because the properties of the abstract components in isolation are very different from the properties of the physical components in isolation: a serial transmitter does not block in the absence of a receiver. In most formal work on distributed systems this kind of distinction is not important because components are interfaced statically. In mobile computing systems, however, components may move about and interface in different ways over the life of the computation. To facilitate realistic and reliable reasoning about such systems, we would like the components to reflect the correct behavior when in both coupled and decoupled modes of operation and when making the transition between the two.

Perhaps the system could have been constructed differently, while remaining within the framework of standard UNITY. However, it is difficult to express this kind of turn-taking synchronization without resorting to shared state indicating which component should act next. The UNITY *superposition* mechanism is designed to express synchronization between two programs, but only in a very limited and stylized way. Superposition on an underlying program F proceeds by adding new statements and variables to F such that the new statements do not assign to any of the original underlying variables of F, and each of the new statements is synchronized with some statement of F. This allows for the maintenance of history variables, that do not change the behavior of the underlying program but are needed for certain kinds of proofs, and construction of layered systems, where the underlying layers are not aware of the higher layer variables. A major contribution of [3] was the examination of program derivation strategies using union and superposition as basic construction mechanisms. From a purely theoretical standpoint, it is natural to ask whether we can rethink these two forms of program composition by reconsidering the fundamentals of program interaction and what abstractions should be used for reasoning about composed programs.

The *sender* [] *receiver* program could have been expressed as a superposition of the receiver on the sender, where the receiver is simply a maintainer of the history of bits transmitted since the last execution of *new*. However, we find this kind of composition unappealing for two reasons. First, it is asymmetric and will not generalize well to situations where the components must communicate in both directions. Second, it is again a static form of composition unsuited for dealing with systems that have mobile components.

Mobile computing systems must operate under conditions of transient connectivity. Connectivity will depend on the current location of components and therefore location may be a part of the model. As we see with the serial communication example, real-time properties are also important, although it may be more elegant to express these constraints with higher-level synchronization constructs rather than explicit models of time. When disconnected, components should behave as expected. This means that the components must not be made too aware of the other programs with which they interface. The sender, for example, must not depend on the presence of a receiver when it transmits a value. It is unrealistic for the sender to block when no receiver is present. However, there are constraints that the two programs must satisfy when they are connected. We wish to express these constraints when the programs are composed, while not

cluttering up the individual components in such a way that they must be aware of and dependent on the existence of other programs. This argues for the development of a coordination language sufficiently powerful to express these interactions and to preserve the modularity of a single program running in isolation. As we will see in the sections that follow, this composition mechanism will have certain aspects in common with UNITY union and other traits characteristic of superposition. In the next section we propose several additions to standard UNITY in preparation for a later introduction of mobile components and transient interactions.

3. Mobile UNITY without Mobility

In this section we define our model of computation employing a UNITY-based notation and proof logic. In the next section we discuss program structuring mechanisms and composition. For now, the notation concerns single programs and, therefore, its applicability to mobile computing will not be immediately obvious. Our contributions to the study of mobile computing will be discussed later—they include explicit modeling of program location and a modular specification of interactions among mobile programs. We postpone for the next section a discussion on how constructs introduced here facilitate the composition of mobile programs in the style of a declarative coordination language.

In standard UNITY, the basic unit of system construction is the program. The structure of a UNITY program was defined in the previous section as consisting of a **declare** section, an **initially** section, and an **assign** section. In our notation we preserve the UNITY syntax for the **declare** and **initially** sections and augment that of the **assign** section. Our investigation into programming abstractions suitable for mobile computing led us to the addition of four constructs to the standard UNITY notation:

- *Transactions* provide a form of sequential execution. They consist of sequences of assignment statements which must be scheduled in the specified order with no other statements interleaved in between. The assignment statements of standard UNITY may be viewed as singleton transactions. We will use the term *normal statement* or simply *statement* to denote both transactions and standard statements in a given program. As before, normal statements are selected for execution in a weakly fair manner and executed either as a single atomic action or as a series of successive atomic actions.
- *Labels* provide a mechanism by which statements can be referenced in other constructs. This provides us with the ability to modify the definitions of existing statements without actually requiring any textual changes to the original formulation.
- *Inhibitors* provide a mechanism for strengthening the guard of an existing statement without modifying the original. This construct permits us to simulate the effect of redefining the scheduling mechanism so as to avoid executing certain statements when their execution may be deemed undesirable.
- *Reactive statements* provide a mechanism for extending the effect of individual assignment statements with an arbitrary terminating computation. All assignment statements of a given program are extended in an identical manner. The reactive statements form a program that is scheduled to

execute to *fixed-point*, a state where no further execution of a reactive statement will modify the system state, after each individual assignment statement including those that appear inside a transaction. This construct allows us to simulate the effects of the interrupt processing mechanisms which are designed to react immediately to certain state changes.

In the remainder of this section we examine each of these new constructs in turn and develop a proof logic that accommodates these notational extensions. A summary of these constructs appear in Figure 1.

Notation	Name	Description
x := e **if** r	assignment	Conditional multiple-assignment statement from standard UNITY.
s :: x := e **if** r	labeled assignment	Label can be added to allow for inhibition.
s :: $\langle s_1; s_2; \ldots ; s_n \rangle$ **if** r	transaction	The sub-statements are executed in sequence. The reactive statements execute to fixed-point after each sub-statement. Transactions themselves may not be reactive.
inhibit s **when** r	inhibiting clause	The statement *s* may not execute when the system is in a state satisfying *r*
x := e **reacts-to** r	reactive statement	Execute the given statement immediately whenever the system is in a state satisfying *r*. May be interleaved with other reactive statements. Reactive statements may not be inhibited.

Figure 1. Mobile UNITY constructs.

The notation for *transactions* assumes the form

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

where s_i must be an assignment statement. Once the scheduler selects this statement for execution, it must first execute s_1, and then execute s_2, etc. In the absence of any reactive statements, the effect is that of an atomic transformation of the program state.

A label may precede any statement and must be followed by the symbol '::' as in

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

All labels must be unique in the context of the entire program and there is no need to label every statement. The primary motivation for the introduction of labels is their use in constructing inhibitors.

The *inhibitor* syntax follows the pattern

inhibit n **when** p

where n is the label of some statement in the program and p is a predicate. The net effect is a strengthening of the guard on statement n by conjoining it with $\neg p$ and thus inhibiting execution of the statement when p is true.

A *reactive statement* is an assignment statement (not a sequence of statements) extended by a reaction clause that strengthens its guard as in

s reacts-to p

The set of all reactive statements, call it $\mathcal{R}$, must be a terminating program. We can think of this program as executing immediately after each assignment statement. To account for the propagation of complex effects, we allow the set of all reactive statements to execute in an interleaved fashion until fixed-point. As $\mathcal{R}$ is merely a standard terminating UNITY program, a predicate $FP(\mathcal{R})$ can be computed which is the largest set of states for which no reactive statement will modify the state when executed. This is the fixed-point of $\mathcal{R}$.

This two-phased mode of computation where every assignment statement is punctuated by a flurry of reactions may seem unreasonable at first, and indeed, it is possible to write completely unrealistic system specifications with many complicated actions relegated to the reactive statements. However, it is also possible to write unrealistic UNITY programs. Assignment statements can be arbitrarily complex and may have no efficient implementation. We favor, however, expressive power over predefined constraints and pursue strategies in which it is the responsibility of the designer to exercise control over the notation in order to achieve an efficient realization on a particular architecture. As shown later, proper use of these constructs will help us to write modular and efficiently implementable specifications of mobile computations.

A program making use of the above constructs is shown below. It consists of two non-reactive statements, one of which is a transaction, one inhibiting clause, and one reactive statement.

```
program toy-example
  declare
    x, debug : integer
  initially
    x = 0
  [] debug = 0
  assign
    s :: x := x + 1
  [] t :: ⟨x := x + 1; x := x - 1⟩
  []      inhibit s when x ≥ 15
  []      debug := x reacts-to x > 15
end
```

The statement s increments x by one. The statement t is a transaction consisting of two sub-statements. The first increments x by one. The second decrements x by one. The programmer might add the inhibiting clause to prevent x from being incremented past 15. This prevents statement s from performing this action, but the statement t may still execute and temporarily increase x to 16. This intermediate state would not be visible to the programmer and indeed the proof logic given below would allow one to prove **inv.** $x \leq 15$ from the text of *toy-example*. Such states can be detected, however, by adding reactive statements such as the last one, which assigns the value of x to *debug* whenever $x > 15$, including during intermediate states of transactions. This is a modular way to add side-effects to a large set of statements without re-writing each statement. We will see later how these aspects of our notation help to model mobile systems.

Now we give a logic for proving properties of programs that use the above constructs. Our execution model has assumed that each non-reactive statement is fairly selected for execution, is executed if not inhibited, and then the reactive program $\mathcal{R}$ is allowed to execute until it reaches a fixed point state, after which the next non-reactive statement is scheduled. In addition, $\mathcal{R}$ is allowed to execute to fixed point between the sub-statements of a transaction. These reactively augmented statements thus make up the basic atomic state transitions of our model and we denote them by s^*, for each non-reactive statement s. We denote the set of non-reactive statements by $\mathcal{N}$. Thus, the definitions for basic **co** and **ensures** properties become:

$$p \textbf{ co } q \equiv \langle \forall s \in \mathcal{N} :: \{p\}\ s^*\ \{q\} \rangle$$

and

$$p \textbf{ ensures } q \equiv p \wedge \neg q \textbf{ co } p \vee q \wedge \langle \exists s \in \mathcal{N} :: \{p \wedge \neg q\}\ s^*\ \{q\} \rangle$$

Even though s^* is really a statement augmented by reactions, we can still use the Hoare triple notation $\{p\}\ s^*\ \{q\}$ to denote that if s^* is executed in a state satisfying p, it will terminate in a state satisfying q. The Hoare triple notation is appropriate for *any* terminating computation.

In hypothesis-conclusion form, we can write an inference rule for deducing $\{p\}\ s^*\ \{q\}$, given some H, a predicate that holds after execution of s in a state where s is not inhibited, and I, an invariant that holds throughout execution of the reactive statements $\mathcal{R}$. We require that H is sufficient to establish I ($H \Rightarrow I$), and that once $\mathcal{R}$ reaches fixed point, q is established ($I \wedge FP(\mathcal{R}) \Rightarrow q$). The following rule holds for non-reactive statements s that are singleton transactions:

$$\frac{p \wedge \iota(s) \Rightarrow q,\ \{p \wedge \neg \iota(s)\} s \{H\},\ H \rightarrow FP(\mathcal{R}) \text{ in } \mathcal{R},\ \textbf{stable } I \text{ in } \mathcal{R},\ H \Rightarrow I,\ I \wedge FP(\mathcal{R}) \Rightarrow q}{\{p\}\ s^*\ \{q\}}$$

For each non-reactive statement s, we define $\iota(s)$ 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. We take $\{p \wedge \neg \iota(s)\} s \{H\}$ from the hypothesis to be a standard Hoare triple for the non-augmented statement s.

For those statements that are of the form $\langle s_1; s_2; \ldots s_n \rangle$ we can use the following inference rule before application of the one above:

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

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

The proof obligations $H \rightarrow FP(\mathcal{R})$ in $\mathcal{R}$ and **stable** I in $\mathcal{R}$ can be proven with standard techniques because $\mathcal{R}$ is treated as a standard UNITY program. We can simplify the rule if we know that the non-reactive statement s will not enable any reactive statements, that is, will leave $\mathcal{R}$ at fixed point. This can be expressed as:

$$\frac{p \wedge \iota(s) \Rightarrow q,\ \{p \wedge \neg \iota(s)\} s \{q\},\ q \Rightarrow FP(\mathcal{R})}{\{p\}\ s^*\ \{q\}}$$

which allows us to substitute the obligation $q \Rightarrow FP(\mathcal{R})$ for the more complicated invariant and fixed-point argument.

The notation and basic inference mechanism provide tools for reasoning about basic programs. Apart from our redefinition of **co** and **ensures**, however, we keep the rest of the UNITY inference toolkit which allows us to derive more complex properties in terms of these primitives. In the following section, we will show how the notation can be used to construct systems of mobile components that exhibit much more dynamic behavior than could be easily expressed with standard UNITY.

4. Adding Mobility and Structured Composition

Our concern with mobility forced us to reexamine the UNITY model. The initial intent was to provide the means for a strong degree of program decoupling, to model movement and disconnection, and to offer high-level programming abstractions for expressing the transient nature of interactions in a mobile setting. Decoupling, defined as the program's ability to continue to function independently of the communication context in which it finds itself, is achieved by making the process namespaces disjoint and by separating the description of the component programs from that of the interactions among components. Mobility is accommodated by attaching a distinguished location variable to each program; this provides both location awareness and location control (locomotion) to the individual programs. The model presented in the previous section is the result of a careful investigation of the implementability of high-level constructs for transient interactions. Reasoning about mobile systems of many components will be carried out in terms of this model.

As distinct from our earlier presentation, this section focuses on composition of several programs rather than the properties of a single program. Coordination is captured implicitly and declaratively by interaction constructs rather than being coded directly into the component programs; we will show how each of the new constructs presented in the previous section contributes to a decoupled style of program composition. The reactive statement captures the semantics of interrupt-driven processing and enables us to express synchronous execution of local and non-local actions. The inhibit clause captures the semantics of processing dependencies. In essence, both kinds of statements express scheduling constraints that cut across the local boundaries of individual components. Extra statements are sometimes added to a composition to capture the semantics of conditional asynchronous data transfer among components. Together, these constructs define a basic coordination language for expressing program interactions. Simple forms of these statements have direct physical realization and can be used to construct a rich set of abstract interactions including UNITY-style shared variables, location-dependent forms of interaction, and clock-based synchronization. Next, we illustrate some of the less tightly coupled forms of interaction by revisiting the serial communication example in a setting in which the participants can actually come together and move apart from each other. After several successive refinements we put forth a version that is faithful to possible physical realizations of the protocol.

Decoupled style of computing. Let us define a system as a closed (static) set of interacting components. In UNITY, a system might consist of several programs which share identically named variables. Each program has a name and a textual description. The operator "[]' is used to specify the assembly of components into a system. In this paper we construct a system in a similar manner but we introduce a syntactic structure that makes clear the distinction between parameterized program types and processes which are the components of the system. A more radical departure from standard UNITY is the isolation of the namespaces of the individual processes. We assume that variables associated with distinct processes are distinct even if they

bear the same name. Thus, the variable *bit* in a program like *sender* from the earlier example is no longer automatically shared with the *bit* in the receiver—they should be thought of as distinct variables. To fully specify a process variable, its name should be prepended with the name of the component in which it appears, for example *sender.bit* or *receiver.bit*. The separate namespaces for programs serve to hide variables and treat them as internal by default, instead of universally visible to all other components. This will facilitate more modular system specifications, and will have an impact on the way program interactions are specified for those situations where programs must communicate.

It is now possible to construct a system consisting of multiple *sender* and *receiver* processes without actually modifying the code presented earlier. We simply add a parameter to the program names and instantiate as many processes as we desire, in this case two senders and one receiver. The resulting system can be specified by a structure such as:

```
System Senders_and_Receivers

  program sender(i)
    ...standard UNITY program...
  end

  program receiver(j)
    ...standard UNITY program...
  end

  Components
    sender(1) []sender(2) []receiver(0)

  Interactions
    ...coordination statements...
end
```

The last section of the system specification, the **Interactions** section, defines the way in which processes communicate with each other. Let's say that we desire *sender(1)* and *receiver(0)* to interact with each other in the style of UNITY by sharing similarly named variables while *sender(2)* remains disconnected. The statements in the **Interactions** section will have to explicitly define these rules using the constructs presented in the previous section, naming variables explicitly by their fully-qualified names. The entire system can be reasoned about using the logic presented in the previous section, because it can easily be re-written into an unstructured program with the name of each variable and statement expanded according to the program in which it appears, and all statements merged into the **assign** section. Our study can now begin in earnest with the issue that motivated us to approach system specifications in this manner in the first place, i.e., the concept of mobility.

Location awareness and control. In mobile computing systems, interaction between components is transient and location-dependent. We consider the individual process to be the natural unit of mobility. Each process has a distinguished variable λ that models its current location. This might correspond to latitude and longitude for a physically mobile platform, or it may be a network or memory address for a mobile agent. A process may have explicit control over its own location which we model by assignment of a new value to the variable modeling its location. In a physically moving system, this statement would need to be compiled into a physical

effect like actions on motors, for instance. Even if the process does not exert control over its own location we can still model movement by an internal assignment statement that is occasionally selected for execution. Any restrictions on the movement of a component should be reflected in this statement.

As an example, we introduce the notion that each *sender* process exists at some fixed location in space. The process is neither aware of nor in control of its own location. We express this fact by the absence of any statements that make reference to or modify the location variable.

```
program sender(i) at λ
  declare
    bit : boolean
  [] word : array[0..N-1] of boolean
  [] c : integer
  initially
    λ = SenderLocation(i)
  assign
    transmit :: bit, c := word[c], c+1        if c < N
  [] new     :: word, c := NewWord(), 0       if c ≥ N
end
```

While the code looks similar to the earlier version, the reader is reminded that henceforth all variables are considered local and only the coordination statements appearing in the **Interactions** section allow components to interface with each other. For readability and clarity in expressing this distinction, some of the variables have been renamed. (Whenever the context is clear we refer to variables by their unqualified names, e.g., *c*, rather than the full name *sender(i).c*.) As before, the *sender* maintains a variable *word* which holds a sequence of bits to be transmitted. The counter *c* is a pointer to the next bit that will be copied to the variable *bit*, which represents the state of some lower-level communications medium. Upon transmitting the current bit, the counter *c* is incremented. When it reaches *N*, no further bits are transmitted until a new word is written to *word* and *c* is reset by the statement *new*. The above program is capable of transmitting bits in complete isolation without any receiver present.

In contrast to the *sender*, let us assume a roving *receiver* that may change location in response to receiving a word containing a valid spatial location. The code assumes the form

```
program receiver(j) at λ
  declare
    bit : boolean
  [] buffer : array[0..N-1] of boolean
  [] c : integer
  assign
    zero    :: c := 0                    if bit = 1 ∧ c ≥ N
  [] receive :: buffer[c], c := bit, c+1  if c < N
  [] move    :: λ := buffer               if ValidLocation(buffer) ∧ c ≥ N
end
```

Upon receipt of a full word which happens to be a valid location the receiver may choose to move to that location before the start of a new data transmission. This happens if the *move* statement is selected for execution. Since we assume the same weak fairness as in standard UNITY, there is no

guarantee that the *move* statement is ever selected upon receipt of a new location. Actually, there is no guarantee that the receiver will detect the arrival of a start bit (value 1) and reset its counter *c* before the sender moves on to sending the next value. A new mechanism is needed to force the scheduler to execute these statements at the right time. We found the solution in the coordination language developed for the **Interactions** section.

Reactive control. Below we give a modified version of the code for the mobile receiver. The statements *move* and *zero* are reactive statements. In the case of statement *zero*, for instance, the statement reacts to the presence of a 1 on the input variable *bit* (while the counter *c* is at least *N*) by resetting the counter *c* to zero. This enables the *receive* statement, which copies bits from the input in sequence into the array *buffer*. Correct execution will therefore require that the first bit of *sender.word* be a 1, and that the last bit be a zero.

```
program receiver(j) at λ
  declare
    bit : boolean
  [] buffer : array[0..N-1] of boolean
  [] c : integer
  assign
    zero     :: c := 0                  reacts-to bit = 1 ∧ c ≥ N
  [] receive :: buffer[c], c := bit, c+1 if c < N
  [] move    :: λ := buffer             reacts-to ValidLocation(buffer) ∧ c ≥ N
end
```

The **reacts-to** *p* construct is used here to model the interrupt triggered by the presence of a 1 on the input line when $c \geq N$, and the actual statement has the effect of zeroing the bit counter and thereby falsifying $c \geq N$.

Asynchronous communication. We now address interprocess communication. Our treatment continues to be informal and focused on refining our example. Because location is modeled like any other state variable, we can use it in the **Interactions** section below to write transient and location-dependent interactions among the components. For example, suppose that the sender and receiver can only communicate when they are at the same location, and we wish to express the fact that *sender(i).bit* is copied to *receiver(j).bit* when this is true. We might begin the **Interactions** section with

```
receiver(j).bit := sender(i).bit    when    sender(i).λ = receiver(j).λ
```

which can appear inside a quantifier over the proper ranges for *i* and *j*. This kind of interaction can be treated like an extra program statement that is executed in an interleaved fashion with the existing program statements. The predicate following **when** is treated like a guard on the statement (**when** can be read as **if**). Note that this interaction alone is not guaranteed to propagate every value written by the sender to the receiver; it is simply another interleaved statement that is fairly selected for execution from the pool of all statements. Thus, *sender(i).transmit* may execute twice before this statement executes once even in a fair execution.

Synchronous communication. Given the observations above, we must strengthen the statement by using **reacts-to** to ensure that every bit transmitted is copied to the receiver, when the two are co-located:

receiver(j).bit := sender(i).bit **reacts-to** sender(i).λ = receiver(j).λ

Recall that the semantics of **reacts-to** imply that the statement will be executed repeatedly as part of a program made up of all reactive statements until that program reaches fixed point. When executed in isolation, this statement reaches fixed point with one execution, after which we can deduce *receiver(j).bit* = *sender(i).bit* ∨ *sender(i).λ* ≠ *receiver(j).λ*. Because this propagation occurs between every step of the two components, it effectively presents a read-only shared-variable abstraction to the *receiver* program, when the two components are co-located. Later we will show how to generalize this notion so that variables shared in a read/write fashion by multiple components can be modeled.

Scheduling constraints. Even if the variable *sender(i).bit* is now copied to the receiver between every high level program statement, we still need additional coordination between the two components. For example, there is no constraint on the number of times *receiver(j).receive* can execute between executions of *sender(i).transmit*. This could lead to undesired behavior where the receiver duplicates bits. Fortunately, each component is maintaining a counter which is the index of the next bit transmitted or received. We can express the synchronization constraint with the **inhibit** interaction construct, continuing the **Interactions** section:

inhibit sender(i).transmit **when** sender(i).c > receiver(j).c ∧ sender(i).λ = receiver(j).λ
inhibit receiver(j).receive **when** receiver(j).c ≥ sender(i).c ∧ sender(i).λ = receiver(j).λ

Operationally, an **inhibit** *s* **when** *p* interaction has the effect of strengthening the guard on the named statement *s* by the conjunct ¬*p*, which is a possibly global state predicate. In this case, the sender is not allowed to transmit when its counter is greater than the receiver's, and the receiver may only receive when its counter is less than that of the sender. Neither constraint has any effect when the components are separated. Thus, a sender that is not co-located with some receiver may increment *sender(i).c* in a free-running fashion without regard to the state of the receiver. Note that when a receiver moves to a new sender the value of *receiver(j).c* is at least *N*, but because the new sender's counter was possibly running free, it may have any value in the range $0 \le$ *sender(i).c* $\le N$. The receiver may then think that any 1 received is a start bit and will reset its counter. The **inhibit** clauses will then cause the sender to wait while the receiver catches up, after which the two processes will be synchronized again. A real system would thus need a more complicated start sequence that does not appear in any data word to avoid fooling receivers in this way. A real receiver would resynchronize only upon receipt of the new start symbol and not somewhere in the middle of a word, as our mechanism might. This is not a failure of our notation but rather the level of abstraction at which we have specified the problem.

The **inhibit** interactions as given may seem to be an unrealistic "action-at-a-distance," but they actually reflect real-time properties that give rise to the turn-taking behavior. In fact, the **inhibit** construct provides a natural way to specify this synchronization at a lower level, if we add a local clock and history variables to each node. The following system specification captures precisely these notions.

```
System Senders_Receivers_Timers

  program sender(i) at λ, t
    declare
      bit : boolean
    [] word : array[0..N-1] of boolean
    [] c, sendstamp : integer
    initially
      λ = SenderLocation(i)
    assign
      transmit :: bit, c := word[c], c+1                    if c < N ∧ t ≥ sendstamp + Δ·c
    [] new     :: word, c, sendstamp := NewWord(), 0, t   if c ≥ N
    [] timer   :: t := t + 1                               if t < sendstamp + Δ·c + Δ/4
  end

  program receiver(j) at λ, t
    declare
      bit : boolean
    [] buffer : array[0..N-1] of boolean
    [] c, recvstamp : integer
    assign
      receive :: buffer[c], c := in, c+1   if c < N ∧ t ≥ recvstamp + Δ·c + Δ/2
    [] zero    :: c, recvstamp := 0, t     reacts-to bit = 1 ∧ c ≥ N
    [] timer   :: t := t + 1               if t < recvstamp + Δ·(c+1) - Δ/4
    [] move    :: λ := buffer              reacts-to ValidLocation(buffer) ∧ c ≥ N
  end

Components
    sender(1) []sender(2) []receiver(0)

Interactions
    receiver(j).bit := sender(i).bit
            reacts-to    sender(i).λ = receiver(j).λ
    inhibit sender(i).timer
            when sender(i).t - sendstamp > receiver(j).t - recvstamp
                ∧ sender(i).λ = receiver(j).λ
    inhibit receiver(j).timer
            when receiver(j).t - recvstamp > sender(i).t - sendstamp
                ∧ sender(i).λ = receiver(j).λ
end
```

The constant Δ is used in each of the programs to represent the nominal time interval (in ticks of the *sender(i).t* or *receiver(j).t* clock) between transmissions or receptions of a bit. The statement *sender(i).transmit* is allowed to execute only if time has advanced to at least the *c*th interval since *sender(i).new* executed. This is a lower bound on the time at which the statement may execute.

The statement *sender(i).timer* is not allowed to execute if it will advance time more than one-fourth of the duration of the current interval before the current bit has been transmitted. This is an upper bound on the time at which *sender(i).transmit* may execute. The receiver has a pair of similar constraints, shifted to allow for reception only after the sender has transmitted a bit, with proper choice of Δ. Reasoning about the correctness of the above protocol will naturally require assumptions about the value of Δ. The expression of the real-time constraints here is similar to the *MinTime* and *MaxTime* of [1], except that we choose here to deal with discrete, local clocks rather than a continuous, global one.

Note that the restrictions on the transmit/receive actions are now completely local and the global **inhibit** interactions merely constrain the two timer values *sender(i).t* and *receiver(j).t* so that they increment at approximately the same rate after initiation of the transmission. The fact that we can again use **inhibit** to express real-time properties in this way suggests that it is fundamental to concurrent composition of realistic programs.

The constructs introduced in this section define a new UNITY-style programming notation. We refer to it as *Mobile UNITY*, in recognition of the driving force behind its development. Even in the absence of mobility, the features of the new notation improve modularity and strengthen separation of concerns. Both movement and interaction statements require a subtle change in the mindset. They represent modeling constructs which are needed to facilitate reasoning about such systems while not over-specifying the component programs. Their possible realization is in terms of mechanical controls (in the case of movement), scheduling constraints and services that are to be provided by the operating system, or physical properties of the transmission medium.

5. Discussion

To control the complexity of mobile-aware applications, researchers will create new programming abstractions that reflect the realities of mobile computing, including disconnections and bandwidth variability, but which are all at once implementable, intuitive, and which facilitate reasoning about the correctness of whole systems of mobile components. While we do not presume to know what these abstractions will be, we hope to show that the notation presented so far is versatile enough to model many different approaches to mobile computing, and therefore can serve as a good basis for describing the formal semantics of these new constructs. Figure 2 provides a summary of the kinds of constructs we were able to build from the primitives introduced in Section 3. These constructs might best be thought of as patterns of program interaction and coordination, derived from traditional communication mechanisms such as shared variables and synchronization. In addition we worked on the formulation of clock synchronization constructs similar in style to the mechanism shown in Section 4.

The first construct to appear in the table supports sharing among variables belonging to different components. The construction is transient in the sense that sharing is controlled by the predicate appearing in the **when** condition. The latter can be made to reflect co-location or some notion of being within radio transmission range. The construct is also transitive. Two variables need not be shared directly. When connectivity is available, a file on the laptop may be shared with a host in the office. In turn the host may share the file across the network with some other distant host. File changes at any one of the three places will propagate atomically. As connections go down, i.e., the **when** condition turns false, the range over which sharing take place is reduced. By maintaining history variables and by employing reactive statements we are able to express this rather complex construct in terms of the basic notation. Special care must be exercised to guarantee that the resulting reactive program does terminate. This can be a problem in Mobile

UNITY because the notation allows the designer a very high degree of freedom with regard to what can be specified. In practice the constructs made available for coordinating mobile programs would be subject to restrictions which will help us offer termination guarantees. We should make it clear that, at the moment, the constructions we built are not to be viewed as practical implementations, only as a demonstration of the expressive power and semantic utility of the basic notation. Three other constructs related to variable sharing appear in the table: one way sharing, engagement which allows for variables having distinct values to agree upon a common value at the time sharing starts, and disengagement which allows, at the termination of a sharing relation, for the parties to assume distinct values.

Notation	Name	Description
A.x ≈ B.y **when** r	shared variable	Changes to either *A.x* or *B.y* are reactively propagated to the other, when the system is in a state satisfying *r*.
A.x ← B.y **when** r	read-only shared variable	Changes to *B.y* are reactively propagated to *A.x*, when the system is in a state satisfying *r*.
engage(A.x, B.y) **when** r **value** ε	engage clause	The expression ε is assigned to both *A.x* and *B.y* reactively upon a transition from a state not satisfying *r* to one that does satisfy *r*.
disengage(A.x, B.y) **when** r **value** δ_1, δ_2	disengage clause	The expression δ_1 is assigned to *A.x* and the expression δ_2 is assigned to *B.y* reactively upon a transition from a state that does satisfy *r* to one that does not satisfy *r*.
A.s][B.t **when** r	coselection	*A.s* and *B.t* are selected for execution simultaneously when the system is in a state satisfying *r*.
xcoselect(A.s, B.t, r)	exclusive coselection	*A.s* and *B.t* are selected for execution simultaneously when the system is in a state satisfying *r*, and may not execute independently even when *r* is false.
coexecute(A.s, B.t, r)	coexecution	*A.s* and *B.t* are selected for execution simultaneously when the system is in a state satisfying *r* and both of the internal guards of *A.s* and *B.t*.
xcoexecute(A.s, B.t, r)	exclusive coexecution	*A.s* and *B.t* are selected for execution simultaneously when the system is in a state satisfying *r* and the internal guards of *A.s* and *B.t*. They may not execute independently even when *r* is false.

Figure 2. High level coordination constructs in Mobile UNITY.

In UNITY, synchronization can be expressed by the use of a parallel bar which is actually treated as a statement constructor more than a synchronization mechanism and by the use of superposition which is asymmetric and allows actions in one program to be extended by actions of

another under certain technical restrictions having to do with variable access rights. Participation in the synchronization is again static. By contrast, we provide transient and transitive forms of synchronization. Coselection forces statements from distinct programs to be selected for execution always together as long as the **when** condition holds true. In this manner, students' laptops entering a classroom could be forced to execute in perfect synchrony with the teacher's stationary host. A stronger form a synchronization involves the added requirement that the synchronized statements all have true guards. In this case the teacher would not be able to start lecturing before all the students in the room are ready. Yet another variant disallows the statement execution unless the **when** condition holds, i.e., coordination is feasible. The teacher may be thus restricted from assigning a grade when the student is not present, in our example. All these constructions involve the use of inhibit statements and rely on the shared variable abstraction discussed above.

The primary motivation for the development of these constructs was the need to explore the expressive power of the notation we proposed and the desire to seek new kinds of high level constructs for building mobile applications. Mobile UNITY, however, is not a language for building systems but a model for the study of fundamental concepts and ideas in mobility. A more pragmatic dimension of this research is also emerging. Mobile UNITY has been used in an exercise on involving the specification and verification of a network protocol, Mobile IP [13] in [9], and to express various forms of code mobility [14]. Several pairwise high-level interaction constructs (e.g., shared variables and statement synchronization) were presented in [15]. These and other efforts to use Mobile UNITY to verify properties of computations involving mobile components will continue. We are also investigating coordination constructs that have effective implementation in the ad-hoc networks setting. There, Mobile UNITY will be used to provide a formal semantic definition for the constructs which would be made available to the developer in the form of some standard application program interface that will offer strong semantic guarantees.

6. Conclusion

The Mobile UNITY notation and logic is the result of a careful reevaluation of the implications of mobility on UNITY, a model originally intended for statically structured distributed systems. We took as a starting point the notion that mobile components should be modeled as programs (by the explicit addition of an auxiliary variable representing location), and that interactions between components should be modeled as a form of dynamic program composition (with the addition of coordination constructs). The UNITY-style composition, including union and superposition, led to a new set of basic programming constructs amenable to a dynamic and mobile setting. We applied these constructs to a low-level communication task in an attempt to show that the basic notation is useful for realistic specifications involving disconnection. The seemingly very strong reactive semantics matched well the need to express dynamically changing side-effects of atomic actions. Finally, we explored the expressive power of the new notation by examining new transient forms of shared variables and synchronization, mostly natural extensions of the comparable non-mobile abstractions of interprocess communication. The notation promises to be a useful research tool for investigating new abstractions in mobile computing. These problems have only recently received attention in the engineering and research community, and formal reasoning has an important role to play in communicating and understanding proposed solutions as well as the assumptions made by each.

References

[1] M. Abadi and L. Lamport, "An Old-fashioned Recipe for Real Time," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 5, pp. 1543-71, 1994.

[2] B. Alpern and F. B. Schneider, "Defining Liveness," *Information Processing Letters*, vol. 21, no. 4, pp. 181-5, 1985.

[3] K. M. Chandy and J. Misra, *Parallel Program Design: A Foundation*. Addison-Wesley, 1988.

[4] E. D. Dijkstra, *A Discipline of Programming*. Prentice-Hall, 1976.

[5] G. H. Forman and J. Zahorjan, "The Challenges of Mobile Computing," *IEEE Computer*, vol. 27, no. 4, pp. 38-47, 1994.

[6] C. G. Harrison, D. M. Chess, and A. Kershenbaum, "Mobile Agents: Are they a good idea?," IBM T.J. Watson Research Center RC 19887, 1995.

[7] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 10, pp. 576-580,583, 1969.

[8] T. Imielinski and B. R. Badrinath, "Wireless Computing: Challenges in Data Management," *Communications of the ACM*, vol. 37, no. 10, pp. 18-28, 1994.

[9] P. J. McCann and G.-C. Roman, "Mobile UNITY Coordination Constructs applied to Packet Forwarding for Mobile Hosts," *Second International Conference on Coordination Languages and Models*, Berlin, pp. 338-354, 1997.

[10] J. Misra, "A Logic for Concurrent Programming: Progress," *Journal of Computer and Software Engineering*, vol. 3, no. 2, pp. 273-300, 1995.

[11] J. Misra, "A Logic for Concurrent Programming: Safety," *Journal of Computer and Software Engineering*, vol. 3, no. 2, pp. 239-72, 1995.

[12] O. Nierstrasz and T. D. Meijler, "Requirements for a Composition Language," *Proceedings of the ECOOP '94 Workshop on Models and Languages for Coordination of Parallelism and Distribution*, Bologna, Italy; 5 July 1994, pp. 193, 147-61, 1995.

[13] C. Perkins, "IP Mobility Support," IETF Network Working Group, Technical Report RFC 2002, October 1996.

[14] G. P. Picco, G.-C. Roman, and P. J. McCann, "Expressing Code Mobility in Mobile UNITY," *Sixth European Software Engineering Conference (ESEC'97)*, Zurich, pp. 500-518, 1997.

[15] G.-C. Roman, P. J. McCann, and J. Y. Plun, "Mobile UNITY: Reasoning and Specification in Mobile Computing," *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 250-282, 1997.

[16] M. Satyanarayanan, J. J. Kistler, L. B. Mummert, M. R. Ebling, P. Kumar, and Q. Lu, "Experience with Disconnected Operation in a Mobile Computing Environment," *Proceedings of the USENIX Symposium on Mobile and Location-Indepedent Computing*, Cambridge, MA, pp. 11-28, 1993.

[17] B. N. Schilit, N. Adams, and R. Want, "Context-Aware Computing Applications," *Proceedings of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, pp. 85-90, 1994.

[18] D. Terry, M. Theimer, K. Petersen, A. Demers, M. Spreitzer, and C. Hauser, "Managing Update Conflicts in Bayou, a Weakly Connected Replicated Storage System," *Operating Systems Review*, vol. 29, no. 5, pp. 172-83, 1995.

[19] G. M. Voelker and B. N. Bershad, "Mobisaic: An Information System for a Mobile Wireless Computing Environment," *Proceedings of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, pp. 185-90, 1994.