Report Number: WUCS-97-01

1997-01-01

Mobile UNITY: A Language and Logic for Concurrent Mobile Systems

Peter J. McCann and Gruia-Catalin Roman

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.**

Mobile UNITY: A Language and Logic for Concurrent Mobile Systems

Peter J. McCann and Gruia-Catalin Roman

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. We also show how the model can contribute to our understanding of mobility by exploring new abstractions for loosely coupled communication and coordination among components.

School of Engineering & Applied Science

Mobile UNITY: A Language and Logic for Concurrent Mobile Systems

Peter J. McCann
Gruia-Catalin Roman

WUCS-97-01

19 December 1996

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

Mobile UNITY: A Language and Logic for Concurrent Mobile Systems

Peter J. McCann
Gruia-Catalin Roman

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. We also show how the model can contribute to our understanding of mobility by exploring new abstractions for loosely coupled communication and coordination among components.

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 Grant No. CCR-9217751. 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.

Correspondence: All communications regarding this paper should be addressed to

Dr. Gruia-Catalin Roman
Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
Saint Louis, MO 63130-4899

office: (314) 935-6190
secretary: (314) 935-6160
fax: (314) 935-7302

roman@cs.wustl.edu
http://www.cs.wustl.edu/~roman/

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 [7, 11]. 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 [20], a filesystem supporting disconnected operation, and Bayou [22], 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 [23]. 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 [21], 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 [18] 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 today's 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 [8]. 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 [17].

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 give an overview of UNITY and the motivation for our later extensions. Section 3 is a succinct introduction to our new notation, called Mobile UNITY, and gives a formal axiomatic definition of each construct. In Section 4, we show how these new basic constructs can play a role in a structured notation for defining systems of mobile components. Section 5 makes use of the new constructs in an investigation of novel high-level abstractions of communication and coordination in mobile systems. Conclusions are presented in Section 6.

2. UNITY Overview

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 [16], 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* [9] 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 [15] 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 increase or be reset to zero. This can be expressed as

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

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.

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

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

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

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 give a formal presentation of our basic model. Subsequent sections are devoted to program composition and examples that justify our design choices and elucidate fundamental mobile computing issues.

3. Notation and Logic

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.

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 label 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 sequences 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⇒I), and that once $\mathcal{R}$ reaches fixed point, q is established (I∧FP($\mathcal{R}$)⇒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 sections, 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. 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. In the next section we will use transaction statements to express statement synchronization across components in the style of UNITY superposition. For now 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. More abstract forms of interaction are discussed in the following section.

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. This is the reason why 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* $\vee$ *sender(i).*$\lambda \neq$ *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 $\wedge$ sender(i).λ = receiver(j).λ
inhibit receiver(j).receive **when** receiver(j).c $\geq$ sender(i).c $\wedge$ 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 $\neg p$, 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 cth 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. High-Level Programming Abstractions for Mobile Computing

Mobile applications present new challenges to distributed systems design and are likely to exhibit unprecedented degrees of complexity. For example, many researchers advocate exposing information about network connectivity to applications—this is presumed to allow an application to make the best use of the limited communication resources by exploiting semantic knowledge about the problem domain [24]. While this might be an effective strategy, it results in increased complexity over comparable non-mobile applications. Throughout the history of computer science, management of complexity has been made possible through the intelligent application of abstraction. We believe that the new field of mobile computing is no different. 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. As illustrations, we examine several promising abstractions inspired by traditional models of interprocess communication in distributed systems.

The notation developed in Section 3 can be used directly, as in Section 4, to create modular and realistic specifications for systems of mobile components. Toolkits for building mobile applications, however, will necessarily provide higher levels of abstraction, and not all of the primitives introduced in Section 3 will be directly available to the programmer. Special purpose languages or libraries [13] must be specified and implemented to aid the programming task. The semantics of new protocols for mobile computing, such as Mobile IP [18], must be carefully investigated and understood. Each of these tasks will require the introduction of new abstractions for program communication and coordination. Will these new abstractions be realistically implementable? Can we reason formally about systems that use such constructs? As a starting point, we will draw inspiration from existing abstractions for communication in distributed systems, such as the shared variables and synchronization of UNITY. We will give a rigorous specification of these concepts in terms of the notation introduced in Section 3, and discuss the implementability of each in the mobile setting.

For each high level construct discussed in this section we will offer some notational shorthand, and we will define its semantics in terms of transactions, reactive statements, and inhibiting clauses, along with assumptions about and constraints on the program units involved. The syntactic shorthand may have unbound parameters representing program variables or statements, and may introduce auxiliary variables that will be used only within the definition. The assumptions made by each abstraction may be that the unbound statements and variables be of a certain form or that the program in which the definition is invoked must contain other variables, such as one representing time. These constraints will be stated explicitly when the definitions are presented. The abstractions might therefore best be thought of as patterns of program interaction and coordination, derived from traditional communication mechanisms via a careful consideration of the impact of disconnected and low-bandwidth operation on each.

5.1 Transient Sharing

An obvious starting point for inter-program communication mechanisms is the atomic shared variable. This is the basic communication primitive of UNITY programs, where any two variables with the same name are considered shared by default. In the mobile setting, variables from two independently moving programs are not always connected, and this is reflected in our model by the isolation of each of the namespaces. Consider, however, a programming language for mobile applications that provides a notion of distributed shared memory. Because connectivity is not always available, such memory must have a weak consistency model during periods of disconnection, which means updates to isolated cached copies of a variable are not immediately propagated. When sufficient connectivity becomes available, we can switch to a strong consistency model. For example, we may have a printer management utility running on a mobile laptop that maintains a queue of pending print requests. We abstract the state of this queue by its length and assume the existence of an integer program variable *Laptop.Q* which, as its name implies, is inside the program *Laptop*. We assume that when a high degree of connectivity is available between the laptop and a given printer, the print queues may be shared in such a way that updates are atomic; i.e., all operations are guarded with sufficient locking mechanisms to prevent overlapping access to the shared state. These operations could include the laptop appending or deleting items from the queue, and the printer deleting items from the head of the queue as it finishes each job. We could denote such transient sharing with an interaction of the form

$$\text{Laptop.Q} \approx \text{Printer.Q} \textbf{ when } \text{Laptop.}\lambda = \text{Printer.}\lambda \qquad (1)$$

Also, we abstract away from many characteristics of the network and assume that two components in the same location are sufficiently connected to carry out the strongly consistent replication protocol. As in Section 4, we assume that each program has a distinguished variable λ denoting its current location. To express clearly the meaning of this construct, we will describe its formal semantics in terms of the primitives introduced in Section 3, and discuss issues relating to both formal reasoning and realistic implementation.

One possible interpretation of (1) is that files are simply sent to the printer one by one, when connectivity is available. When the programs move apart, no communication is possible and files cannot be transmitted. This could be expressed by the following interaction:

Laptop.Q, Printer.Q := Laptop.Q - 1, Printer.Q + 1 **when** Laptop.λ = Printer.λ $\wedge$ Laptop.Q > 0 (2)

This is an extra statement in the **Interactions** section that according to our semantics is fairly interleaved with the rest of the statements in the system. It models the transfer of one file from the laptop to the printer by decrementing the *Laptop.Q* variable and incrementing the *Printer.Q* variable. While this might be a valid abstraction of some mechanism for propagating files, it is not a straightforward translation of the $\approx$ construct presented above because we had to use the semantics of the queue to express the transfer of information. Also, it is not really what we had in mind by a shared variable.

A shared variable implies not only that updates to one variable are somehow propagated to another, but that the propagation happens in the same atomic step as the update that initiates it. Recall that the serial sender/receiver pair studied in the previous section has already exhibited behavior resembling a shared variable in this respect. The systems *Senders_and_Receivers* and *Senders_Receivers_Timers* implemented the shared variable *bit* with the **reacts-to** construct. This variable was write-only from the *sender*'s point of view, and read-only from the *receiver*'s perspective. We can construct a bi-directional read/write shared variable, the traditional shared variable of UNITY, by adding another reactive statement that copies values in the other direction. We add some auxiliary variables (the ones with subscripts below) to make sure that new values are not overwritten by old values. For example, sharing the printer queues might be expressed by the two interactions:

Laptop.Q, Laptop.$Q_{Printer.Q}$, Printer.$Q_{Laptop.Q}$:=
Printer.Q, Printer.Q, Printer.Q
reacts-to Printer.Q $\neq$ Printer.$Q_{Laptop.Q}$ $\wedge$ Laptop.λ = Printer.λ (3)

Printer.Q, Printer.$Q_{Laptop.Q}$, Laptop.$Q_{Printer.Q}$:=
Laptop.Q, Laptop.Q, Laptop.Q
reacts-to Laptop.Q $\neq$ Laptop.$Q_{Printer.Q}$ $\wedge$ Laptop.λ = Printer.λ (4)

This construction propagates changes to either variable, when the two programs are connected, as a reaction to any update. The auxiliary variables are used to maintain a one-step history so that only new values are propagated. For instance, the *Laptop.*$Q_{Printer.Q}$ variable holds the last value that was transmitted from the laptop to the printer. Only when *Laptop.Q* holds a different value, and connectivity is available, will changes be propagated to the printer.

Modeling the shared printer queue in the above way follows our intuition for the case where connectivity is continuous, but may diverge from our expectations in the case of disconnection and reconnection. For instance, if changes have been made to both queues during a period of disconnection, e.g., the laptop has added files and the printer has serviced some requests and shortened its copy of the queue, and connectivity is then re-established, then both reactive statements are enabled. Depending on which is selected for execution first, the changes made by one side will be wiped out. We might fix this problem with the construction below, for two programs *A* and *B* and variables *A.x* and *B.y*, where the predicate *r* may reference the locations of the two components and therefore reflects the state of connectivity between the two programs:

$A.x \approx B.y$ **when** r $\equiv$

$A.x, A.x_{B.y}, B.y_{A.x} := B.y, B.y, B.y$	**reacts-to** $B.y \neq B.y_{A.x} \wedge r$
$B.y_{A.x} := B.y$	**reacts-to** $B.y \neq B.y_{A.x} \wedge \neg r$
$B.y, B.y_{A.x}, A.x_{B.y} := A.x, A.x, A.x$	**reacts-to** $A.x \neq A.x_{B.y} \wedge r$
$A.x_{B.y} := A.x$	**reacts-to** $A.x \neq A.x_{B.y} \wedge \neg r$ (5)

In some circumstances, it may be convenient to split this two-way sharing into two one-way constructs, so that read-only variables can be expressed:

$A.x \leftarrow B.y$ **when** r $\equiv$

$A.x, A.x_{B.y}, B.y_{A.x} := B.y, B.y, B.y$	**reacts-to** $B.y \neq B.y_{A.x} \wedge r$
$B.y_{A.x} := B.y$	**reacts-to** $B.y \neq B.y_{A.x} \wedge \neg r$ (5a)

$B.y \leftarrow A.x$ **when** r $\equiv$

$B.y, B.y_{A.x}, A.x_{B.y} := A.x, A.x, A.x$	**reacts-to** $A.x \neq A.x_{B.y} \wedge r$
$A.x_{B.y} := A.x$	**reacts-to** $A.x \neq A.x_{B.y} \wedge \neg r$ (5b)

Here the local history variables are updated every time a variable changes even if connectivity is not available. Now when two independently modified variables become connected, one of them is not immediately wiped out. However, these semantics still might not match our intuition for a shared variable. As soon as one is changed subsequent to the reconnection, propagation takes place and the other value is overwritten.

Instead of wiping out these changes we would like to integrate them according to some programmer-specified policy. For inspiration we can look to filesystems and databases like [20] and [22] that operate in a disconnected mode. Here the program variables would be replicated files or records of a database, and update propagation is possible only when connectivity is available. These systems also provide a way for the programmer to specify reintegration policies, which indicate what values the variables should take on when connectivity is re-established after a period of disconnection. We call this an **engage** value. The programmer may also wish to specify what values each variable should have upon disconnection. We call these **disengage** values. We can use our notation to specify **engage** and **disengage** values that react to changes in the truth of predicate *r*, using **reacts-to**. Returning to our queue length example, we might specify it as follows:

engage(Laptop.Q, Printer.Q)

when Laptop.λ = Printer.λ

value Laptop.Q + Printer.Q ≡

Laptop.Q, Printer.Q, $\text{status}_{\text{Laptop.Q, Printer.Q}}$:=

Laptop.Q + Printer.Q, Laptop.Q + Printer.Q, true

reacts-to Laptop.λ = Printer.λ $\wedge$ $\neg\text{status}_{\text{Laptop.Q, Printer.Q}}$ (6)

disengage(Laptop.Q, Printer.Q)

when Laptop.$\lambda \neq$ Printer.λ

values 0, Printer.Q ≡

Laptop.Q, Printer.Q, $\text{status}_{\text{Laptop.Q, Printer.Q}}$:=

0, Printer.Q, false

reacts-to Laptop.$\lambda \neq$ Printer.λ $\wedge$ $\text{status}_{\text{Laptop.Q, Printer.Q}}$ (7)

Here the engagement policy is to add the queue lengths together, which models a transfer of all files from the laptop to the printer while maintaining the laptop's access to the queue and the ability to add and remove items from it. The disengage policy is to leave all files on the printer, and the laptop's queue length is set to zero to model the fact that it no longer has access to the shared state. The variable $\textit{status}_{\textit{Laptop.Q, Printer.Q}}$ is an auxiliary variable that must not appear in the original system definition. It keeps track of the changes to connectivity and allows the reactive statements to modify the shared variables appropriately. For instance, when the programs move into the same location after a period of disconnection, the *status* variable will be false, which will allow the **engage** statement to react. Similarly, when the programs move apart after a period of connection, the *status* variable will be true, which will allow the **disengage** statement to react. In general, we can combine the sharing, engagement, and disengagement into one construct for transiently shared variables, although in some cases we may still use (5a), (5b), (6), and (7) as separate constructs:

A.x $\approx$ B.y **when** r

engage ε

disengage δ_1, δ_2 ≡

A.x, $\text{A.x}_{\text{B.y}}$, $\text{B.y}_{\text{A.x}}$:= B.y, B.y, B.y	**reacts-to** B.y $\neq$ $\text{B.y}_{\text{A.x}}$ $\wedge$ r
$\text{B.y}_{\text{A.x}}$:= B.y	**reacts-to** B.y $\neq$ $\text{B.y}_{\text{A.x}}$ $\wedge$ $\neg$r
B.y, $\text{B.y}_{\text{A.x}}$, $\text{A.x}_{\text{B.y}}$:= A.x, A.x, A.x	**reacts-to** A.x $\neq$ $\text{A.x}_{\text{B.y}}$ $\wedge$ r
$\text{A.x}_{\text{B.y}}$:= A.x	**reacts-to** A.x $\neq$ $\text{A.x}_{\text{B.y}}$ $\wedge$ $\neg$r
A.x, B.y, $\text{status}_{\text{A.x, B.y}}$:= ε, ε, true	**reacts-to** r $\wedge$ $\neg\text{status}_{\text{A.x, B.y}}$
A.x, B.y, $\text{status}_{\text{A.x, B.y}}$:= δ_1, δ_2, false	**reacts-to** $\neg$r $\wedge$ $\text{status}_{\text{A.x, B.y}}$ (8)

Systems like [20] and [22] have a definite notion of reintegration policies like **engage** values when a client reconnects to a fileserver or when two replicas come into contact. Specification of **disengage** values may be of less practical significance unless disconnection can be predicted in advance. Although this is not feasible for rapidly reconfiguring systems like mobile telephone networks, it may in fact be a good abstraction for the

file hoarding policies of [20], which can be carried out as a user prepares to take his laptop home at the end of a workday, for instance.

The transient sharing construct given above is a relationship between two variables. It is natural to ask how a system might behave in the presence of many such pairwise interactions and whether the relation is transitive, i.e., if x is shared with y and y with z, is z updated whenever x changes? Figure 5.1 shows this situation, where the variables are represented as circles and edges indicate propagation of a change in x.

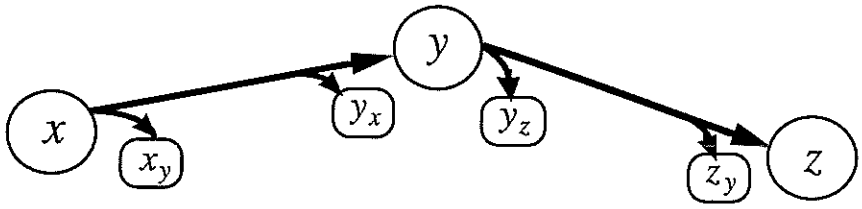

Figure 5.1. Three variables making up a maximal connected component and shared according to (8). An update to any single member of the component is reactively propagated to all other members. Auxiliary variables, shown with subscripts, are updated accordingly.

Given a set of program variables and a collection of sharing constraints of the form (8), we can construct a graph where the vertices are the variables and an edge is placed between any two nodes for which the predicate r is true. Because the state of the computation evolves over time, this graph changes dynamically to reflect the state of each connection between pairs of variables. At any instant, a given variable x is part of a unique maximal connected component consisting of the largest set of vertices v for which a path exists from x to v. Because all **reacts-to** statements are executed until fixed point, any change to one variable of a connected component is propagated to all members of that component, and the interaction as specified can in fact express multi-way and transitive sharing.

However, there are some unexpected consequences that may arise unless we make some simple restrictions on the ways in which shared variables may be used. Recall that to prove properties of a Mobile UNITY program, we must prove that the reactive program terminates after the execution of each non-reactive statement. First, consider what happens if a non-reactive statement assigns distinct values to two or more variables in the same shared group. The above constructs for reactive propagation of values may then execute in such a way that they never reach fixed point, if there are any cycles in the graph modeling the connections among the variables. To see this, consider a cycle of three nodes, x, y, and z, with directed edges (x, y), (y, z), and (z, x), and consider the case where x and y have each been updated to distinct values different from z. Reactions are selected for execution in a fairly interleaved manner, and so propagation from y to z may occur before propagation from x to y. There exists a perfectly fair execution in which a value is updated at each step, but in which there are always two enabled propagations. Termination cannot be proven. Therefore, we must either require that there are no cycles in the shared variable graph, or that assignment of distinct values to more than one variable in a group by the same step be prohibited. This will become even more important in the next subsection which discusses synchronized statement execution.

Second, we must be careful that the reactive program as a whole, not just propagation but including engagements and disengagements, eventually arrives at fixed point. This means we must be very careful when

specifying sharing predicates r that may depend on the values of the variables controlled by that predicate. It is very easy to write a pair of shared variables whose engagement value causes disengagement and whose disengagement value causes engagement. This will obviously lead to a non-terminating reactive program. In most of our work to date, we have emphasized sharing that depends on the relative location of the two components and not on the values of the variables to be shared. However, our model is very general and would allow the specification of such sharing conditions. Either this must be disallowed or great care must be taken when proving properties of such programs.

In the above constructs, our semantics have made very strong assumptions about when movement is allowed. Because all updates to shared variables are made in what we consider to be one atomic step, movement is only allowed between updates. This is sometimes a reasonable assumption and sometimes not. For example, if we are modeling a system of mobile agents which each have control over their own location, we can also assume a runtime system or scheduler that handles requests for movement and truly interleaves them with computation in an atomic way. Even in the case where programs reside on different machines that communicate via a wireless link subject to disconnection, we might envision a user interface that allows the user to warn the system when he intends to carry the laptop out of range. Such an interface in fact exists in the Coda system [20], which allows users to hoard needed files before disconnection.

Of course this is not a valid assumption in every situation, for example when we try to model directly a mobile telephone system. Users travel between base stations at will and without warning. Also, an operating system for a wireless laptop may be attempting to hide mobility from its users and should be written in such a way that it can handle sudden, unpredictable disconnection. Such a system would also be more robust against network failures not directly related to location. Because of well known results on the impossibility of distributed consensus in the presence of failures [6], it would be impossible to implement (3), (4), (6), or (7) in this situation. Interestingly, the semantics of (5) are in fact implementable, because they embody at-most-once semantics for update of the remote variable. If we consider the case where movement occurs during the propagation, we could implement some mechanism (such as checksums) that would enable the receiver to throw away the contents of a partial message, and so the semantics would be the same as if the movement had happened before the update. Because movement is modeled as another interleaved statement, any proofs must be done for both the case where movement occurs before the update and the case where it happens after. Therefore, if we confine ourselves to interactions of the form (5), any properties that hold under the assumption that movement is an atomically interleaved action also hold in the more realistic case that updates have a non-zero duration that may be interrupted by movement.

5.2 Transient Synchronization

The previous subsection presented new abstractions for shared state among mobile components, where such sharing is necessarily transient and location dependent, and where the components involved execute asynchronously. However, synchronous execution of statements is also a central part of many models of distributed systems. In this section we investigate some new high-level constructs for synchronizing statements in a system of mobile components, trying to generalize the synchronization mechanisms of existing

non-mobile models. For example, CSP [10] provides a general model in which computation is carried out by a static set of sequential processes and communication (including pure synchronization) is accomplished via blocking, asymmetric, synchronous, two-party interactions called *Input/Output Commands.* The I/O Automata model [14] expresses communication via synchronization of a named output action with possibly many input actions of the same name. Statement synchronization is also a part of the UNITY model, where it provides a methodology for proving properties of systems that can only be expressed with history variables and for the construction of layered systems.

In UNITY, synchronous execution is expressed via *superposition*, in which a new system is constructed from an *underlying program* and a collection of new statements. Each new statement may be added in one of two ways. First, a new statement r may augment an existing statement s of the underlying program to produce a new statement $s \parallel r$, which is a statement whose semantics are the same as evaluating the right hand sides of both s and r simultaneously, and then assigning the result to the left hand sides. Second, a new statement may simply be added to the collection of statements that will be executed in an interleaved fashion, analogous to the UNITY *union* mechanism for combining programs. Because the goal is to preserve all properties of the underlying program in the new system, the new statements must not assign values to any of the variables of the underlying program. In this way, all execution behaviors that were allowed by the underlying program executing in isolation are also allowed by the new superposed system, and any properties of the underlying program that mention only underlying variables and were proven only from the text of the underlying program are preserved. The augmented statements may be used to keep histories of the underlying variables or to present an abstraction of the underlying system as a service to some higher layer environment.

UNITY superposition is an excellent example of how synchronization can be used as part of a design methodology for distributed systems. It also shows an important distinction between our notion of synchronization, which is the construction of new, atomic statements from two or more simpler atomic statements by executing them in parallel, and the notion of *synchronous computing* which is a system model characterized by bounded communication and computation delays [5]. While the latter is a very important component of current understanding of distributed systems, and in many circumstances is perhaps a prerequisite to implementation of the former, it is not our focus here. Rather, we examine mechanisms that allow us to compose programs and combine a group of statements into a new one through parallel execution. This idea of statement co-execution was inspired by superposition.

However, UNITY superposition is limited in two important ways. First, a superposed system is statically defined and synchronization relationships are fixed throughout the execution of the system. Continuing our theme of modeling mobility with a kind of transient program composition, we would like the ability to specify dynamically changing and location dependent forms of synchronization where the participants may enter into and leave synchronization relationships as the computation evolves. Static forms of statement synchronization are more limited as discussed in [19]. Second, superposition is an asymmetric relationship that subsumes one program to another and disallows any communication from the superposed to the underlying program. While this is the source of the strong formal results about program properties, such a

restriction may not be appropriate in the mobile computing domain where two programs may desire to make use of each other's services and carry out bi-directional communication while making use of some abstraction for synchronization.

Inspired by UNITY superposition which combines statements into new atomic actions, we will now explore some synchronization mechanisms for the mobile computing domain. These take the form of coordination constructs involving statements from each of two separate programs. Informally, the idea is to allow the programmer to specify that the two statements should be combined into one atomic action when a given condition is true. For example, consider two programs A and B, where A contains the integer x and B contains the integer y. Assume there is a statement named *increment* in each program, where *A.increment* is

increment :: x := x + 1

and *B.increment* is

increment :: y := y + 1

Let us assume the programs are mobile, so each contains a variable λ, and that they can only communicate when co-located. Also, assume that the counters represent some value that must be incremented simultaneously when the two hosts are together. We might use the following notation in the **Interactions** section to specify this coordination:

$$\text{A.increment} \;]\!|[\; \text{B.increment} \;\; \textbf{when} \;\; (\text{A}.\lambda = \text{B}.\lambda)$$

Note that this does not prohibit the statements from executing independently when the programs are not co-located. If our correctness criteria state that the counters must remain synchronized at all times, we could add the following two **inhibit** clauses to the **Interactions** section:

$$\textbf{inhibit} \;\; \text{A.increment} \;\; \textbf{when} \;\; (\text{A}.\lambda \neq \text{B}.\lambda)$$
$$\textbf{inhibit} \;\; \text{B.increment} \;\; \textbf{when} \;\; (\text{A}.\lambda \neq \text{B}.\lambda)$$

As distinct from standard UNITY superposition, the]|[construct is a mechanism for synchronizing pairs of statements rather than specifying a transformation of an underlying program. Also, the interaction is transient and location dependent, instead of static and fixed throughout system execution.

To reason formally about transient statement synchronization, we must express it using lower-level primitives. The basic idea is that each statement should react to selection of the other for execution, so that both are executed in the same atomic step. We can accomplish this by separating the selection of a statement from its actual execution, and assume for example that a statement $A.s$ is of the form:

$$\begin{aligned} \text{A.s.driver} :: \; & \langle \text{A.s}_{\text{phase}} := \text{GO};\; \text{A.s}_{\text{phase}} := \text{IDLE} \rangle \\ & \text{A.s.action} \;\|\; \text{A.s}_{\text{f}} := \text{false} \quad \textbf{reacts-to} \;\; \text{A.s}_{\text{phase}} = \text{GO} \;\wedge\; \text{A.s}_{\text{f}} \\ & \text{A.s}_{\text{f}} := \text{true} \qquad\qquad\;\; \textbf{reacts-to} \;\; \text{A.s}_{\text{phase}} = \text{IDLE} \end{aligned} \tag{1}$$

where $A.s_{phase}$ is an auxiliary variable that can hold a value from the set {GO, IDLE}, and *A.s.action* is the actual assignment that must take place. Note that *A.s.action* reacts to a value of GO in $A.s_{phase}$ and that $A.s_f$ is

simultaneously set to *false* so that the action executes only once. When $A.s_{phase}$ returns to IDLE, the flag $A.s_f$ is reset to *true* so that the cycle can occur again. The non-reactive statement *A.s.driver* will be selected fairly along with all other non-reactive statements, and because *A.s.action* reacts during this transaction, the net effect will be the same as if *A.s.action* were listed as a simple non-reactive statement. However, expressing the statement with the three lines above gives us access to and control over key parts of the statement selection and execution process. Most importantly, we can provide for statement synchronization by simply sharing the *phase* variable between two statements. Assuming both statements are of the form (1), we can define ⟩⟨ as:

$$A.s \rangle\langle B.t \textbf{ when } r \quad \equiv \quad A.s_{phase} \approx B.t_{phase} \textbf{ when } r \tag{2}$$

Then, whenever one of the statements is selected for execution by executing *A.s.driver* or *B.t.driver*, the corresponding *phase* variable will propagate to the other statement and reactive execution of *B.t.action* or *A.s.action* will proceed. Also, transitive and multi-way sharing will give us transitive and multi-way synchronization. Note that if we wish to disable the participants from executing, as we did with **inhibit** above, we must be sure to inhibit all participants at the level of the named *driver* transactions. If we inhibit only some of them, they may still fire reactively if ⟩⟨ is used to synchronize them with statements that are not inhibited. We call the ⟩⟨ operator *coselection* because it represents simultaneous selection of both statements for execution. When used in the **Interactions** section of a system, it embodies the assumption that statement execution is controlled by a *phase* variable, as in (1).

The semantics of (1) and (2) do not really guarantee simultaneous execution of the statements in the same sense as UNITY ||, but rather that the statements will be executed in some interleaved order during the reactive program. In many cases this will be equivalent to simultaneous execution because neither statement will evaluate variables that were assigned to by the other. However, there may be cases when we desire both statements to evaluate their right-hand-sides in the old state without using values that are set by the other statement. For these cases, we can add another computation phase to (1) which models the evaluation of right-hand sides as a separate step from assignment to left-hand variables:

$$\begin{aligned}
A.s.driver :: \ & \langle A.s_{phase} := LOAD;\ A.s_{phase} := STORE;\ A.s_{phase} := IDLE \rangle \\
& A.s.load \parallel A.s_{lf} := false \quad \textbf{reacts-to} \quad A.s_{phase} = LOAD \wedge A.s_{lf} \\
& A.s.store \parallel A.s_{sf} := false \quad \textbf{reacts-to} \quad A.s_{phase} = STORE \wedge A.s_{sf} \\
& A.s_{lf}, A.s_{sf} := true, true \quad \textbf{reacts-to} \quad A.s_{phase} = IDLE
\end{aligned} \tag{3}$$

Here the *phase* variables may hold values from the set {LOAD, STORE, IDLE} and the original *A.s.action* is split into two statements, one for evaluating and one for assigning. *A.s.load* is assumed to evaluate the right-hand side of *A.s.action* and store the results in some internal variables that are not given explicitly here. *A.s.store* is assumed to assign these values to the left-hand variables of *A.s.action*. In this way, statements can still be synchronized by sharing *phase* variables as in (2), but now all statements will evaluate right-hand sides during the LOAD phase, will assign to left-hand variables during the STORE phase, and will reset the two flags during the IDLE phase. This prevents interference between any two synchronized statements, even if the

two are connected indirectly through a long chain of synchronization relationships, and even if variables assigned to by the statements are shared indirectly. For example, returning to the *increment* example and considering the following set of interactions:

A.increment ⟦ B.increment **when** (A.λ = B.λ)
A.x ≈ C.z **when** r
C.z ≈ B.y **when** r

Here the statements *A.increment* and *B.increment* are synchronized, but the variable *A.x* is indirectly shared with the variable *B.y*, via the intermediate variable *C.z*, when the predicate *r* is true. If the increment statements are of the form (1), then changes to one variable may be inadvertently used in computing the incremented value of the other variable, which seems to violate the intuitive semantics of simultaneous execution. In contrast, increment statements of the form (3) have a separate LOAD phase for computing the right-hand sides of assignment statements and shared variables are not assigned to during this phase. Assignment to shared variables, and the associated reactive propagation of those values, is reserved until the STORE phase. This isolates the assignment statements from one another and prevents unwanted communication.

There may be situations, however, where we do wish the two statements to communicate during synchronized execution. This strategy is central to models like CSP and I/O Automata, where communication occurs along with synchronized execution of statements. I/O Automata, for example, can pass arbitrary parameters from an output statement to all same-named input statements. In CSP, a channel is used to communicate a value from a single sender to a single receiver. In what follows, we examine constructs that allow for communication between synchronized statements. These constructs could be used to build mobile variants of I/O Automata or CSP, the variation being that components may move and disconnect during execution of the model, which may cause a previously synchronized pair of statements to become decoupled.

First we consider I/O Automata [14], where a distributed system is modeled as a collection of state machines. Each machine has a set of input actions, a set of internal actions, and a set of output actions. The execution of an action may modify the state of the machine to which it belongs; in addition, the execution of any output action takes place simultaneously with the execution of all input actions of the same name in all other machines. We can assume that output actions are of a form similar to (1), but with an important addition:

$$\begin{aligned} \text{A.s.driver} :: \; & \langle A.s_{params} := \textbf{exp};\ A.s_{phase} := \text{GO};\ A.s_{phase} := \text{IDLE} \rangle \\ & \text{A.s.action} \parallel A.s_f := \text{false} \quad \textbf{reacts-to}\ A.s_{phase} = \text{GO} \wedge A.s_f \\ & A.s_f := \text{true} \quad \textbf{reacts-to}\ A.s_{phase} = \text{IDLE} \end{aligned} \tag{4}$$

We have added the assignment $A.s_{params} := \textbf{\textit{exp}}$ as the first statement of the transaction. This assignment models binding of the output parameters to a list of auxiliary variables $A.s_{params}$. Here ***exp*** is assumed to be a vector of expressions that may reference other program variables. For example, assume for a moment that the function of the *A.increment* statement from our earlier example is to increment the variable by a value *e* which

is a function of the current state of A. Assume also that *B.increment* must increment $B.y$ by the same amount when the two are co-located. This value could be modeled as a parameter $A.increment_p$ of the synchronization, and *A.increment* would then be of the form (4):

$$\begin{aligned} \text{A.increment.driver} :: \langle \text{increment}_p := e;\ \text{increment}_{\text{phase}} := \text{GO};\ \text{increment}_{\text{phase}} := \text{IDLE} \rangle \\ x := x + e \parallel A.s_f := \text{false} \quad \textbf{reacts-to}\ A.s_{\text{phase}} = \text{GO} \wedge A.s_f \\ A.s_f := \text{true} \quad \textbf{reacts-to}\ A.s_{\text{phase}} = \text{IDLE} \end{aligned}$$

And *B.increment* could also be of this form, with perhaps a different expression e for the increment value. We could then express the sharing of the parameter with the interaction

$$\text{A.increment}_p \approx \text{B.increment}_p \ \textbf{when}\ (A.\lambda = B.\lambda)$$

and the synchronization of the statements with

$$\text{A.increment} \parallel \text{B.increment} \ \textbf{when}\ (A.\lambda = B.\lambda)$$

Thus, either statement may execute its *driver* transaction, which in the first phase assigns a value to the parameter which is propagated to the other component, in the second phase triggers execution of both statements, and in the third resets the flags associated with each statement.

The coordination between output actions and input actions under the I/O Automata model is asymmetric. For instance, parameters are only passed from output action to input action. Also, in our mobile variant, the output action may execute alone if the component containing the input action is currently disconnected, but the input action may never execute alone. For these reasons, we can express the coordination with the following definition, assuming that output statements are of the form (4) and input statements are of the form (1):

$$\begin{aligned} &\text{OI}(A.s, B.s, r) \equiv \\ &\quad B.s_{\text{params}} \leftarrow A.s_{\text{params}} \ \textbf{when}\ r \\ &\quad B.s_{\text{phase}} \leftarrow A.s_{\text{phase}} \quad \textbf{when}\ r \\ &\quad \textbf{inhibit}\ \text{A.s.driver} \quad \textbf{when}\ \neg \text{A.s.guard} \\ &\quad \textbf{inhibit}\ \text{B.s.driver} \end{aligned} \tag{5}$$

The name OI stands for Output/Input, and can be used to specify a pair of statements where the first is an output statement and the second is an input statement. Here the $\leftarrow$ notation denotes a one-way variable sharing where information propagates in the direction of the arrow as in Section 5.1, and the view from the input action is that of a read-only variable. We abuse the notation slightly by applying it to a whole list of variables *params*, but the meaning is simply to share each pair of variables whose positions in the list match. For well-formedness, therefore, the number and types of parameters in the output and input actions must match. The expression *A.s.guard* is assumed to be the enabling predicate or guard on the output action, which should be true exactly when there exists a binding of variables that satisfies the precondition from the output action's textual description. Input actions are always enabled and so we do not need a second predicate. Note

that we must inhibit the input statement's *driver* transaction so that it does not fire alone, but only in response to the output action, and only when the predicate r is true. To fully specify an I/O Automata system, (5) would have to appear in the **Interactions** section numerous times, probably inside a quantifier over pairs of same-named statements, to achieve the one-to-many synchronization offered by the I/O Automata model.

Note that the I/O Automata model does not make the explicit distinction between parameters and action names that we make here, because parameters are thought of as being encoded in the action names themselves. However, we choose to model each textual description of an I/O automaton's output action with one Mobile UNITY statement. Each such textual description in effect determines the values of the parameters as a function of the state of the automaton, by specifying a state predicate that determines when the output action is enabled. Therefore, we are free to make the distinction in Mobile UNITY between statement and parameters.

Coordination under CSP [10] is very similar to I/O Automata, but here we are limited to passing one parameter between a single sender and a single receiver. Also, this model differs from I/O Automata in that the receive actions are not always enabled as is the case with input actions in I/O Automata. For this reason, we choose to block both the sender and the receiver when the two modules are not co-located, and also when one or both of the statements' guards are false. We assume that the receiver statement is of the form (1) and the sender's driver statement is of the form:

$$\text{A.s.driver} :: \langle \text{A.s}_{\text{channel}} := v;\ \text{A.s}_{\text{phase}} := \text{GO};\ \ \text{A.s}_{\text{phase}} := \text{IDLE} \rangle$$

where v represents some expression over the current local state which will be the value transmitted from sender to receiver. We could then express the coordination between statements with the following definition for SR, which stands for Send/Receive:

$$\begin{aligned}
&\text{SR(A.s, B.t, r)} \equiv \\
&\quad \text{B.t}_{\text{channel}} \leftarrow \text{A.s}_{\text{channel}} \ \textbf{when}\ \text{r} \\
&\quad \text{B.t}_{\text{phase}} \leftarrow \text{A.s}_{\text{phase}} \ \ \textbf{when}\ \text{r} \\
&\quad \textbf{inhibit}\ \text{A.s.driver} \ \ \textbf{when}\ \neg(\text{r} \wedge \text{A.s.guard} \wedge \text{B.t.guard}) \\
&\quad \textbf{inhibit}\ \text{B.s.driver}
\end{aligned} \tag{6}$$

Here *A.s.guard* and *B.t.guard* stand for the internal guards on the sender and receiver, respectively. Because we do not wish to explicitly model CSP's flow-of-control inside the UNITY program, we assume that these predicates may be modified at any time during the computation by other statements or by the sending and receiving actions themselves. This models flow of control entering and leaving the send and receive actions of a CSP program.

We can easily make use of the guards on the statements to specify many different and interesting forms of synchronization, just by varying the predicates supplied in the **inhibit** clauses. For example, in addition to the definition of *coselection* defined by (2), we can specify a notion of *coexecution* which has the added meaning that when co-located, the statements may only execute when both guards are enabled. This might be defined as:

coexecute(A.s, B.t, r) ≡

$A.s_{phase} \approx B.t_{phase}$ **when** r

inhibit A.s.driver **when** r ∧ ¬(A.s.guard ∧ B.t.guard)

inhibit B.t.driver **when** r ∧ ¬(A.s.guard ∧ B.t.guard)

which still allows the statements to execute in isolation when not co-located. In contrast, we might require that the statements may not execute in isolation when disconnected. We call this *exclusive coexecution* and it could be specified as

xcoexecute(A.s, B.t, r) ≡

$A.s_{phase} \approx B.t_{phase}$ **when** r

inhibit A.s.driver **when** ¬(r ∧ A.s.guard ∧ B.t.guard)

inhibit B.t.driver **when** ¬(r ∧ A.s.guard ∧ B.t.guard)

A similar notion of *exclusive coselection* could be defined if we ignore the guards on the statements

xcoselect(A.s, B.t, r) ≡

$A.s_{phase} \approx B.t_{phase}$ **when** r

inhibit A.s.driver **when** ¬r

inhibit B.t.driver **when** ¬r

Each of these constructions could be generalized to pass parameters from a sender to a receiver. In fact, the CSP style interaction specified in (6) is very close to exclusive coexecution; the asymmetric nature of the synchronization, however, means that the receiver can never initiate communication. In general, if both driver statements are of the form specified in (4), either statement may bind parameters and propagate them to the other as long as the sharing specified is bi-directional and the driver statement itself is not inhibited. Because the semantics of a transaction mean that it will finish before another is allowed to begin, there is no ambiguity about which statement is currently executing and no conflict in assigning parameters to the synchronized execution. Also, any of the above could be used with statements of the form (3) instead of (1) to provide truly simultaneous access instead of interleaved access to any other shared variables that might be referenced by or assigned to by the various actions.

Our point in examining the many different forms of synchronization is to show the versatility and broad applicability of our model. Because the field of mobile computing is so new, we cannot predict which high-level abstractions will become dominant and gain acceptance in the research community. However, we believe that the examples above show that we can at least handle direct generalizations to the mobile setting of existing mechanisms for synchronous statement execution in models of non-mobile concurrency.

5.3 Partial Synchrony

Yet another way to express synchronization constraints might be the use of **inhibit** as we saw with the sender-receiver example, where actions in one component are inhibited until another component "catches up".

This is usually an abstraction of some set of real-time constraints which we can also use **inhibit** to specify more realistically. We can package this into the **synch** abstraction, shown below. This expresses synchronization of local clocks upon some event, and keeps them synchronized within some drift.

A.t **synch** B.t **when** r
within D ≡
$A.t_B, B.t_A$:= A.t, B.t **reacts-to** r
inhibit A.timer **when** $A.t - A.t_B > D \cdot (B.t - B.t_A)$
inhibit B.timer **when** $B.t - B.t_A > D \cdot (A.t - A.t_B)$

This construct assumes the existence of local variables *A.t* and *B.t* representing local clock values. Whenever some non-reactive statement leaves the system in a state satisfying *r*, the two clocks are synchronized from that point forward within a drift *D*, which is at least 1. Note that neither clock value is actually changed, we simply record the value of each clock in the auxiliary variables $A.t_B$ and $B.t_A$. The closer *D* is to 1, the less the clocks will drift apart. This lets us reason about properties given the relative inaccuracies in the two local clock implementations. The first line of the construct adds a reactive statement that records the value of each local clock in an auxiliary variable. The **inhibit** statements constrain the rate at which these two clocks are incremented to be nearly the same. Once these timers have been synchronized, real-time properties of other statements can be specified by inhibiting a statement until the clock reaches a certain value, which provides a lower bound on execution time, or by inhibiting the local *timer* statement, which provides an upper bound on execution time. These expressions are similar to the *MinTime* and *MaxTime* predicates of [1].

Unlike [1], this mechanism for specifying real-time properties uses only local, discrete clocks instead of a continuous, global one. This contributes to more modular components that are capable of operating in isolation. After prolonged periods of disconnection where presumably the predicate *r* is false, the clocks will have simply drifted far apart. If a global clock were used in the specification this behavior would not be allowed. As with the sender-receiver example, we have a modular and at the same time realistic specification because of the decoupling of components.

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 very 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—indeed others may propose radically different communication abstractions for mobile computing. Our notation was able to express formally all extensions we considered and promises to be a useful research tool for investigating whatever new abstractions may appear. Current plans for future work include the application of Mobile UNITY to formal reasoning about routing algorithms for mobile hosts both in the context of the Internet, i.e. Mobile IP [18], and in the context of *ad-hoc networks* [12], where no fixed routing infrastructure is assumed. 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] D. Dolev, C. Dwork, and L. Stockmeyer, "On the Minimal Synchronism Needed for Distributed Consensus," *Journal of the Association for Computing Machinery*, vol. 34, no. 1, pp. 77-97, 1987.

[6] M. J. Fischer, N. A. Lynch, and M. S. Paterson, "Impossibility of Distributed Consensus with One Faulty Process," *Journal of the Association for Computing Machinery*, vol. 32, no. 2, pp. 374-382, 1985.

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

[8] 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.

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

[10] C. A. R. Hoare, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666-677, 1978.

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

[12] D. B. Johnson, "Routing in Ad Hoc Networks of Mobile Hosts," *Proceedings of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, pp. 158-163, 1994.

[13] A. D. Joseph, A. F. deLespinasse, J. A. Tauber, D. K. Gifford, and M. F. Kaashoek, "Rover: a Toolkit for Mobile Information Access," *Operating Systems Review*, vol. 29, no. 5, pp. 156-71, 1995.

[14] N. A. Lynch and M. R. Tuttle, "An Introduction to Input/Output Automata," *CWI Quaterly*, vol. 2, no. 3, pp. 219-246, 1989.

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

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

[17] 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.

[18] C. Perkins, "IP Mobility Support," Internet Engineering Task Force, ftp://ftp.ietf.cnri.reston.va.us/internet-drafts/draft-ietf-mobileip-protocol-16-txt, Internet draft draft-ietf-mobileip-16, April 22 1996.

[19] G.-C. Roman, J. Y. Plun, and C. D. Wilcox, "Dynamic Synchrony Among Atomic Actions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 6, pp. 677-685, 1993.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

[24] T. Watson, "Application Design for Wireless Computing," *IEEE Workshop on Mobile Computing Systems and Applications*, pp. 91-4, 1994.

Appendix A - List of Constructs

I. Basic Constructs

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.

II. Structuring Conventions

A structured system of mobile components should be expressed in three parts. First should appear a sequence of program definitions, each of which may be parameterized by unbound variables. The **assign** section of each may contain any of the *basic constructs*, but each may reference only variables local to the program in which it appears. Next, the **Components** section is given which instantiates programs from the definitions, binding all unbound variables. Finally, the **Interactions** section consists of a set of constructs which may be either *basic* or *derived*, each of which may reference any variables or statements from the set of instantiated components. The informal system outline given below is intended to illustrate this structure.

System SystemName

program ProgramName(i) **at** λ

declare

always

initially

assign

basic constructs that reference only local variables.

end

Components

⟨[] vars : range :: instances⟩ [] instances

Interactions

Coordination statements, ultimately defined in terms of

basic constructs that may reference variables in any component.

end

III. Derived Constructs

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.