Compositional Programming Abstractions for Mobile Computing

Peter J. McCann, Gruia-Catalin Roman

P. McCann is with Lucent Technologies, Naperville, Illinois. E-mail: mccap@research.bell-labs.com. G.-C. Roman is with the Department of Computer Science, Washington University, St. Louis, Missouri. E-mail: roman@cs.wustl.edu.

Abstract

Recent advances in wireless networking technology and the increasing demand for ubiquitous, mobile connectivity demonstrate the importance of providing reliable systems for managing reconfiguration and disconnection of components. Design of such systems requires tools and techniques appropriate to the task. Many formal models of computation, including UNITY, are not adequate for expressing reconfiguration and disconnection and are therefore inappropriate vehicles for investigating the impact of mobility on the construction of modular and composable systems. Algebraic formalisms such as the π-calculus have been proposed for modeling mobility. This paper addresses the question of whether UNITY, a state-based formalism with a foundation in temporal logic, can be extended to address concurrent, mobile systems. In the process, we examine some new abstractions for communication among mobile components that express reconfiguration and disconnection and which can be composed in a modular fashion.

Keywords

Formal methods, mobile computing, Mobile UNITY, weak consistency, shared variables, synchronization, transient interactions.

I. Introduction

The UNITY [1] approach to concurrency has been influential in the study of distributed systems in large part because of its emphasis on design aspects of the programming process, rather than simply serving as a tool for verification. The technique has been used to derive concurrent algorithms for a wide range of problems, and to specify and verify correctness even in large software systems [2]. However, because of the essentially static structure of computations that can be expressed, standard UNITY is not a suitable tool for addressing the problems faced by the designers of mobile computing systems, such as cellular telephone networks. This paper addresses the problem of modeling dynamically reconfiguring distributed systems with an extension of the UNITY methodology, which we refer to as Mobile UNITY.

While formal models capable of expressing reconfiguration have been explored from the algebraic perspective [3] and from a denotational perspective [4], [5], very few state-based models can naturally express reconfiguration of components. Also, while algebraic models such as the π-calculus may be adequate for expressing reconfiguration, it is not so clear how to handle the issue of disconnection. Recent work has recognized the importance of introducing location and failures as concepts in mobile process algebras [6], [7], [8], but

because the authors are primarily concerned with modeling mobile software agents and the effect of host failures on such systems, they do not directly address disconnection of components that continue to function correctly but independently.

In addition to directly modeling reconfiguration and disconnection, Mobile UNITY attempts to address design issues raised by 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. Broadly speaking, mobile computing leads to systems that are *decoupled* and *context dependent*, and brings new challenges to implementing the illusion of *location-transparency*. By examining the trends in applications and services currently being implemented by system designers, we hope to gain insight into the fundamentals of the new domain and outline opportunities for extensions to models of computation such as UNITY.

Decoupling. 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. 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 filesystems and databases that relax consistency so that disconnected hosts can continue to operate [9], [10].

Decoupling corresponds to the issue of modularity in system design, although in the case of mobility modularity is taken to a new extreme. Because of user demands, components must continue to function even while disconnected from the services used. Also, components must be ready to interface with whatever services are provided at the current location; the notion that a component is statically composed with a fixed set of services must be abandoned. The separation of interfaces from component implementations has long been advocated in the programming language community, but these notions need to be revisited from the more dynamic perspective of mobile computing.

Context Dependencies. 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 [11]. 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 [12], 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.

In Mobile UNITY, although interaction is specified on a pairwise basis and is usually conditioned on the proximity of two components, the model in general can express interaction that is conditioned on arbitrary global predicates, such as the willingness of two components to participate in an interaction, the presence of other components, or the presence of interference or noise on a wireless link. Also, any given collection of pairwise interactions compose naturally to produce compound interactions that may span many components.

Location Transparency. While some systems will be mobile-aware and require explicit reasoning about location and context, other applications naturally make use of location-transparent messaging. For example, Mobile IP [13] attempts to provide this in the context of the Internet. It is illustrative of the mobility management issues that must be addressed by designers. Our previous work [14] modeled Mobile IP in Mobile UNITY. Other location registration schemes have also been dealt with formally, for example with the π-calculus [15] and with standard UNITY [16]. Although the latter work is similar to ours in that it is an application of UNITY to mobility, it deals mainly with the part of the algorithm running in the fixed network and only indirectly with communication between the mobile nodes and the fixed network.

From the perspective of formal modeling, location registration provides a rich source of problems to use as examples. Such mobility management algorithms show that 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. Also, location registration protocols may form the very basis for location- and context-dependent services, which might make use of location information for purposes other than routing.

The remainder of the paper is organized as follows. Section II presents a brief introduction to standard UNITY and the modifications we have made to express context-dependent interactions. The last part of the section gives a proof logic that accommodates the changes. Section III makes use of the new notation to express a new abstraction for communication called transient sharing. Section IV continues by introducing and formally expressing a new communication mechanism called transient synchronization. Concluding remarks are presented in Section V.

II. Mobile UNITY Notation

Our previous work [17] presented a notation and logic for pairwise interactions among mobile components. The pairwise limitation simplified some aspects of the discussion, for instance, side-effects of an assignment were limited to only those components directly interacting with the component containing that assignment. However, the proof logic presented was very complex and operational, sometimes relying on sequencing of operations to define precise semantics. In this paper we present a simpler expression of transient interactions that focuses attention on the implications that component mobility has for the basic atomicity assumptions made by a system model, and provide a proof logic that is much more concise than our earlier work. In the process, we generalize interactions to include multiple participants.

The new model is developed in the context of very low-level wireless communication, in order to focus on the essential details of transient interaction among mobile components. The key concept introduced in this section is the reactive statement, which allows for the modular specification of far-reaching and context-dependent side effects that a statement of one component may have. Using this primitive and a few others as a basis, in subsequent sections we present high-level language constructs that may be specified and reasoned about.

A. Standard UNITY

Recall that UNITY 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. Two example programs, one called *sender* and the other *receiver*, are shown in Figure 1. Program *sender* starts off by introducing the variables it uses 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.

```
program sender
  declare
    bit : boolean
  initially
    bit = 0
  assign
    bit := 0
  [] bit := 1
end
```

```
program receiver
  declare
    bit : boolean
  [] history : sequence of boolean
  initially
    bit = 0
  [] history = ε
  assign
    history := history · bit
end
```

Fig. 1. Two standard UNITY programs.

The execution of program *sender* is a weakly-fair interleaving of its two assignment statements. The assignment statements here are each single assignment statements, but in general they may be multiple-valued, assigning different right-hand expressions to each of several left-hand variables. Such a statement could be written $\vec{x} := \vec{e}$, where $\vec{x}$ is a comma-separated list of variables and $\vec{e}$ is a comma-separated list of expressions. All right-hand side expressions are evaluated in the current state before any assignment to variables is made. Execution of a UNITY program is a nondeterministic but fair infinite interleaving of the assignment statements. Each produces an atomic transformation of the program state and in an infinite execution, each is selected infinitely often. The program *sender*, for example, takes the variable *bit* through an infinite sequence of 1's and 0's. The

next value assigned to *bit* is chosen nondeterministically, but neither value may be forever excluded.

The program *receiver* has two variables, one of which is a sequence of boolean values, but contains only one assignment statement. The statement uses the notation $history \cdot bit$ to denote the sequence resulting from appending *bit* to the end of *history*. This expression is evaluated on the right-hand side and assigned to *history* on the left hand side, essentially growing the history sequence by one bit on each execution. Note that we have not yet introduced the notion of composition, so the two programs should be considered completely separate entities for now.

A.1 Proofs.

Rather than dealing directly with execution sequences, the formal semantics of UNITY are given in terms of program properties that can be proven from the text. The fair interleaving model leads to a natural definition of safety and liveness properties, based on quantification over the set of assignment statements. We choose to use the simplified form of these definitions presented in [18] and [19] for the operators **co** and **transient**. Each operator may be applied to simple non-temporal state predicates constructed from variable names, constants, mathematical operators, and standard boolean connectives. For example, if p and q are state predicates, the safety property p **co** q means that if the program is in a state satisfying p, the very next state after any assignment is executed must satisfy q. Proof of this property involves a universal quantification over all statements s, showing that each will establish q if executed in a state satisfying p.

$$p \ \mathbf{co} \ q \triangleq \langle \forall s :: \{p\}s\{q\} \rangle \ \wedge \ p \Rightarrow q$$

The notation $\{p\}s\{q\}$ is the standard Hoare-triple notation [20]. In addition to the quantification, we are also obligated to show $p \Rightarrow q$, which in effect takes the special statement *skip* (which does nothing) to be part of the quantification. Without this qualification, some cases of **co** would not be true safety properties, because they could be violated by the execution of an action which does nothing.

As an example of **co**, consider the property

$$bit = 0 \ \vee \ bit = 1 \ \mathbf{co} \ bit = 0 \ \vee \ bit = 1.$$

Clearly, this property holds of program *sender*, because every statement, when started in a state where bit is 1 or 0, leaves bit in a state satisfying 1 or 0. This property is an example of a special case of **co** because the left- and right-hand sides are identical. Such a property may be abbreviated with the operator **stable**, as in **stable** $bit = 0 \lor bit = 1$. Because the initial conditions satisfy the predicate, i.e., $bit = 0 \Rightarrow bit = 0 \lor bit = 1$, it is also an invariant, written **invariant** $bit = 0 \lor bit = 1$. When it is not clear from context to which program a property applies, it can be specified explicitly, as in

$$\textbf{invariant } bit = 0 \lor bit = 1 \text{ in } sender.$$

Progress properties can be expressed with the notation **transient** p, which states that the predicate p is eventually falsified. Under UNITY's weak fairness assumption, this can be defined using quantification as

$$\textbf{transient } p \triangleq \langle \exists s :: \{p\} s \{\neg p\} \rangle$$

which denotes the existence of a statement which, when executed in a state satisfying p, produces a state that does not satisfy p. For example, the property **transient** $bit = 0$ can be proven of the program *sender*, because of the statement that sets bit to 1.

The **transient** operator can be used to construct other liveness properties. The reader may be more familiar with the **ensures** operator from UNITY, which is really the conjunction of a safety and a liveness property.

$$p \textbf{ ensures } q \triangleq (p \land \neg q \textbf{ co } p \lor q) \land \textbf{transient } (p \land \neg q)$$

The **ensures** operator expresses the property that if the program is in a state satisfying p, it remains in that state unless q is established, and, in addition, it does not remain forever in a state satisfying p but not q.

While **ensures** can express simple progress properties that are established by a single computational step, proofs of more complicated progress properties often require the use of induction to show that the program moves through a whole sequence of steps in order to achieve some goal. This notion is captured with the *leads-to* operator, written $\mapsto$. Informally, the property $p \mapsto q$ means that if the program is in a state satisfying p, it will

eventually be in a state satisfying q, although it may not happen in only one step and the property p may be falsified in the meantime. For example, consider the property

$$\mathsf{length}(\mathit{history}) = 3 \;\mapsto\; \mathsf{length}(\mathit{history}) = 5 \;\text{ in } \mathit{receiver} \tag{1}$$

which states that if *receiver* is ever in a state where the length of the *history* sequence is 3, it will eventually be in a state where this length is 5. In the meantime, however, the length of the sequence may (and does!) take on the value 4, which satisfies neither side of the relation.

Proofs of *leads-to* properties are carried out inductively, with **ensures** as a base case. Formally, the rules of inference can be summarized as:

(basis) $$\frac{p \textbf{ ensures } q}{p \mapsto q}$$

(transitivity) $$\frac{p \mapsto q,\; q \mapsto r}{p \mapsto r}$$

(disjunction) $$\frac{\langle \forall p : p \in S :: p \mapsto q \rangle}{\langle \exists p : p \in S :: p \rangle \mapsto q}$$ where S is any set of predicates.

The rules are written in hypothesis-conclusion form; each has an assumption above the line, and a deduction below the line. The basis rule, for instance, allows one to conclude $p \mapsto q$ from p **ensures** q. The transitivity rule could be used in a proof of Equation 1, taking p to be the formula $\mathsf{length}(\mathit{history}) = 3$, q to be the formula $\mathsf{length}(\mathit{history}) = 4$, and r to be the formula $\mathsf{length}(\mathit{history}) = 5$. The disjunction rule is useful for breaking up a complicated proof into cases.

The proof rules introduced above come from standard UNITY, but they are also a part of Mobile UNITY. However, the notion of what is a basic state transition is different in the two models, because Mobile UNITY can express the location- and context-dependent state transitions that typify mobile computing. Although this means the basic Hoare triple must be redefined, the rest of the UNITY inference toolkit, including other rules for carrying out high-level reasoning which are not shown here, are preserved.

A.2 Composition.

Before giving the new composition mechanisms of Mobile UNITY, we should first describe the standard UNITY mechanisms for program composition. The most basic composition mechanism is known as program *union*, and we can 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 physical memory; the union of all the assignment statements, which are executed in a fair atomic interleaving; and the intersection of the initial conditions.

Communication between *sender* and *receiver* thus takes place via the shared variable *bit*. The sender writes an infinite sequence of 1's and 0's to this variable, fairly interleaved, and the receiver occasionally reads from this variable to build its *history* sequence. Note that the receiver may not see every value written by the sender, because execution is a fair interleaving, not turn-taking. Also, the resulting *history* generated by the receiver may have duplicate entries because the assignment statements of *sender* may be excluded from execution for a finite amount of time.

Another way to compose systems is through the use of *superposition*, which combines the components by synchronizing statements rather than sharing variables. 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 (1) the maintenance of history variables that do not change the behavior of the underlying program but are needed for certain kinds of proofs, and (2) the construction of layered systems, where the underlying layers are not aware of the higher layer variables.

For example, the receiver, instead of being composed via program union, could have used superposition to synchronize its assignment to *history* with the assignments in the sender that update *bit*, thus ensuring that it would receive an exact history of the values written to *bit*. However, superposition is limited because communication can take place in only one direction. Also, like program union, it is an essentially static form of composition that provides a fixed relationship between the components. It also would require that the single statement in the program *receiver* be broken up into two statements, one for

recording 1's and the other for recording 0's. The challenge of mobile computing is to model the system in a more modular fashion, where the receiver does not know about the internal workings of the sender, and which allows the receiver to be temporarily decoupled from the sender during periods of disconnection. Towards this end we must investigate novel constructs for expressing coordination among the components, so that for instance, the receiver can get an exact history sequence while the components are connected, but may lose information while disconnected.

A major contribution of [1] 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.

B. System Structuring

If the two programs *sender* and *receiver* represent mobile components, or software running on mobile hardware, then it is not appropriate to represent the resulting system as a static composition $sender \ [] \ receiver$. Mobile computing systems exhibit reconfiguration and disconnection of the components, and we would like to capture these essentially new features in our model. Composition with standard UNITY union would share the variable *bit* throughout system execution and would prohibit dynamic reconfiguration and disconnection of the components.

In this section 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. For example, the variable *bit* in the 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 compo-

nents. 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. Figure 2 shows the system *sender-receiver* which embodies these concepts.

System *sender-receiver*
  program *sender* **at** λ
    ...
  end
  program *receiver* **at** λ
    ...
  end
  Components
    receiver **at** λ_0
  [] *sender* **at** λ_0

  Interactions
    ...
end

Fig. 2. Example system notation.

The system starts out by declaring its name, in this case *sender-receiver*. Then, a set of programs are given, each of which is structured like a standard UNITY program, the details of which are elided here. A new feature of these programs is the addition of a program variable λ which stands for the current location of the program. It could have been placed in the **declare** section with the other program variables, but it is promoted here to the same line as the program name to emphasize its importance when reasoning about mobile computations. The precise semantics of the location variable will be discussed in Section II-C.

Assume for now that the internals of each program are as given in Figure 1. In Figure 2, these programs are really type declarations that are instantiated in the **Components** section. In general, the program types may contain free parameters that are bound by

the instantiations; for example, the *receiver* could have been declared as *receiver*(i), and might have been instantiated as *receiver*(1) **at** λ_1. A whole range of *receiver*s could have been instantiated in this way.

The transient interactions among the program instances should be given in the **Interactions** section. Constructs used for specifying interactions are unique to Mobile UNITY and will be introduced in Section II-D. For now we leave the details of this section blank, but it will be developed further to capture the location- and context-dependent aspects of communication between the components.

C. Location

Mobile computing systems must operate under conditions of transient connectivity. Connectivity will depend on the current location of components and therefore location is a part of our model. Just as standard UNITY does not constrain the types of program variables, we do not place restrictions on the type of the location variable λ. It may be discrete or continuous, single or multi-dimensional. 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. For instance, a mobile *receiver* might contain the statement $\lambda := \mathsf{NewLoc}(\lambda)$, where the function NewLoc returns a new location, given the current location. In general, such an assignment could compute a new location based on arbitrary portions of the program state, not just the current location. In a physically moving system, this statement would need to be compiled into a physical effect like actions on motors, for instance. In a mobile code (agents) scenario, this statement would have the effect of migrating an executing program to a new host. 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. Also, λ may still appear on the right-hand side of some assignment statements if there is any location-dependent behavior internal to the program.

D. Interactions

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.

The new constructs presented here, although they are primarily motivated by the need to manage context-dependent coordination of the components in the **Interactions** section, are really orthogonal to the system structuring; one can put these in stand-alone UNITY programs as well. In fact, the formal proof logic abstracts away from the structuring conventions and assumes a flat set of program variables (properly qualified by the name of the program in which they appear) and assignment statements. However, for now, we present each construct and give an example of how each may be used in the **Interactions** section of the system *sender-receiver*.

D.1 Extra Statements.

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.bit* is copied to *receiver.bit* when this is true. We might begin the **Interactions** section with

$$receiver.bit := sender.bit \ \mathbf{when} \ sender.\lambda = receiver.\lambda.$$

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**). The statement

as written copies the value from *sender.bit* to *receiver.bit* when the two programs are at the same location. Here "at the same location" is taken to mean that the programs can communicate, but in general the **when** predicate may take into account arbitrary factors such as the distance between the components or the presence of other components. 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. Therefore, the sender may write several values to *bit* before the extra statement executes once even when the programs are co-located. The receiver may of course move away (by assigning a new value to $receiver.\lambda$ before any value is copied. Also, the construction of *receiver.history* is not necessarily an accurate account of the history of bits written to *receiver.bit*, because the execution of the history-recording action is completely unconstrained with respect to the extra statement and will be interleaved in a fair but arbitrary order.

D.2 Reactions.

A *reactive statement* provides a mechanism for making certain that each and every value written to *sender.bit* also appears at *receiver.bit*. Such a statement would appear in the **Interactions** section as

$$receiver.bit := sender.bit \textbf{ reacts-to } sender.\lambda = receiver.\lambda.$$

Operationally, the reactive statement is scheduled to execute whenever the predicate following **reacts-to** is true. In this sense it is a statement with higher priority than the other statements in the system. In general, there may be other reactive statements implementing other interactions. Informally, all of the reactive statements have equal priority and are executed in an interleaved fashion, much like a standard UNITY program. The set of reactive statements, sometimes denoted with the symbol $\mathcal{R}$, continues to execute until no statement would have an effect if executed. Formally this is known as the *fixed point* of $\mathcal{R}$. Note that this particular statement is idempotent, so if there is no interference from other reactive statements, it reaches fixed point after one execution. In Section II-E we show how this construct can be captured in an axiomatic semantics.

Because this propagation occurs after every step of either component, 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. In general, reactive statements allow for the modeling of side effects that a given non-reactive statement may have when executed in a given context, such as a particular arrangement of components in space.

D.3 Inhibitions.

Note that even with reactive propagation of updates to *sender.bit*, the receiver still will not construct an accurate history of the values that appear on *receiver.bit*. Because of the nondeterministic interleaving of statements, several values may be written to *receiver.bit* between executions of the statement that updates *receiver.history*. In a real wireless communication system, closely synchronized clocks and timing considerations would ensure that values are read at the proper moment so as not to omit or duplicate any bits in the sequence.

An *inhibitor* provides a mechanism for constraining UNITY's nondeterministic scheduler when execution of some statement would be undesirable in a certain global context. Adding a label to a statement lets us express inhibition in a modular way, without modifying the original statement. For example, consider a new *sender* program, given in Figure 3. The

program *sender* **at** λ
  declare
    bit : boolean
  [] *counter* : integer
  initially
    $bit = 0$
  [] $counter = 0$
  assign
    $s0 :: bit, counter := 0, counter + 1$
  [] $s1 :: bit, counter := 1, counter + 1$
end

Fig. 3. A new version of program *sender* that counts bits.

two statements each now carry a label; we can refer to the first as *sender.s0* and the second

as *sender.s1*. Each also updates the integer *counter* to reflect the number of bits written so far. This counter serves as an abstraction for a real-time clock, virtual or actual, that may be running on program *sender*.

If we assume that the statement of program *receiver* is labeled *read*, for example,

$$read :: history := history \cdot bit$$

then we might add the following set of clauses to the **Interactions** section:

$$\begin{array}{llll} & \textbf{inhibit}\ sender.s0 & \textbf{when} & sender.counter > \mathsf{length}(receiver.history) \\ [] & \textbf{inhibit}\ sender.s1 & \textbf{when} & sender.counter > \mathsf{length}(receiver.history) \\ [] & \textbf{inhibit}\ receiver.read & \textbf{when} & \mathsf{length}(receiver.history) \geq sender.counter. \end{array}$$

The net effect of **inhibit** s **when** p is a strengthening of the guard on statement s by conjoining it with $\neg p$ and thus inhibiting execution of the statement when p is true. The inhibitions given above constrain execution so that the receiver reads exactly one bit for every bit written by the sender. Note that the constraint applies equally well when the components are disconnected; this is not unrealistic because we can assume that realtime clocks can remain roughly synchronized even after disconnection, even though the reactive propagation of values will cease.

Reactive statements must not be inhibited.

D.4 Transactions.

With reactive propagation and inhibitions as given above, execution of the system will append values to *receiver.history* even when the two components are disconnected. Thus there will be subsequences of *receiver.history* containing redundant copies of the last value written by the *sender*. In an actual wireless transmission system, the receiver does have some indication of receipt of a transmission, and would not build a history that depended only on timing constraints. In our model, this might be represented as an extra third state that can be taken on by the wireless transmission medium. Assume that the *bit* in each component was therefore declared:

$$bit : \mathsf{boolean} \cup \{\perp\}$$

and initialized to $\perp$, which means simply that no transmission is currently taking place.

Transmission of a bit by the sender then involves placing a value on the communications medium, and then returning it to a quiescent state. The receiver may then reactively record the value written. A *transaction* provides a form of sequential execution, and can be used by the statements in *sender* that write new values to *sender.bit*:

$$s0 :: \langle bit := 0;\ bit := \perp \rangle$$
$$s1 :: \langle bit := 1;\ bit := \perp \rangle.$$

A transaction consists of a sequence of assignment statements, enclosed in angle brackets and separated by semicolons, which must be scheduled in the specified order with no other nonreactive statements interleaved in between. The assignment statements of standard UNITY may be viewed as singleton transactions. Note that reactive statements are allowed to execute to fixed point at each semicolon and at the end of the transaction; this lets us write a new *receiver* program, shown in Figure 4. Here the first reactive statement

program *receiver* **at** λ
  declare
    bit : boolean $\cup \{\perp\}$ [] $flag$: boolean
  [] $history$: sequence of boolean
  initially
    $bit = 0$ [] $flag = 0$
  [] $history = \epsilon$
  assign
    $history, flag := history \cdot bit, 1$ **reacts-to** $bit \neq \perp \wedge \neg flag$
  [] $flag := 0$ **reacts-to** $bit = \perp$
end

Fig. 4. A new version of program *receiver* that reacts to transactions.

records values written to the shared *bit*, and the variable *flag* is added to make the reactive recording idempotent. Another reactive statement is added to reset *flag* when the communications medium returns to a quiescent state.

Transactions may be inhibited, but may not be reactive.

E. Proof Logic

Now we give a logic for proving properties of programs that use the above constructs. The execution model has assumed that each non-reactive statement is fairly selected for execution, is executed if not inhibited, and then the set of reactive statements, denoted $\mathcal{R}$, is allowed to execute until it reaches fixed point, 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 the model and we denote them by s^*, for each non-reactive statement s. We denote the set of non-reactive statements (including transactions) by $\mathcal{N}$. Thus, the definitions for basic **co** and **transient** properties become:

$$p \ \mathbf{co} \ q \triangleq \langle \forall s \in \mathcal{N} :: \{p\}s^*\{q\}\rangle \ \wedge \ p \Rightarrow q$$

and

$$\mathbf{transient} \ p \triangleq \langle \exists s \in \mathcal{N} :: \{p\}s^*\{\neg p\}\rangle.$$

Even though s^* is really a possibly inhibited 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.

We first deal with statement inhibition. The following rule holds for non-reactive statements s, whether they are transactions or singleton statements:

$$\frac{p \wedge i(s) \Rightarrow q, \ \{p \wedge \neg i(s)\}s^{\mathcal{R}}\{q\}}{\{p\}s^*\{q\}} \tag{2}$$

We define $i(s)$ 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. The notation $s^{\mathcal{R}}$ denotes the statement s extended by execution of the reactive statement set $\mathcal{R}$. For singleton, non-transactional statements, $\{r\}s^{\mathcal{R}}\{q\}$ can be deduced from

$$\frac{\{r\}s\{H\}, \ H \mapsto (FP(\mathcal{R}) \wedge q) \text{ in } \mathcal{R}}{\{r\}s^{\mathcal{R}}\{q\}} \tag{3}$$

where H may be computed as the strongest postcondition of r with respect to s, or guessed at as appropriate. We take $\{r\}s\{H\}$ from the hypothesis to be a standard Hoare triple for

the non-augmented statement s. The notation $FP(\mathcal{R})$ denotes the fixed-point predicate of the set of reactive statements, which can be determined from its text. The "in $\mathcal{R}$" must be added because the proof of termination is to be carried out from the text of the reactive statements, ignoring other statements in the system. This can be accomplished with a variety of standard UNITY techniques.

For statements that consist of multiple steps in a transaction, we have the rule

$$\frac{\{r\}\langle s_1; s_2; ...; s_{n-1}\rangle^{\mathcal{R}}\{w\},\ \{w\}s_n^{\mathcal{R}}\{q\}}{\{r\}\langle s_1; s_2; ...; s_n\rangle^{\mathcal{R}}\{q\}} \tag{4}$$

where w may be guessed at or derived from r and q as appropriate. This represents sequential composition of a reactively-augmented prefix of the transaction with its last sub-action. Then Equation 3 can be applied as a base case. 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 notation and proof logic presented above provide tools for reasoning about concurrent, mobile systems. Apart from the redefinition of the basic notion of atomic transitions, 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 programming notation can be used to construct systems of mobile components that exhibit much more dynamic behavior than could be easily expressed with standard UNITY.

III. Transient Sharing

In the previous sections, we presented a notation and logic for reasoning about systems of mobile components. In this section and in Section IV, we attempt to build higher level abstractions out of those low-level primitives that will contribute to the design of systems that are decoupled and context-sensitive. To be successful, such abstractions should be familiar to designers, should take into account the realities of mobile computing, should be implementable, and should have a strong underlying formal foundation. An obvious starting point is the inter-program communication mechanisms from standard UNITY, namely shared variables and statement synchronization. This section examines a variant of sharing suited to mobile computing systems and gives an underlying semantics in terms of the notation we have already developed.

In the mobile setting, variables from two independently moving programs are not always connected, and this is reflected in the model by the isolation of each of the namespaces, as was the case with *sender.bit* and *receiver.bit* from our earlier example. However, with the addition of a reactive propagation statement to the **Interactions** section, these two variables took on some of the qualities of a shared variable. While the two components were at the same location, any value written by the *sender* was immediately visible to the *receiver*. The semantics of reactive statements guarantee that such propagation happens in the same atomic step as the statement *sender.s0* or *sender.s1*.

Sharing may also be an appropriate abstraction for communication at a coarser granularity; for example, one might think of two mobile hosts as communicating via a (virtual) shared packet, instead of a single shared bit. This is realistic because of the lower level protocols, such as exponential back-off, that are providing serialized access to the communications medium. At an even coarser (more abstract) level, there might be data structures that are replicated on each host, access to which is serialized by a distributed algorithm implementing mutual exclusion. Of course, no such algorithm can continue to guarantee both mutual exclusion and progress in the presence of disconnection, but our (so far informal) notion of a transiently shared variable does not require consistency when disconnected.

In what follows, we package these notions into a coordination construct that can be formally specified and reasoned about. As a running example, we consider a queue of documents to be output on a printer. Assume that a laptop computer, connected by some wireless communication medium, is wandering in and out of range of the printer, so it maintains a local cache of this queue. When the laptop is in range of the printer, updates to the queue are atomically propagated, expressed as a transient sharing of the queue. This may be denoted by the expression

$$laptop.q \approx printer.q \ \textbf{when} \ laptop.\lambda = printer.\lambda$$

The operations on the queue 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.

The $\approx$ relationship can be defined formally in terms of reactive statements that propa-

gate changes. Because the sharing is bidirectional, there is slightly more complexity than the earlier example where a single reactive statement could propagate values in one direction. In the present case, we need a mechanism for detecting changes and selectively propagating only new values. Therefore we add additional variables to each program that model the previous state of the queue. In program *laptop*, this variable is called $q_{printer.q}$, and in program *printer*, this variable is called $q_{laptop.q}$. The reactive statements that detect and propagate changes are

$$printer.q,\ printer.q_{laptop.q},\ laptop.q_{printer.q} := laptop.q,\ laptop.q,\ laptop.q$$
$$\textbf{reacts-to}\ \ laptop.q \neq laptop.q_{printer.q} \wedge laptop.\lambda = printer.\lambda$$
$$laptop.q,\ laptop.q_{printer.q},\ printer.q_{laptop.q} := printer.q,\ printer.q,\ printer.q$$
$$\textbf{reacts-to}\ \ printer.q \neq printer.q_{laptop.q} \wedge printer.\lambda = laptop.\lambda$$

which execute when any history variable is different from the current value of the variable which it is tracking, when the components are connected. Each statement updates both history variables as well as the remote copy of the queue. This can be thought of as “echo cancellation,” in that the remote copy is kept the same as its history variable, and the reverse reaction is kept disabled. In addition, we add statements that simply update the history variables, without propagating values, when the components are disconnected:

$$laptop.q_{printer.q} := laptop.q$$
$$\textbf{reacts-to}\ \ laptop.\lambda \neq printer.\lambda$$
$$printer.q_{laptop.q} := printer.q$$
$$\textbf{reacts-to}\ \ printer.\lambda \neq laptop.\lambda$$

These statements reflect the fact that because disconnection may take place at any moment, one component cannot know that its change actually did propagate to the remote component and so the local behavior (update of the history variable) must be exactly the same in both cases.

Although the reactions given above may meet our informal expectations for a shared variable while connection is continuous, there are some subtle issues that arise when disconnection and reconnection are allowed. For instance, when disconnection takes place, the laptop and printer each have separate identical copies of the queue. If changes are made independently, for instance, if the laptop adds a few items and the printer deletes a

few items, an inconsistent state arises which may present a problem upon re-connection. The semantics given above are well defined for this case: whichever component makes the first assignment to the reconnected queue will have its copy propagated to the other component. This may be undesirable; documents which have already been printed may be re-inserted into the queue, or documents which have been added by the laptop while disconnected may be lost.

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 [9] and [10] 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. For example, the print queue example may be extended with the following notation:

$$\begin{array}{lll} laptop.q \approx printer.q & \textbf{when} & laptop.\lambda = printer.\lambda \\ & \textbf{engage} & laptop.q \cdot printer.q \\ & \textbf{disengage} & \epsilon,\ printer.q \end{array}$$

The **engage** value specifies that upon reconnection, the shared queue should take on the value constructed by appending *printer.q* to *laptop.q*. The **disengage** construct contains two values; the first is assigned to *laptop.q* and the second is assigned to *printer.q* upon disconnection. The values given empty *laptop.q* and leave the *printer.q* untouched. This is justified because the queue would realistically reside on the printer during periods of disconnection and the laptop would have no access to it. However, any documents appended to the queue by the laptop are appended to the print queue upon reconnection.

Formally, the above construct would translate into reactions that take place when a change in the connection status is detected. Again we add an auxiliary history variable, this time to record the status of the connection, denoted by $status_{laptop.q,printer.q}$. For engagement, we add

$$laptop.q,\ printer.q,\ status_{laptop.q,printer.q} := \\ laptop.q \cdot printer.q,\ laptop.q \cdot printer.q,\ true \\ \textbf{reacts-to} \quad \neg\ status_{laptop.q,printer.q} \wedge printer.\lambda = laptop.\lambda$$

which integrates both values when the connection status changes from down to up. For disengagement, we add

$$laptop.q,\ printer.q,\ status_{laptop.q,printer.q} := \epsilon,\ printer.q,\ false \\ \textbf{reacts-to} \quad status_{laptop.q,printer.q} \wedge printer.\lambda \neq laptop.\lambda$$

which assigns different values to each variable when the connection status changes from up to down. In the absence of interference, each of these statements executes once and is then disabled.

Systems like [9] and [10] 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 [9], which can be carried out as a user prepares to take his laptop home at the end of a workday, for instance.

Predictable disconnection is not possible 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 [21], providing **engage** and **disengage** semantics in these settings is possible only in a probabilistic sense. This may be adequate; consider, for instance, the phenomenon of metastable states [22]. Almost every computing device in use today is subject to some probability of failure due to metastability, but the probability is so low that it is almost never considered in reasoning about these systems. A similarly robust implementation of **engage** and **disengage** may be possible. However, the basic semantics of $\approx$ do not imply distributed consensus and are in fact implementable.

Construct	*Description*	*Definition*
$A.x \rightarrow B.y$ **when** p	Read-only transient sharing ($A.x$ is read by $B.y$)	$B.y,\ B.y_{A.x},\ A.x_{B.y} := A.x,\ A.x,\ A.x$ **reacts-to** $A.x \neq A.x_{B.y} \wedge p$ $A.x_{B.y} := A.x$ **reacts-to** $\neg p$
$A.x \approx B.y$ **when** p	Read-write transient sharing	$A.x \rightarrow B.y$ **when** p $B.y \rightarrow A.x$ **when** p
engage$(A.x,\ B.y)$ **when** p **value** e	Engagement[1]	$A.x,\ B.y,\ status_{A.x,B.y} := e,\ e,\ true$ **reacts-to** $\neg\ status_{A.x,B.y} \wedge p$
disengage$(A.x,\ B.y)$ **when** p **value** d_1, d_2	Disengagement[2]	$A.x,\ B.y,\ status_{A.x,B.y} := d_1,\ d_2,\ false$ **reacts-to** $status_{A.x,B.y} \wedge \neg p$

TABLE I

TRANSIENT SHARING NOTATIONAL CONSTRUCTS.

The transient sharing construct given above is a relationship between two variables, but it is compositional in a very natural way. For instance, suppose we would like to distribute the print jobs among two different printers. This could be accomplished by simply adding another sharing relationship of the form

$$printer.q \approx printer2.q \textbf{ when } true$$

which specifies that the queue should be shared with *printer2* always. Each printer would have atomic access to this shared queue and could remove items from the head as they are printed. Because all **reacts-to** statements are executed until fixed point, any change to one of the three variables is propagated to the other two, when the laptop is co-located with the printer. This transitivity is a major factor contributing to the construction of modular systems, as it allows the statement of one component to have far-reaching implicit effects that are not specified explicitly in the program code for that component.

A summary of the notation developed in this section appears in Table I, which also

breaks up the $\approx$ construct into two uni-directional sharing relationships. In general these can be combined in arbitrary ways, but remember that any proofs of correctness require proof that $\mathcal{R}$ terminates, so not every construction will be correct. For example, if there are any cycles in the sharing relationship, and if two different variables on a given cycle are set to distinct values in the same assignment statement, then it is not possible to prove termination. This is analagous to UNITY's restriction that each statement assigns a unique value to each left-hand variable. Also, termination of $\mathcal{R}$ may be difficult to prove if some engagement or disengagement values cause other **when** predicates to change value.

The transient sharing abstraction presented here has shown promise as a way to manage the complexity of concurrent, mobile systems. Based on a familiar programming paradigm, that of shared memory, it provides a mechanism for expressing highly decoupled and context-dependent systems. The abstraction is apparently a good one for low-level wireless communication, and mutual exclusion protocols that implement the abstraction at a coarser level of granularity may be simple generalizations of existing replication, transaction, or consistency algorithms. This section presented a formal definition for the concept that facilitates reasoning about systems that make use of it.

IV. Transient Synchronization

The previous section 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 [23] 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 [24] expresses communication

[1] If engagement is used without a corresponding disengagement, an extra reaction must be added to reset $status_{A.x,B.y}$ to *false* when p becomes false.

[2] Similarly, if disengagement is used without a corresponding engagement, an extra reaction must be added to set $status_{A.x,B.y}$ to *true* when p becomes true.

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. Synchronous composition can also be used in the refinement process [25], although our emphasis here is on composition of mobile programs rather than refinement.

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. 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 [26]. While the latter is a very important component of our current understanding of distributed systems, and in many circumstances is perhaps a prerequisite to the implementation of the former, it is not our focus here. Rather, we examine mechanisms that allow us to compose programs and to combine a group of statements into a new one through parallel execution. This idea of statement co-execution was inspired by UNITY superposition.

However, 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 the 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 [27]. 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 communicate only 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:

$$A.increment \rrbracket\!\llbracket\; B.increment\ \textbf{when}\ (A.\lambda = B.\lambda)$$

Note that this does not prohibit the statements from executing independently when the programs are not co-located. If the correctness criteria state that the counters must

remain synchronized at all times, we could add the following two **inhibit** clauses to the **Interactions** section:

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

As distinct from standard UNITY superposition, the $\rceil\lceil$ 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:

$$\begin{aligned} A.s.driver :: \langle A.s_{phase} := \mathrm{GO}; A.s_{phase} := \mathrm{IDLE}\rangle \\ A.s.action \| A.s_f := \mathit{false} \quad \textbf{reacts-to} \quad A.s_{phase} = \mathrm{GO} \wedge A.s_f \\ A.s_f := \mathit{true} \quad \textbf{reacts-to} \quad A.s_{phase} = \mathrm{IDLE} \end{aligned} \tag{5}$$

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 given in Equation 5, we can define $\rceil\lceil$ as:

$$A.s \rceil\lceil B.t\ \textbf{when}\ r \triangleq A.s_{phase} \approx B.t_{phase}\ \textbf{when}\ r \tag{6}$$

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 Equation 6.

The semantics of Equations 5 and 6 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 $\mathcal{R}$. 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 Equation 5 which models the evaluation of right-hand sides as a separate step from assignment to left-hand variables:

$$\begin{aligned} A.s.driver :: \langle A.s_{phase} := \text{LOAD}; A.s_{phase} := \text{STORE}; A.s_{phase} := \text{IDLE}\rangle \\ A.s.load \| A.s_{lf} := \mathit{false} \quad \textbf{reacts-to} \quad A.s_{phase} = \text{LOAD} \wedge A.s_{lf} \\ A.s.store \| A.s_{sf} := \mathit{false} \quad \textbf{reacts-to} \quad A.s_{phase} = \text{STORE} \wedge A.s_{sf} \\ A.s_{lf}, A.s_{sf} := \mathit{true}, \mathit{true} \quad \textbf{reacts-to} \quad A.s_{phase} = \text{IDLE} \end{aligned} \tag{7}$$

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 Equation 6, 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, we return to the increment example and consider the following set of interactions:

$$
\begin{aligned}
&A.\mathit{increment} \;\rceil\lceil\; B.\mathit{increment} \;\; \textbf{when}\; (A.\lambda = B.\lambda) \\
&A.x \approx C.z \;\; \textbf{when}\; r \\
&C.z \approx B.y \;\; \textbf{when}\; r
\end{aligned}
$$

Here the statements $A.\mathit{increment}$ and $B.\mathit{increment}$ 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 given in Equation 5, then changes to one variable may inadvertently be 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 given in Equation 7 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. In CSP, a channel is used to communicate a value from a single sender to a single receiver. I/O automata can pass arbitrary parameters from an output statement to all same-named input statements. As an example, we will now give a construction that expresses I/O Automata-style synchronization. Recall that each automaton has a set of input actions, a set of internal actions, and a set of output actions. The execution of any 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 Equation 5, but with an important addition:

$$\begin{aligned} A.s.driver :: \langle A.s_{params} := \mathbf{exp}; A.s_{phase} := \mathrm{GO}; A.s_{phase} := \mathrm{IDLE}\rangle \\ A.s.action \| A.s_f := false \quad \textbf{reacts-to} \quad A.s_{phase} = \mathrm{GO} \wedge A.s_f \\ A.s_f := true \quad \textbf{reacts-to} \quad A.s_{phase} = \mathrm{IDLE} \end{aligned} \tag{8}$$

We have added the assignment $A.s_{params} := \mathbf{exp}$ 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 the 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 given in Equation 8:

$$\begin{aligned} A.increment.driver :: \quad & \langle increment_p := e; \\ & increment_{phase} := \mathrm{GO}; \\ & increment_{phase} := \mathrm{IDLE}\rangle \\ x := x + increment_p \| A.s_f := false \quad & \textbf{reacts-to} \quad A.s_{phase} = \mathrm{GO} \wedge A.s_f \\ A.s_f := true \quad & \textbf{reacts-to} \quad A.s_{phase} = \mathrm{IDLE} \end{aligned} \tag{9}$$

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

$$A.increment_p \approx B.increment_p \ \mathbf{when} \ (A.\lambda = B.\lambda)$$

and the synchronization of the statements with

$$A.increment \rceil\lceil B.increment \ \mathbf{when} \ (A.\lambda = B.\lambda)$$

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.

We can easily make use of the guards on the statements to specify many different and interesting forms of synchronization with the use of appropriately tailored **inhibit** clauses. For example, in addition to the definition of coselection defined by Equations 5 and 6, 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

$$\begin{aligned}
&\text{coexecute}(A.s,\ B.t,\ r) \triangleq \\
&\qquad A.s_{phase} \approx B.t_{phase}\ \ \textbf{when}\ r \\
&\qquad \textbf{inhibit}\ A.s.driver\ \ \textbf{when}\ r \wedge \neg(A.s.guard \wedge B.t.guard) \\
&\qquad \textbf{inhibit}\ B.t.driver\ \ \textbf{when}\ r \wedge \neg(A.s.guard \wedge B.t.guard)
\end{aligned}$$

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

$$\begin{aligned}
&\text{xcoexecute}(A.s,\ B.t,\ r) \triangleq \\
&\qquad A.s_{phase} \approx B.t_{phase}\ \ \textbf{when}\ r \\
&\qquad \textbf{inhibit}\ A.s.driver\ \ \textbf{when}\ \neg(r \wedge A.s.guard \wedge B.t.guard) \\
&\qquad \textbf{inhibit}\ B.t.driver\ \ \textbf{when}\ \neg(r \wedge A.s.guard \wedge B.t.guard)
\end{aligned}$$

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

$$\begin{aligned}
&\text{xcoselect}(A.s,\ B.t,\ r) \triangleq \\
&\qquad A.s_{phase} \approx B.t_{phase}\ \ \textbf{when}\ r \\
&\qquad \textbf{inhibit}\ A.s.driver\ \ \textbf{when}\ \neg r \\
&\qquad \textbf{inhibit}\ B.t.driver\ \ \textbf{when}\ \neg r
\end{aligned}$$

Each of these constructions could be generalized to pass parameters from a sender to a receiver. In general, if both driver statements are of the form specified in Equation 8, 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 in Equation 7 instead of Equation 5 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.

In contrast to the symmetric forms of synchronization considered so far, the coordination between actions in models such as I/O Automata and CSP is asymmetric. In each model, actions are divided into input and output classes; parameters are passed from output actions to input actions. The two models differ in the number of participants in a synchronization. I/O Automata are capable of expressing one-to-many synchronization styles, while CSP emphasizes pairwise rendezvous of output actions with input actions. All of these synchronization styles can be expressed in Mobile UNITY with appropriate use of transactions, variable sharing, and inhibitions. For example, one-to-many synchronization with parameter passing can be simulated by a quantified set of one-way variable sharing relationships, where the output statement executes as a transaction and the input statements are simply reactive. For rendezvous style synchronization, the *phase* variable must be propagated to at most one input action, which can be ensured by a flag which is set by the first (nondeterministically chosen) reactive statement and which prevents other propagations from taking place. Of course, other aspects of CSP, such as the dynamic creation and deletion of terms, could not be so easily captured in a UNITY-style model because of fundamental differences in the underlying approaches. Similarly, the discussion of I/O Automata has assumed that it is acceptable to model a whole set of IOA actions with one parameterized UNITY action, which may not be appropriate in every case. Even so, modeling the basic synchronization mechanisms of the two models in Mobile UNITY can be a useful exercise.

Our point in examining the many different forms of synchronization is to show the versatility and broad applicability of the 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 Mobile UNITY can at least formalize direct generalizations to the mobile setting of existing mechanisms for synchronous statement execution in models of non-mobile con-

currency, and there is good reason to believe it is capable of expressing new constructs that may be proposed in the future.

V. Discussion

The Mobile UNITY notation and logic presented in this paper is the result of a careful reevaluation of the implications of mobility on UNITY. 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. Previous work extended the UNITY proof logic to handle pairwise forms of such interaction. This paper presented a more modular and compositional construction of transient sharing and synchronization that allowed for multi-party interactions among components.

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. The notation was able to express formally all extensions that were considered and promises to be a useful research tool for investigating whatever new abstractions may appear. Plans for future work include the application of Mobile UNITY to distributed databases with weak consistency semantics, capable of continuing operation in the presence of disconnection. These problems only recently have 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.

VI. Acknowledgements

This paper is based upon work supported in part by the National Science Foundation of the United States under Grant Numbers CCR-9217751 and CCR-9624815. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] K. Mani Chandy and Jayadev Misra, *Parallel Program Design: A Foundation*, Addison-Wesley, New York, NY, 1988.

[2] Mark G. Staskauskas, "Formal derivation of concurrent programs: An example from industry," *IEEE Transactions on Software Engineering*, vol. 19, no. 5, pp. 503–28, May 1993.

[3] Robin Milner, Joachim Parrow, and David Walker, "A calculus of mobile processes. I," *Information and Computation*, vol. 100, no. 1, pp. 1–40, 1992.

[4] William D. Clinger, "Foundations of actor semantics," Tech. Rep. AI-TR-633, MIT Artificial Intelligence Laboratory, 1981.

[5] Gul Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*, MIT Press, Cambridge, Massachusetts, 1986.

[6] Roberto M. Amadio, "An asynchronous model of locality, failure, and process mobility," in *Coordination Languages and Models*, Berlin, 1997, pp. 374–91, Springer-Verlag.

[7] James Riely and Matthew Hennessy, "Distributed processes and location failures," Tech. Rep. 2/97, Computer Science, School of Cognitive and Computing Sciences, University of Sussex, Brighton, England, 1997.

[8] Cédric Fournet, Georges Gonthier, Jean-Jacques Lévy, Luc Maranget, and Didier Rémy, "A calculus of mobile agents," in *Proceedings of the International Conference on Concurrency Theory*. 1996, vol. 1119 of *Lecture Notes in Computer Science*, pp. 406–421, Springer-Verlag.

[9] M. Satyanarayanan, James J. Kistler, Lily B. Mummert, Maria R. Ebling, Puneet Kumar, and Qi Lu, "Experience with disconnected operation in a mobile computing environment," in *Proceedings of the USENIX Symposium on Mobile and Location-Indepedent Computing*, Cambridge, MA, 1993, pp. 11–28.

[10] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser, "Managing update conflicts in Bayou, a weakly connected replicated storage system," *Operating Systems Review*, vol. 29, no. 5, pp. 172–83, 1995.

[11] Geoffrey M. Voelker and Brian N. Bershad, "Mobisaic: An information system for a mobile wireless computing environment," in *Proceedings of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, 1994, pp. 185–90, IEEE.

[12] Bill N. Schilit, Norman Adams, and Roy Want, "Context-aware computing applications," in *Proceedings of the Workshop on Mobile Computing Systems and Applications*, Santa Cruz, CA, 1994, pp. 85–90, IEEE.

[13] Charles Perkins, "IP mobility support," RFC 2002, IETF Network Working Group, 1996.

[14] Peter J. McCann and Gruia-Catalin Roman, "Mobile UNITY coordination constructs applied to packet forwarding for mobile hosts," in *Coordination Languages and Models*, Berlin, 1997, vol. 1282 of *Lecture Notes in Computer Science*, pp. 338–54, Springer-Verlag.

[15] Fredrik Orava and Joachim Parrow, “An algebraic verification of a mobile network,” R91:02, Swedish Institute of Computer Science, 1991.

[16] Beverly Sanders, Berna Massingill, and Svetlana Kryukova, “Specification and proof of an algorithm for location management for mobile communication devices,” in *Proceedings of the International Workshop on Formal Methods for Parallel Programming: Theory and Applications*, Geneva, April 1997, IPPS ’97.

[17] Gruia-Catalin Roman, Peter J. McCann, and Jerome Y. Plun, “Mobile UNITY: Reasoning and specification in mobile computing,” *ACM Transactions on Software Engineering and Methodology*, vol. 6, no. 3, pp. 250–82, 1997.

[18] Jayadev Misra, “A logic for concurrent programming: Safety,” *Journal of Computer and Software Engineering*, vol. 3, no. 2, pp. 239–72, 1995.

[19] Jayadev Misra, “A logic for concurrent programming: Progress,” *Journal of Computer and Software Engineering*, vol. 3, no. 2, pp. 273–300, 1995.

[20] C.A.R. Hoare, “An axiomatic basis for computer programming,” *Communications of the ACM*, vol. 12, no. 10, pp. 576–580,583, 1969.

[21] Michael J. Fischer, Nancy A. Lynch, and Michael 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.

[22] Tom J. Chaney and Charles E. Molnar, “Anomalous behavior of synchronizer and arbiter circuits,” *IEEE Transactions on Computers*, vol. C-22, no. 4, pp. 421–422, 1973.

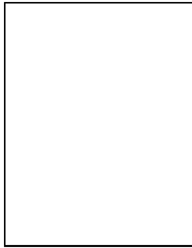
[23] C.A.R. Hoare, “Communicating sequential processes,” *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, 1978.

[24] Nancy A. Lynch and Mark R. Tuttle, “An introduction to input/output automata,” *CWI Quaterly*, vol. 2, no. 3, pp. 219–246, 1989.

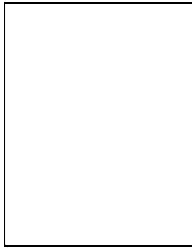
[25] Reino Kurki-Suonio, “Fundamentals of object-oriented specification and modeling of collective behaviors,” in *Object-Oriented Behavioral Specifications*, Haim Kilov and William Harvey, Eds., pp. 101–120. Kluwer Academic Publishers, 1996.

[26] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer, “On the minimal synchronism needed for distributed consensus,” *Journal of the Association for Computing Machinery*, vol. 34, no. 1, pp. 77–97, 1987.

[27] Gruia-Catalin Roman, Jerome Y. Plun, and C. Donald Wilcox, “Dynamic synchrony among atomic actions,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 4, no. 6, pp. 677–685, 1993.



Peter J. McCann received his B.S. degree (1993) in engineering and applied science from the California Institute of Technology, and the M.S. degree (1995) and D.Sc. degree (1997) in computer science from Washington University in St. Louis, Missouri. He is currently a Member of Technical Staff at Bell Laboratories. His research interests include formal reasoning about concurrent, mobile systems and programming abstractions for weak-consistency distributed systems.



Gruia-Catalin Roman was a Fulbright Scholar at the University of Pennsylvania, in Philadelphia, where he received a B.S. degree (1973), an M.S. degree (1974), and a Ph.D. degree (1976), all in computer science. He has been on the faculty of the Department of Computer Science at Washington University in Saint Louis since 1976. Roman is a professor and chairman of the department. His current research involves the study of formal models and design methods for mobile computing and the development of techniques for the visualization of distributed computations. His previous research has been concerned with models of concurrency, declarative visualization methods, design methodologies, systems requirements, interactive computer vision algorithms, formal languages, biomedical simulation, computer graphics, and distributed databases. Roman is also an active software engineering consultant. His consulting work involves development of custom software engineering methodologies and training programs.