Report Number: WUCS-95-26

1995-01-01

Assertional Reasoning about Pairwise Transient Interactions in Mobile Computing

Authors: Gruia-Catalin Roman, Peter J. McCann, and Jerome Plun

Mobile computing represents a major point of departure from the traditional distributed computing paradigm. The potentially very large number of independent computing units, a decoupled computing style, frequent disconnections, continuous position changes, and the location-dependent nature of the behavior and communication patterns of the individual components present designers with unprecedented challenges in the areas of modularity and dependability. This paper describes two ideas regarding a modular approach to specifying and reasoning about mobile computing. The novelty of our approach rests with the notion of allowing transient interactions among programs which mobe in space. In this paper we restrict our concert to pariwise interactions involving variable sharing and action asynchronization. The motivation behind the transient nature of the interactions comes from the fact that components can communicate with each other only when they are within a certain range. The notation we propose is meant to simplify the writing of mobile applciaitons and is a direct extension of that used in UNITY. Reasoning about mobile bomputations relies on the UNITY proof logic.
... **Read complete abstract on page 2.**

Assertional Reasoning about Pairwise Transient Interactions in Mobile Computing

Complete Abstract:

Mobile computing represents a major point of departure from the traditional distributed computing paradigm. The potentially very large number of independent computing units, a decoupled computing style, frequent disconnections, continuous position changes, and the location-dependent nature of the behavior and communication patterns of the individual components present designers with unprecedented challenges in the areas of modularity and dependability. This paper describes two ideas regarding a modular approach to specifying and reasoning about mobile computing. The novelty of our approach rests with the notion of allowing transient interactions among programs which mobe in space. In this paper we restrict our concert to pariwise interactions involving variable sharing and action asynchronization. The motivation behind the transient nature of the interactions comes from the fact that components can communicate with each other only when they are within a certain range. The notation we propose is meant to simplify the writing of mobile applciaitons and is a direct extension of that used in UNITY. Reasoning about mobile bomputations relies on the UNITY proof logic.

School of Engineering & Applied Science

Assertional Reasoning about Pairwise Transient Interactions in Mobile Computing

Gruia-Catalin Roman
Peter J. McCann
Jerome Plun

WUCS-95-26

1 August 1995

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

Assertional Reasoning about Pairwise Transient Interactions in Mobile Computing

Gruia-Catalin Roman
Peter J. McCann
Jerome Plun

Abstract

Mobile computing represents a major point of departure from the traditional distributed computing paradigm. The potentially very large number of independent computing units, a decoupled computing style, frequent disconnections, continuous position changes, and the location-dependent nature of the behavior and communication patterns of the individual components present designers with unprecedented challenges in the areas of modularity and dependability. This paper describes two ideas regarding a modular approach to specifying and reasoning about mobile computing. The novelty of our approach rests with the notion of allowing transient interactions among programs which move in space. In this paper we restrict our concern to pairwise interactions involving variable sharing and action synchronization. The motivation behind the transient nature of the interactions comes from the fact that components can communicate with each other only when they are within a certain range. The notation we propose is meant to simplify the writing of mobile applications and is a direct extension of that used in UNITY. Reasoning about mobile computations relies on the UNITY proof logic.

Keywords: formal methods, mobile computing, shared variables, synchronization

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

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

1. Introduction

As the miniaturization of electronics progressed, the shrinking size of computing devices made it possible for users to carry computers and even to use them while in motion. The term "mobile computing" came to characterize computations and environments in which computers are free to change location while still able to communicate with one another when they so desire. Mobile computing devices can range from simple emitters ("active badges") and receivers ("beepers", GPS positioning) to full-fledged personal computers (laptop, notebook, subnotebook, palmtop), and include cellular phones and other similar communication devices. While some of these devices can perform self-contained computations, the vast majority of them require communication with a support environment that is designed to accommodate their mobile nature. This is accomplished by adding the notion of a "base station" or "mobile support station." In order to communicate with another mobile peer, a mobile host must route all messages through the mobile support station.

However, there are circumstances where mobile networking functionality is desired, but in which a fixed infrastructure of base stations is inappropriate or simply not available. These situations are the ones that make most use of mobile computing's inherent advantages: flexible, timely, and cost-effective deployment of computing resources to where they are needed most, in an environment where those needs are rapidly changing. One such application is disaster management, where the fixed network might be damaged beyond usability. A similar situation arises when one considers using mobile computers on the battlefield, where there is not enough time to set up a vulnerable mobile support infrastructure. Yet another case is that of a group of conference participants engaged in a formal or informal meeting, sharing data and communicating among themselves. They make up a transient community for which it would not make sense to set up a fixed infrastructure, only to tear it down later. Even in the corporate context there are instances which favor a highly dynamic and low-investment communication structure.

In all these situations, a new kind of mobile computing paradigm is needed: the Ad-hoc Wireless Network (AWN). First coined by Johnson [6], this term refers to a set of mobile nodes that have no (or only a very sparse) support infrastructure and which therefore rely on individual mobile nodes for control and routing functions. This provides a very different environment for protocol writers, application developers, and end users. The AWN presents us with the most extreme example of distribution and we view it an ideal testbed for trying out new ideas regarding how to design and reason about programs in a mobile setting. The AWN challenges us to abandon the old notions of fixed routing tables and easy availability of reliable connections, and to develop new abstractions about program interaction, program composition, and resource availability.

Our research goal is to understand the fundamental issues facing mobile computing and to develop practical design solutions for the flexible and rapid deployment of mobile applications, particularly in the AWN context. Towards this aim, we are investigating new models, constructs, design techniques, and distributed algorithms. One of our objectives is to extend formal methods to the specification of and reasoning about mobility. This entails the addition of novel constructs to existing formal models in order to allow the specification of location-dependent properties and the exploration of proof methodologies that facilitate careful reasoning about systems that have been specified using these new constructs. We are particularly interested in programming language constructs that facilitate modular development, rapid deployment, and easy verification of mobile applications. The main thesis of this paper is that existing models can be adapted to meet these new demands. To argue this point, we first introduce mobility in the UNITY model [4] by augmenting the program state with a location attribute whose change in value is used to represent motion. Next we introduce two extensions to UNITY: transient variable sharing and transient action synchronization. The former allows mobile programs to share data when in close proximity, i.e., a variable owned by one program may be shared in a transparent manner with different programs at different times depending upon their relative location in space. The latter provides a similar mechanism for action synchronization, i.e., a statement owned by one program is executed in parallel with statements owned by other programs when certain spatial conditions are met. Transient data sharing turns out to be a convenient mechanism for the development of highly decoupled mobile computations while transient action synchronization appears to be a good low level model for wireless communication.

For the sake of brevity, we limit our discussion only to pairwise interactions even though the approach can be extended to multi-party interactions. Furthermore, throughout the paper we rely only on very simple examples to illustrate the concepts and notation. All illustrations are inspired by an application involving an airport baggage delivery system with no centralized control (Figure 1) but are greatly simplified by considering only program fragments and special cases.

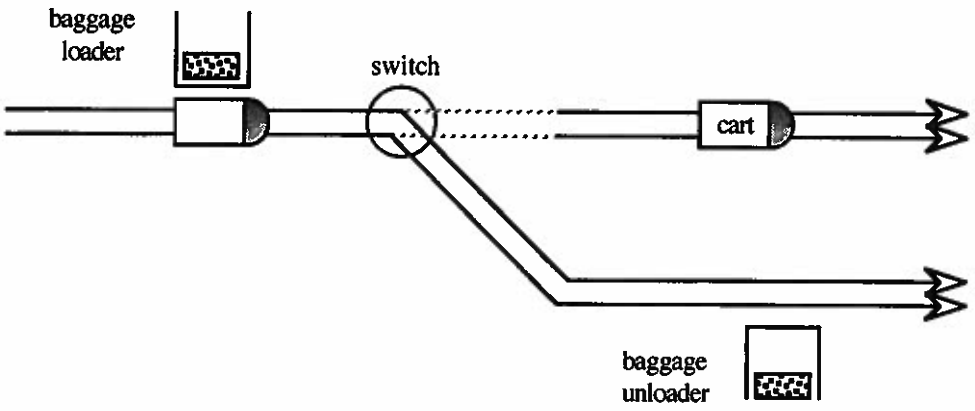

Figure 1. *Carts move along one-way tracks between baggage stations. Wireless communication is used by stations to indicate that baggage is present, by carts to control the track switching, and among carts to avoid collisions.*

In the remainder of this paper we show how transient interactions can be expressed in a highly modular fashion using an augmented UNITY notation and can be reasoned about using the UNITY logic. Section 2 reviews briefly the UNITY syntax and proof logic. A notation for expressing the concepts of location and movement is introduced in Section 3. Section 4 focuses on transient data sharing while Section 5 deals with transient action synchronization. A brief overview of related work on mobile computing and concluding remarks appear in Section 6.

2. A Model of Computing

Before discussing the mobility of programs, we first review the UNITY [4] notation which is used to express the computation taking place within the mobile components of a system and the UNITY proof logic which is applied to reasoning about mobile computations. The general structure of a UNITY program is evident in the program *VirtualCart* shown below

Program VirtualCart
  declare
    position, LoadingPoint, UnloadingPoint: **integer**
  [] *full, BaggageToLoad, ReadyToUnload*: **boolean**
  always
    LoadingEnabled = ¬*full* ∧ *position*=*LoadingPoint* ∧ *BaggageToLoad*
  [] *UnloadingEnabled* = *full* ∧ *position*=*UnloadingPoint* ∧ *ReadyToUnload*
  initially
    full = **false** [] *position* =0
  assign
    position := *position* +1 **if** ¬ *LoadingEnabled* ∧ ¬ *UnloadingEnabled*
  [] *full* := **true** **if** *LoadingEnabled*
  [] *full* := **false** **if** *UnloadingEnabled*
end

This program is designed to simulate the basic actions of one single cart moving along one track (Figure 2) and does not cover the entire cart behavior or any interactions with most other components of a complete system. Issues having to do with switches, baggage destination labels, other carts, etc. are all ignored.

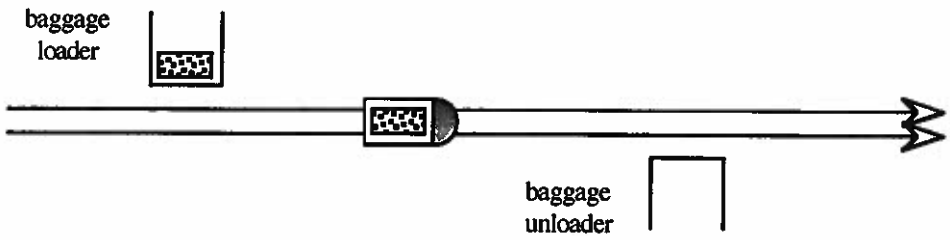

Figure 2. A single cart moving along a linear track.

The variable *position* keeps track of the current location of the cart and the variable *full* indicates whether the cart carries any baggage or not; *LoadingPoint* and *UnloadingPoint* refer to the location of two distinct baggage stations; the boolean constants *BaggageToLoad* and *ReadyToUnload* are true if and only if baggage can be loaded and unloaded at the respective stations. All the Pascal-like variable declarations appear in the **declare** section. The **always** section contains macro definitions for later use. The cart is initially at the left end of the straight-line track and has no baggage—the **initially** section contains a set of equations or constraints giving the acceptable range of initial values. The cart's actions including the move to a new location and random loading and unloading, when feasible, appear in the **assign** section of the program. In general, the **assign** section consists of a set of conditional multiple-assignment statements separated by the symbol "[]". The execution of a program starts in a state satisfying the constraints imposed by the **initially** section. At each step one of the statements is executed. The selection of the statements is arbitrary but weakly fair, i.e., each statement is selected infinitely often in an infinite computation. All computations are infinite.

The UNITY logic is a specialization of temporal logic. Safety properties specify that certain actions (i.e., state transitions) are not possible, while progress properties specify that certain actions will eventually take place. In UNITY, the basic safety property is the **unless** relation. The formula p **unless** q states that if the program enters a state in which the predicate p is true and q is false, every program action will either preserve p or establish the predicate q. All other safety properties, such as **invariant** (written as **inv.**), **constant**, and **stable** are defined in terms of **unless**. The basic progress properties are **ensures** and **leads-to** (written $\mapsto$). The formula p **ensures** q states that p **unless** q holds and, in addition, there is some statement which establishes q, a statement the program is guaranteed to execute in a bounded number of steps. Similarly, the formula $p \mapsto q$ states that if the program enters a state in which p is true, the program will eventually enter a state in which q holds, although p need not remain true until q becomes true.

For illustration purposes, here are several properties one may want to prove about the *VirtualCart* program (all free variables are universally quantified by convention):

$$position = k \ \textbf{unless} \ \langle \exists\, k' : k' > k :: position = k' \rangle^{\dagger}$$

$$\textbf{inv.}\ position \geq 0$$

$$\neg full \wedge position = LoadingPoint \wedge BaggageToLoad \ \textbf{ensures} \ full$$

The first property states that the cart can move only in the forward direction; the second property states that the cart position is always positive; and the last property states that an empty cart in a position to load a piece of baggage is guaranteed to do so in a single step.

3. Mobile Components

By analogy with real-time modeling where time is most often introduced as a distinguished variable not accessible to the programmer, one can capture mobility by introducing a distinguished location variable for each individual program. Restrictions on how such a location variable is accessed and updated would have to reflect the mobility characteristics of the computation. In a cellular network, for instance, the location of the mobile units is

† The three-part notation ⟨ **op** quantified_variables : range_constraint :: expression ⟩ used throughout the text is defined as follows: The variables from *quantified_variables* take on all possible values permitted by *range_constraint*. If *range_constraint* is missing, the first colon is omitted and the domain of the variables is restricted by context. Each such instantiation of the variables is substituted in *expression* producing a multiset of values to which *op* is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range_constraint*, the value of the three-part expression is the identity element for *op*, e.g., *true* if *op* is ∀.

determined by the car or person carrying the computer but constrained to movements from one cell to a neighboring one (as long as the unit is on). The verification of any hand-off algorithm must rely on this assumption. Protocols involved in reestablishing connectivity at the time a mobile computer is powered up may have to assume that initial locations are arbitrary. In some applications a program may have to know its own location while not in others. In the former case the location is directly accessible by the program while in the latter the location plays a role only in reasoning about the computation. As automation technology advances it is also conceivable that certain programs may actually have the ability to control the movement of their carrier. In this case, movement is no longer under the control of the environment but planned by the program which could request future data delivery at specific locations to be reached along the movement path.

In this section we propose a notation for specifying individual mobile programs. We introduce only the notation for specifying computations consisting of a single mobile program—later sections will deal with interactions among mobile programs. We illustrate the notation on a new variant of the cart program, one which allows the cart to change its location. While the variable *position* discussed earlier represented the simulated location of the cart along the track, in the program below we replace *position* by a variable λ which is meant to refer to the actual physical position of the cart. In all other respects the program looks the same.

Program MobileCart **at** λ
  declare
    position, LoadingPoint, UnloadingPoint: **integer**
  [] *full, BaggageToLoad, ReadyToUnload*: **boolean**
  always
    LoadingEnabled = $\neg$*full* $\wedge$ *position* = *LoadingPoint* $\wedge$ *BaggageToLoad*
  [] *UnloadingEnabled* = *full* $\wedge$ *position* = *UnloadingPoint* $\wedge$ *ReadyToUnload*
  initially
    full = **false**
  assign
    $\lambda := \lambda + 1$ **if** $\neg$ *LoadingEnabled* $\wedge$ $\neg$ *UnloadingEnabled*
  [] *full* := **true** **if** *LoadingEnabled*
  [] *full* := **false** **if** *UnloadingEnabled*
end

The semantics of *MobileCart* are different because the assignment $\lambda := \lambda + 1$ is not compiled as an assignment statement but reflects the fact that the program has direct control over and knowledge of its position. The statement is a model of the real movement and must be a correct reflection of the physical world, accurate enough to facilitate reasoning about both functional and mobility aspects of the cart's behavior. More complex models are used for more complicated situations and assumptions must be made about the kind of physical failures one is willing to consider in verifying the correct behavior of a system.

Notation wise, the distinguished variable λ holding the location is declared implicitly alongside the name of the program. Because the program names are assumed to be unique, all variables acquire unique names if prepended by the name of the program in which they are declared, e.g., *MobileCart.full*. This convention is different from that of UNITY and will become important in later sections where communication among mobile programs is discussed. The notation *MobileCart.*λ is used to reference a program's location variable. Implicit in our notational conventions is the notion that a program and its variables are co-located and move as a single unit. The type of λ was left unspecified. Throughout the paper we assume the existence of a global declaration for the spatial context in which the programs move. In most cases, we assume a two-dimensional grid of points. Allowing programs to have diverse perceptions of the basic structure of space adds unnecessary complexity and would obscure the issues we want to consider in this paper. The initial position of individual components will be specified separately from the component itself. In the following sections, we describe two communication mechanisms which use this notation and provide programs with the ability to interact with each other.

4. Transient Variable Sharing

The kinds of AWN applications likely to be built in the next decade are expected to involve large numbers of components (we call them mobile programs) which function in a totally decentralized fashion and have minimal knowledge about each other. The presence of other components cannot be predicted or guaranteed. We characterize this style of computing as decoupled and opportunistic. Each program carries out its own task and communicates with other programs if and when they are present. Yet, when considered as a community (we call it a system), they perform purposeful tasks that the individual members could not accomplish in isolation. In this section we explore a way to transform the earlier version of the *MobileCart* into a new program which accomplishes its task in the context of a community. Towards this aim, we consider three programs corresponding to the control logic for a cart,

a loading station, and an unloading station. Along the way we introduce a new data sharing mechanism, a direct generalization of the shared variable model as employed by UNITY.

Variables are shared only when programs are in each other's proximity. When a variable provides a way to read information belonging to some other program, the variable is called *reflective*—for the sake of simplicity we allow all program variables to be available as potential sources of information for reflective variables. If a variable is shared in a read/write mode by both parties it is called a *coop* variable. Since all the variables have unique names, a new kind of statement is introduced to specify the conditions under which two distinct variables in two distinct programs become logically one. These statements are called *interactions*. They provide the rules by which a set of independent programs becomes a system. Each interaction is a model of both physical reality and specialized operating system services. The physical reality aspect comes from the fact that wireless communication is limited by distance, a key property that should be well understood before proving anything about the overall system. The service provided by the operating system (or its data transport protocol) is the atomic update of the shared variables.

Returning to our example, let us first see how the control logic of the cart can be extracted from the code of *MobileCart* and how its activities can be decoupled from those of the loader and the unloader. If one makes the assumption that the loading and the unloading of the cart is performed by the baggage stations, the only thing the cart needs to do is to control its movement. To do so, however, it requires information about whether it is full or not (let us say that this is available through a coop variable called *full*—see the **declare** section) and whether it ought to wait for baggage to be loaded and unloaded (made known through the reflective variables *loading* and *unloading*—see the **always** section which in standard UNITY holds macro definitions). The resulting program assumes the form

Program Cart **at** λ
  declare
    full : **shared integer**
  always
    loading, *unloading* : **shared boolean**
  initially
    full = **false**
  assign
    $\lambda := \lambda + 1$ **if** $\neg(\neg full \wedge loading) \wedge \neg(full \wedge unloading)$
end

Ignoring for the moment the manner in which data sharing is accomplished, it is clear that this program is concerned only with advancing the position of the cart along the track subject to the conditions that it ought not to pass empty by a loader that has luggage and it ought not to pass full by an unloader that has room to take the luggage. It should be noted, however, that this interpretation is only one of many possible interpretations. What the cart actually does depends upon the structure of the system in which the cart finds itself, the computational and mobility behavior of other programs in the system (not known to the cart), and the definition of the interactions among programs.

The code for the loading controller consists of only two statements. The first one checks if the cart is present and empty ($\neg$ *no_cart_available*) and there is baggage (*baggage*) to load; if this is so it marks the fact that the cart is no longer empty and that the baggage was loaded. The second statement simulates the arrival of a new piece of luggage.

Program Loader **at** λ
  declare
    no_cart_available : **shared boolean**
  [] *baggage* : **boolean**
  initially
    no_cart_available = **true**
  assign
    no_cart_available, *baggage* := **true**, **false if** $\neg$*no_cart_available* $\wedge$ *baggage*
  [] *baggage* := **true**
end

The code for the unloading controllers is very similar but here baggage taken off the cart marks the cart as not having anything to deliver. A second statement simulates the fact that the unloaded baggage was picked up.

Program Unloader **at** λ
  declare
    baggage_arriving : **shared boolean**
  ▯ *no_baggage* : **boolean**
  initially
    baggage_arriving = **false**
  assign
    baggage_arriving, no_baggage := **false**, **false if** *baggage_arriving* ∧ *no_baggage*
  ▯ *no_baggage* := **true**
end

We can specify now a system consisting of a cart, a loader, and an unloader by assigning to each program an initial location along the track (which remains unchanged for the loader and the unloader) and by defining the interactions among these programs which are totally unaware of each other's existence. The result is the specification below.

System *FromHereToThere*

  Program Cart **at** λ ...

  Program Loader **at** λ ...

  Program Unloader **at** λ ...

  Components
    Cart **at** 0; Loader **at** 10; Unloader **at** 30

  Interactions
    Loader.*no_cart_available* ≈ Cart.*full*
      when Cart.λ = Loader.λ
      engage Cart.*full*
      disengage Cart.*full* = **current** || Loader.*no_cart_available* = **true**

    Cart.*loading* = Cart.λ = Loader.λ ∧ Loader.*baggage*

    Unloader.*baggage_arriving* ≈ Cart.*full*
      when Cart.λ = Loader.λ
      engage Cart.*full*
      disengage Cart.*full* = **current** || Unloader.*baggage_arriving* = **false**

    Cart.*unloading* = Cart.λ = Unloader.λ ∧ Unloader.*no_baggage*
end

The **Interactions** section makes use of two distinct notations corresponding to the reflective and the coop variables. Their syntax and semantics is considered below. An interaction statement such as

Cart.*loading* = Cart.λ = Loader.λ ∧ Loader.*baggage*

is essentially a definition involving a reflective variable on the left and an expression on the right, a predicate in this case. The value of the reflective variable is computed in terms of the variables of a second program. The expression is usually contingent upon the current location of the two programs. Above the cart becomes aware of the loading process whenever it is at the same location as a loader which has baggage to send along.

Specifying interactions between coop variables is more complicated. The transient nature of the sharing allows the variables to become one—the term we use is *engage*—at a time when their values are distinct and thus raising the question of which value to be used. There is also the issue of what is left in each variable when the programs move away and the sharing is disabled—we use the term *disengage*. For these reasons, the coop interactions specify not only the condition under which the sharing takes place (in the **when** clause) but also the value to be used when sharing is activated (the expression in the **engage** clause) and the values left in each variable at the time the sharing condition becomes false (see the **disengage** clause in which **current** is used to refer to the common value just prior disengagement). For instance, the interaction

Loader.*no_cart_available* ≈ Cart.*full*
when Cart.λ = Loader.λ
engage Cart.*full*
disengage Cart.*full* = **current** || Loader.*no_cart_available* = **true**

states that the cart being full is the same as no cart being available to the loader when the cart is lined up with the loader. When the cart is present, the variables *no_cart_available* and *full* act as if they were a single shared variable whose initial value is determined by the state of the cart. Once the cart moves away, it retains the information about its status while the loader simply defaults to the assumption that the cart is no longer available. Of course, the cart may still be present but not available because the loader recorded that fact in the shared variable. Furthermore, when the cart is not present, the loader may change the value of *no_cart_available* in any arbitrary manner but its true value is regained as soon as a cart arrives.

At this point it is natural to ask several critical questions. Is the concept of transient data sharing useful? Can one use UNITY to reason about these kinds of systems? What is the technical significance of this concept? In the remainder of this section we address these specific issues.

In the long term, we seek to identify abstract constructs that enhance the modularity and the flexibility of mobile applications. Because the concept of shared variable plays a central role (both formally and pragmatically) in concurrency, it seemed to us that it might be a good starting place, especially given the fact that mobile components often do share data directly (e.g., a laptop and a docking station). Our experience to date indicates that transient data sharing can contribute to a very significant extent to decoupling among the components. Individual programs have no knowledge of the identity of other programs in the system. The cart, for instance, is not aware of how many loaders and unloaders there are. Changes in the design of individual components often have effects limited only to the definition of the interactions. The loader and unloader can freely change data representation and behavior as long as the same basic information can be communicated to the cart through minor changes in the **Interactions** section; actually, a baggage station that supports both loading and unloading can be created without the cart code being affected in the least. The same programs work in a multitude of configurations using varying numbers of components. The baggage stations can be fixed or even mobile and their number and location can be changed in arbitrary ways. Finally, by hiding the communication behind what amounts to be a set of declarative rules the programming task is greatly simplified; all communication responsibilities are relegated to the runtime support system.

Because at a logical level location is simply a local variable, it is relatively easy to see that the UNITY logic is directly applicable. The transient nature of the data sharing is the only potential source of complications. Fortunately, extending the proof logic to accommodate transient data sharing involves only the redefinition of the **unless** and **ensures** properties in a manner that factors the effects of interactions. Since this paper presents a highly simplified version of our model, the task is not particularly difficult. The key simplifying assumptions are: (1) all interactions are pairwise, i.e., at any one time a variable is either disengaged or engaged with one and only one other variable; (2) data sharing is controlled only by the relative position of components and not by their state, i.e., the **when** clause can refer only to the location variables of two programs; (3) the engagement and disengagement values are determined only by the values of the variables involved in the sharing relation; and (4) each program has full control over its own location.

Using these assumptions and ignoring the reflective variables which can be seen as a special case of coop variables, the proof of **unless** and **ensures** remain unchanged except for the domain of statements over which the property is verified. In UNITY, proving p **unless** q requires one to show that

$$\{p \wedge \neg q\}\ s\ \{p \vee q\}$$

holds for every statement s in the whole program. In our model we have to carry out the same proof but over a set of statements obtained by (logically) modifying each individual program statement so as to account for its potential side effects when one of its left-hand variables is involved in or controls sharing. To illustrate this, let us consider the statement

$$x := x + 1$$

(We omit the program names by assuming that they are known from context.) If x is shared with y when p holds, the proof must consider the modified statement of the form

$$x := x + 1 \textbf{ if } \neg p \parallel x, y := x+1, x+1 \textbf{ if } p$$

Similar (logical) changes must be considered for statements that reflect movement. Given a statement such as

$$\lambda := \lambda + 1$$

which controls the sharing among x and y through an interaction of the form

$$x \approx y \textbf{ when } \lambda{=}\mu \textbf{ engage } x \textbf{ disengage } x{=}0 \parallel y{=}1$$

the modified statement is logically equivalent to

$$\lambda := \lambda + 1 \parallel x, y := x, x \textbf{ if } \lambda{+}1{=}\mu \parallel x, y := 0,1 \textbf{ if } \lambda{+}1{\neq}\mu$$

The same domain of modified statements must be considered in proving an **ensures** property. Of course, there is no need to actually transform the program text. The rules of inference (not shown in this paper) take into account these logical side-effects.

The notion of transient shared variables is an interesting generalization of a well established computing paradigm. The engagement and disengagement protocols are closely tied to the notions of cache coherence and reconciliation among multiple versions of a database. The declarative nature of the interactions shift the communication burden from the programmer to the runtime system and leads to a highly decoupled style of computing. They also provide an interesting generalization of one of the two program composition strategies of UNITY, program union. We are currently working on the development of proof rules that cover the most general case. We are also investigating strategies for proving system properties solely from the properties of the individual programs and the definition of the interactions. Finally, we plan to explore mechanisms for implementing transient data sharing via wireless communication.

5. Transient Synchronization

In the previous section, we considered transient variable sharing, which is the natural extension of UNITY union to the mobile setting. In this section, we turn to transient action synchronization, a generalization of UNITY superposition. Recall that in UNITY, program a is said to be superposed on program b if every statement of a is synchronized with some statement of b and no statement of a writes to (but can read) any variable of b. Inference rules for deriving the properties of a composite program from those of the components are also available in standard UNITY. In the mobile setting, we allow for a similar relationship, but here the interaction is transient. This reflects the highly dynamic nature of mobile computations: they must continually adapt to their environment by reconfiguring the connections among programs.

Transient synchronization might be desired in situations where action coordination is needed, but where location plays an important role in who the participants are. This is true if two programs must coordinate access to a location-dependent resource, or if one program needs to communicate synchronously with a co-located neighbor. Keep in mind that at some physical level communication between mobile hosts is inherently synchronous—a receiver must be listening to receive a transmitted bit. Our synchronization abstraction should be able to capture this form of interaction as well. This section discusses some of the ramifications of a particular model of transient synchronization, including its impact on the proof logic.

We turn again to the airport baggage system for an example. Figure 3 shows a track intersection. Some form of synchronization will be necessary to avoid the possibility of collision.

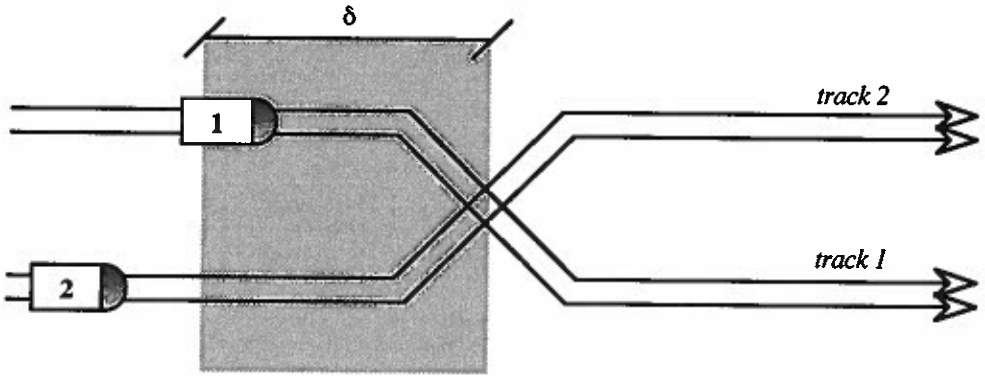

Figure 3. *Two carts are approaching an intersection. Coordination of some sort is required to avoid collision.*

The distance δ represents the "danger zone" in which we require the two carts to coordinate their movements. If both carts are closer than δ, but neither has passed the intersection, they must interact to avoid collision. We can model this interaction as a transient synchronization of statements controlling the carts' movement. This can be used to speed up one cart and retard the motion of the other. If cart *1* is closer to the intersection than cart *2*, then the predicate controlling the interaction must reflect that fact and adjust the sense of the interaction accordingly.

The notation for transient synchronization is very similar to that for transient variable sharing, except that now the objects that interact are statements instead of variables:

```
System MeetingAtCrossing

    Program Cart(i) at λ
        assign
            slow    ::  λ := λ +(1,0)
        ▯   fast    ::  λ := λ +(2,0)
    end

    Components
        Cart(1) at (0,1);  Cart(2) at (0,2)

    Interactions
        coselect Cart(i).fast, Cart(j).slow || inhibit Cart(j).fast
            when  i≠j ∧ Cart(i).λ>Cart(j).λ
                ∧  distance_to_crossing(Cart(i).λ)≤δ
                ∧  distance_to_crossing(Cart(j).λ)≤δ
end
```

The basic code for the two programs is shown in the first section, the **Program** definition. The code assumes that each location variable is of type ordered pair, with the first element of the pair representing location on the track, and the second element representing the track number. We label each statement so that it can be referred to later in the **Interactions** section. Note that each program can move the cart at one of two speeds during an atomic step. The position of a cart is incremented by either one or two units, depending on which statement (*slow* or *fast*) is selected. Carts never switch tracks. The **Components** section declares two carts, both at location zero but on different tracks. The **Interactions** section specifies how the statements become synchronized as the programs move relative to each other and the location of the intersection. Statement labels, like the variable names from transient sharing, must be prepended with the program name to disambiguate them.

The **when** predicate above controls both the **coselect** and **inhibit** interactions. It is enabled whenever both carts are close to the intersection (distance less than δ) and when cart *i* is closer than cart *j*. This ensures that cart *i* must execute statement *Cart(i).fast* synchronously with cart *j* executing *Cart(j).slow*. Also, *Cart(j).fast* is inhibited under these conditions. These constraints ensure that cart *j* will not catch up to cart *i* through some interleaving of statements that would have otherwise been perfectly fair and allowed by the scheduler.

Extending the **coselect** relationship to account for transitivity is a matter for future work. For now, we assume that if a statement appears in more than one **coselect** statement, the **when** predicates must be disjoint (i.e., a statement can be synchronized with at most one other statement). Operationally, the **coselect** construct synchronizes the pair of statements given whenever the **when** predicate is true. This can be thought of as a constraint on the scheduler for the system as a whole so that it is inhibited from selecting either statement alone. Of course, conditional statements, those followed by an **if** clause, have no effect if their condition is false, even if selected to execute by the scheduler.

The operational semantics of the **inhibit** construct are such that the statement given is inhibited from executing (as a singleton statement) whenever the **when** predicate is true. This could be accomplished through the use of conditional statements except that programs are no longer allowed to reference variables from other programs, and so predicates that make use of variables from more than one program, such as relative locations, must always be a part of the **Interactions** section. This helps to achieve greater modularity by isolating the interface constraints to this one section and removing them from the code of individual programs. Note that an **inhibit** clause cannot inhibit a member of a **coselect**ed group from executing as a group.

A critical evaluation of the transient synchronization idea is currently underway. Transient synchronization promises to be a useful construct that helps solve real problems. Second, it is possible to use the construct in the formal framework provided by UNITY. Finally, it makes a technically meaningful contribution to the field of formal methods as an extension of UNITY-style program composition. We address these issues in the remainder of this section.

The transient synchronization abstraction presented here allows for modular design of flexible components. The carts in the above example have no knowledge of the identity of any programs with which they are interfaced, and the synchronization framework assures correct operation through coordinated action. The isolation of the interface to the **Interactions** section means that programs can be written without knowledge of the constraints that must be satisfied by two programs that must co-exist in some locale for a brief period of time. The designer can then focus on the problem at hand. Location-dependent synchronized action is required in a variety of settings that designers are likely to encounter. These include coordinated access to a local resource (such as a shared printer), communication with co-located neighbors, or situations like the example above where synchronous movement is required to avoid erroneous behavior.

The UNITY proof logic can certainly be used to prove properties of mobile computations that use the transient synchronization construct. In what follows, we attempt to characterize the axiomatic semantics of the **coselect** and **inhibit** clauses. Here we ignore transient sharing and consider a system where the components interact only through synchronization. Unifying the two concepts is a matter for future work.

Proving **unless** properties for a system (p **unless** q) entails proving the basic Hoare triple $\{p \wedge \neg q\}\ s\ \{p \vee q\}$ for every statement s in the system. Under transient synchronization, we must prove this property not only for every singleton statement but also for every pair of statements that might execute together. For example, if we have the interaction

coselect a, b **when** r

then we must prove $\{p \wedge \neg q\}\ a \| b\ \{p \vee q\}$. We can refer to a pair of statements (a, b) as a "feasible synchronic group" when r is true and an "infeasible synchronic group" when r is false. Note that if a pair of statements is an infeasible synchronic group (r false or no **coselect** relation specified), we are not obliged to prove the above, i.e., if $p \wedge \neg q \Rightarrow \neg r$, then the substitution holds trivially, because infeasible pairs do not change the state and thus preserve p. Note also that even though the domain of statements over which properties must be proven has grown (potentially every pair of statements in addition to the original singleton statements), in practice, the proof complexity depends only upon the number of clauses in the **Interactions** section, which usually is very small.

The **ensures** property can also be proven by simply changing the domain of statements over which it operates. We can prove p **ensures** q by proving (1) p **unless** q for the system, (2) there exists some action that remains a part of some feasible synchronic group until q is established, and (3) any feasible synchronic group containing the action establishes q upon execution in a state satisfying $p \wedge \neg q$. We can preserve the rest of the UNITY logic, because other program properties can be specified in terms of **unless** and **ensures**. This gives us a powerful axiomatic system for reasoning about program correctness.

We can thus think of the **coselect** interaction as transiently adding the pair (a, b) to the set of feasible synchronic groups and removing the singleton groups corresponding to a or b executing alone. An **inhibit** clause can be thought of as transiently removing the singleton statement corresponding to its argument from the set of feasible synchronic groups. A very general theory of dynamic (state-dependent) synchronic groups and their implications for a proof logic was developed in [5]. The Swarm model [9] is one of the few systems to provide both language-level support for dynamic synchronization of actions and an assertional proof system. Transient synchronization is an interesting generalization of one of the two program composition strategies for UNITY, superposition. We have tried to eliminate the asymmetry of program superposition while retaining its usefulness as a program interface mechanism.

6. Related Work and Conclusions

A number of researchers are actively involved in creating environments for mobile-aware applications. This issue is discussed by Noble, Price, and Satyanarayanan in [8], and by Badrinath in [1, 2, 3]. Badrinath describes the overall impact of mobility on distributed applications in [1], and explains how traditional distributed algorithms typically rely on a fixed pattern of connectivity among nodes. In [2], he describes a programming methodology for mobile applications that pushes as much of the computation as possible into the fixed infrastructure. He stresses the importance of mobile-aware applications in [3], and presents a framework for delivering notifications about changes in the dynamic mobile environment to applications. Noble [8] describes an interface for mobile-aware applications that allows them to more carefully manage the scarce and unpredictable resources of the mobile computer such as bandwidth and connectivity.

These strategies for the design of mobile-aware applications either attempt to remove the dynamic factors from the mobile setting (by pushing computation into a fixed infrastructure) or focus on bringing more information about the dynamically changing environment up to the application itself, which is supposedly the only layer with enough information to manage changing resources. Our work differs from these strategies in that our goal is to

provide dependable systems that work in more decoupled fashion and in situations where fixed infrastructures might not be present, and we accomplish this by keeping the **Interactions** section separate from the underlying computation. We hope that this separation of program interface concerns from component implementation concerns will prove to be an important advantage over the strategy of micro-managing resources for maximum efficiency.

Other more formal work related to mobile computing is Robin Milner's pi-calculus [7]. He presents a formal framework for reasoning about reconfigurable systems from the perspective of an event-based behavioral model of computation. Our work adopts the UNITY approach of axiomatic reasoning, which historically has been an important counterpoint to the process algebra style of reasoning. Also, we hope that from an operational standpoint, UNITY and our transient interaction constructs might be more accessible to the average designer and more directly implementable.

This paper has presented two models of pairwise transient interaction and has shown that axiomatic reasoning can be applied to the mobile computing domain. One of the basic themes of this paper is the idea that location is central to specifying and reasoning about mobile computations. We have shown that location can be introduced into a traditional programming notation in a straightforward manner and with little or no impact on the ability to reason about the resulting computations. Furthermore, when combined with the concept of transient data sharing and transient synchronization, spatial qualifications contribute to achieving a high degree of modularity in the specification of mobile computations. The examples illustrate the kind of modular designs one can specify by using location to control (directly or indirectly) the interactions among programs. Research to date supports the conjecture that the abstractions introduced here represent a reasonable starting point for the study of mobile computing and new ways to interface mobile programs. Transient data sharing and transient synchronization are both potentially useful programming constructs and promising research vehicles for investigating ways to generalize the UNITY style of program composition.

References

[1] Badrinath, B. R., Acharya, A., and Imielinski, T., "Impact of Mobility on Distributed Computations," *ACM Operating Systems Review*, vol. 27, no. 2, 1993.

[2] Badrinath, B. R., Acharya, A., and Imielinski, T., "Structuring Distributed Algorithms for Mobile Hosts," *14th IEEE Intl. Conf. on Distributed Computing Systems*, Poznan, Poland, 1994.

[3] Badrinath, B. R., and Welling, G., "Event Delivery Abstractions for Mobile Computing," Rutgers University, Technical Report LCSR-TR-242, 1995.

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

[5] Cunningham, H. C., Roman, G.-C., and Plun, J. Y., "Assertional Reasoning about Dynamic Systems," in *Parallel Computing: Paradigms and Applications*, A. Y. Zomaya, Eds., Chapman & Hall, London, UK, 1994.

[6] Johnson, D. B., "Routing in Ad Hoc Networks of Mobile Hosts," *Proceedings of the IEEE Workshop on Mobile Computing Systems and Applications*, 1994.

[7] Milner, R., Parrow, J., and Walker, D., "A Calculus of Mobile Processes, Part I," Laboratory for Foundations of Computer Sciences, Department of Computer Science, University of Edinburgh, UK, LFCS Report Series ECS-LFCS-89-85, 1989.

[8] Noble, B. D., Price, M., and Satyanarayanan, M., "A Programming Interface for Application-Aware Adaptation in Mobile Computing," School of Computer Science, Carnegie Mellon University, Technical Report CMU-CS-95-119, 1995.

[9] Roman, G.-C., and Cunningham, H. C., "Reasoning about Synchronic Groups," in *Research Directions in High-Level Parallel Programming Languages*, J. P. Banâtre, D. L. Métayer, Eds., Springer-Verlag, New York, NY, vol. 574, pp. 21-38, 1992.