School of Engineering & Applied Science

Mobile UNITY: Reasoning and Specification in Mobile Computing

Gruia-Catalin Roman
Peter J. McCann

WUCS-96-08

7 May 1996

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

Mobile UNITY: Reasoning and Specification in Mobile Computing

Gruia-Catalin Roman
Peter J. McCann

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 present designers with unprecedented challenges in the areas of modularity and dependability. So far, the literature on mobile computing is dominated by concerns having to do with the development of protocols and services. This paper complements this perspective by considering the nature of the underlying formal models that will enable us to specify and reason about such computations. The basic research goal is to characterize fundamental issues facing mobile computing. We want to achieve this in a manner analogous to the way concepts such as shared variables and message passing help us understand distributed computing. The pragmatic objective is to develop techniques that facilitate the verification and design of dependable mobile systems. Towards this goal we employ the methods of UNITY. To focus on what is essential we center our study on *ad-hoc networks* whose singular nature is bound to reveal the ultimate impact of movement on the way one computes and communicates in a mobile environment. To understand interactions we start with the UNITY concepts of union and superposition and consider direct generalizations to transient interactions. 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 employ is a highly-modular extension of the UNITY programming notation. Reasoning about mobile computations relies on extensions to the UNITY proof logic.

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

Correspondence: All communications regarding this paper should be addressed to

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

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

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

1. Introduction

Mobile computing is a new paradigm characterized by the ability of computers to change location while still able to communicate with one another when they so desire. Because of movement, frequent disconnections, power limitations, bandwidth restrictions, and limited local resources, designers tend to think of mobile computing as being distinct from traditional distributed computing. Nevertheless, much of the current research attempts to cast mobility in terms of established distributed programming solutions. So far, the literature on mobile computing is dominated by concerns having to do with the development of protocols and services. Perhaps the most prominent of these efforts is Mobile IP [1], a protocol under development by the IETF Mobile IP Working Group. This work attempts to hide mobility at the network level, allowing higher-layer services such as TCP to continue uninterrupted as hosts move from subnet to subnet. Ideally, this would reduce correctness of a mobile algorithm to that of a simple distributed one, and no special tools for reasoning about mobility would be required.

There are situations, however, in which mobile networking functionality is desired, but where a fixed infrastructure such as the one presumed by Mobile IP is inappropriate or simply not available. These situations are the ones that make most use of the inherent advantages of mobile computing: flexible, timely, and cost-effective deployment of computing resources in an environment where needs are rapidly changing. One such application is disaster management, where the fixed network might be damaged beyond usability. Another application may involve a group of conference participants engaged in a formal or informal meeting, sharing data and communicating among themselves. Both applications require an *ad-hoc network*. First coined by Johnson [2], this term denotes a set of mobile nodes that have no (or only a very sparse) infrastructure of base stations and thus a greater reliance on individual nodes for control and routing functions. The ad-hoc network 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. A key premise of this paper is the notion that ad-hoc networks, due to their singular nature, are likely to play a critical role in helping us develop a better understanding of mobile computing.

Even with a routing infrastructure in place, there are still aspects of mobile networks that are not completely transparent to applications. An end-to-end mobile application must adapt to the widely varying bandwidth of the wireless connection, which might range from many megabits per second if a laptop is docked with a desktop workstation all the way down to zero as it leaves a wireless coverage area completely. Badrinath and Welling [3] describe a C++ abstraction for delivering events such as bandwidth variations, disconnections, and battery measurements to applications. Noble, Price, and Satyanarayanan [4] present the *Odyssey* application library for managing changing resources and emphasize the importance of application- and data type-specific policies for reacting to changes in the environment. These works demonstrate that the issues in mobile computing are broader than just the packet routing problem.

The dynamically changing resources present in the mobile setting can be dealt with in other ways as well. Mobile-tolerant filesystems attempt to accommodate environmental changes by relaxing the consistency guarantees traditionally offered by distributed filesystems. Satyanarayanan et al. [5] describe results from usage experience with the Coda filesystem, which offers users the ability to operate while completely disconnected from the network. This is accomplished by *hoarding* user files on client machines and emulating remote servers from the data in the local cache. Writes to the local cache are allowed to proceed, and files must be re-integrated when contact with the remote server is re-established. The authors note that conflicts occur less often than might be expected due to the low degree of write-sharing in a user filesystem. Tait and Duchamp [6] present an implementation of a variable-consistency filesystem that allows the application to determine the consistency model used when reading data. Although the authors do not emphasize disconnected operation, they do highlight the need for application-specific levels of consistency. Thus, mobile applications can be written using a traditional model for interaction with the outside world (the filesystem), but reasoning must be carried out under weakened consistency assumptions.

Replicated databases bear a strong resemblance to replicated filesystems, although here the patterns of access may be very different and many more write conflicts can be expected if weak consistency models are introduced. Here too, however, work has illustrated the need for weaker consistency models in the face of mobile computing. The Bayou system described by Terry et al. [7] focuses on two applications, a meeting room scheduler and a bibliography database, and provides for application-specific integration policies so that updates to the database that were written while disconnected can be integrated later into a primary copy. Re-integration is accomplished sporadically on a pairwise basis, and the only guarantee offered is *eventual consistency*.

In addition to disconnected operation, users of mobile computers will also demand *location-dependent* services. Two independent groups have worked on location-dependent World Wide Web applications for mobile computing. Voelker and Bershad [8] describe Mobisaic, a modification to existing web browser technology that resolves queries for pages based on the current location. Acharya et al. [9] describe a similar system. Schilit,

Adams, and Want [10] argue that applications need to change their behavior in a location-dependent (and more generally in a context-dependent) manner.

Though most of the work discussed above is directed towards the production of systems, the literature indicates directions in which formal models of computation need to be extended in order to facilitate reasoning about mobile computing. Research on distributed (and parallel) computing shares a fundamental set of assumptions which are made manifest in the way one thinks about computations, verifies programs, develops algorithms, and designs systems. The two dominant paradigms, shared variables and message passing and their synchronous and asynchronous variants, are not only convenient programming abstractions but also the distillation of our current understanding of what is the essence of distributed computing. If, as tentatively suggested by the case of the ad-hoc network, these abstractions are no longer adequate, one needs to ponder some very basic issues. What is mobile computing? What are proper abstractions for communication and composition in the presence of mobility? These are precisely the kinds of questions we start posing in this paper. A highly abstract version of the ad-hoc network serves as a vehicle for our investigation while UNITY [11] provides the needed programming notation and formal tools. The parsimony and static nature of UNITY is both a challenge and an opportunity. Extensions are needed to accommodate mobility but, by being faithful to its minimalist philosophy, the additions to the programming notation and their implications on the proof logic are likely to provide valuable insights on how one might answer the questions we posed earlier. We call the resulting model Mobile UNITY.

In Mobile UNITY, each program is a unit of mobility. We capture movement by augmenting the program state with a location attribute whose change in value is used to represent motion. Communication, in its ephemeral flavor typical of mobile computations, is expressed using two novel constructs: 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.

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—for the sake of simplicity, we limit our discussion to pairwise interactions. Section 2 reviews briefly the UNITY syntax and its proof logic and proposes a notation for expressing the concepts of location and movement. Section 3 focuses on the transient data sharing mechanisms while Section 4 deals with transient action synchronization. Simple examples, inspired by an application involving an airport baggage delivery system with no centralized control, are used to illustrate the concepts, the notation, and their support for modular specification. Section 5 shows how the UNITY proof logic can be extended to accommodate the new constructs. This is an important technical result of this paper. Section 6 illustrates our proof method on an example constructed by restructuring the program fragments presented in earlier sections. Of particular significance is the fact that, despite apparent increases in the proof complexity, easily identified simplifications often reduce the proof obligations to the level of those encountered in standard UNITY. Brief concluding remarks appear in Section 7.

2. A Model of Mobility

We favor a state-based model, axiomatic reasoning, and explicit representation of space and its properties. While the choice of underlying model may be a matter of personal taste, we contend that modeling the space explicitly is desirable when one hopes to take into account the physical reality of moving objects and its implications on the behavior of the software that they carry. Because the behavior exhibited by a component is affected by what other components are in its proximity, location is likely to play an important role in reasoning about mobile computations. This is the reason why we treat mobility as a change in the location of a component—a mobile program.

To the best of our knowledge, the only formal model of concurrency to refer to "mobility" explicitly is π-calculus [12], a process algebra proposed by Milner and his colleagues. In π-calculus, however, there is no formal concept of space. Mobility is equated to the ability to "express processes which have changing structure." Under this definition any model able to pass processes as values, e.g., the Actor model [13], or link names as values (π-calculus) qualifies. This is clearly distinct from the modeling strategy we explore in this paper.

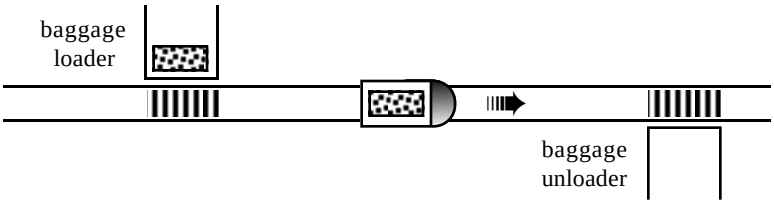

Fig. 1. *A single cart moving along a linear track.*

In this paper we use the UNITY [11] notation to express the computation taking place within the mobile components of a system and the UNITY proof logic to reason about mobile computations—both are extended appropriately to account for the effects of movement and transient interactions. To introduce the reader to the UNITY notation we consider first a program called *VirtualCart* designed to simulate the basic actions of one single cart moving along one linear track at one of two speeds (Fig. 1). For the time being, we ignore interactions with other components of a complete system and the existence of switches, baggage destination labels, multiple carts, etc. The code below illustrates the general structure of a UNITY program:

```
Program VirtualCart
    declare
        position : real
    initially
        position > 0
    always
            smallstep     = 0.1
        []  largestep     = 0.2
        []  Loading       = NextLoadingPoint(position)
        []  Unloading     = NextUnloadingPoint(position)
    assign
            position := position + smallstep     if ¬Loading ∧ ¬Unloading
        []  position := position + largestep     if ¬Loading ∧ ¬Unloading
end
```

Pascal-like variable declarations appear in the **declare** section. The variable *position* is a real number which keeps track of the current location of the cart. The **initially** section contains a set of equations or constraints giving the acceptable range of initial values—the cart starts at some positive location on the straight-line track. The **always** section contains macro definitions for later use. *NextLoadingPoint* and *NextUnloadingPoint* are functions used to compute the values of the booleans *Loading* and *Unloading*. The two functions return true whenever the cart is aligned with a station where bags must be loaded or unloaded, respectively. Throughout the paper we employ italics inside the code to denote functions whose definitions are omitted for the sake of brevity. The constants *smallstep* and *largestep* allow the cart to move slowly or fast, respectively. The symbol "[]" separates the definitions and serves as a separator among equations and statements as well.

The cart's actions, the move to a new location in this case, 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 "[]". In our example the cart moves only if it is not loading or unloading. 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 **co** relation. The formula p **co** q states that in a state in which the predicate p is true, every program action will establish the predicate q. All other safety properties, such as **unless** (defined as $p \wedge \neg q$ **co** $p \vee q$), **invariant** (written as **inv.**), **constant**, and **stable** are defined in terms of **co**. The basic progress properties are **ensures** and **leads-to**. 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 **leads-to** 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 **unless** ⟨ ∃ k' : k' > k :: position = k' ⟩†

inv. position ≥ 0

position = k ∧ ¬Loading ∧ ¬Unloading **ensures** position > k

The first property states that the cart can move only in the forward direction; the second property states that the cart position is always non-negative; and the last property states that a cart which is not loading or unloading is guaranteed to advance to a new location in a single step. Any attempt to prove that the cart will eventually reach some arbitrary position *k'* greater than *k*, i.e.,

position = k ∧ k' > k **leads-to** position = k'

will fail because in a large step the location may be passed over and also because once reaching its next loading or unloading station the cart can no longer move, unless the station communicates the fact that the operation is complete. But this requires communication, a subject outside the scope of this section.

Next, we propose a notation for specifying movement. For now, we are concerned only with 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
    initially
        λ > 0
    always
            smallstep     = 0.1
        []  largestep     = 0.2
        []  Loading       = NextLoadingPoint(λ)
        []  Unloading     = NextUnloadingPoint(λ)
    assign
            λ := λ + smallstep     if ¬Loading ∧ ¬Unloading
        []  λ := λ + largestep     if ¬Loading ∧ ¬Unloading
end
```

Nevertheless, the semantics of *MobileCart* are different because the assignments to λ are not compiled as assignment statements but reflect the fact that the program has direct control over and knowledge of its position. Such statements model real movement and must represent a correct reflection of the physical world, accurate enough to facilitate reasoning about both functional and mobility aspects of the cart's behavior. For the sake of simplicity we assume that all location increments represent unit time moves, i.e., velocity multiplied by one unit of time. Even though movement is continuous, the movement statements must be viewed as atomic state changes associated with the arrival at the new location in order to make them fit with the interleaved model of concurrency used by UNITY. This has interesting implications on the statement scheduling strategy in the runtime system supporting the execution of Mobile UNITY programs, e.g., a guard on a movement statement ought not to change during the unit of time it takes to complete the move. In this paper we simply assume that the implementation maintains the appearance of an interleaved atomic execution and we use this fact when reasoning about such programs.

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.Loading.* 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

† 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 ∀.

unspecified. Throughout the paper we assume the existence of a global declaration for the spatial context in which the programs move.

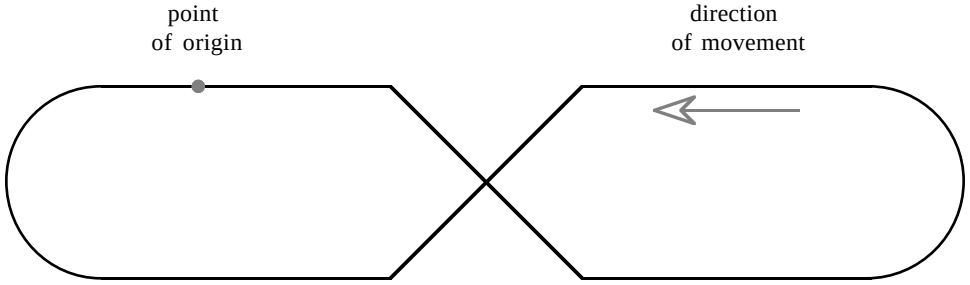

Fig. 2. *A liner space used in the baggage delivery illustration.*

The space is a figure-eight loop (Fig. 2). Some arbitrary point is selected to represent the location zero and the location of all other points is given relative to this distinguished location. The length of the loop is a whole number of units and movement occurs in one-tenth increments. All movement is in the positive direction and the place where the loop crosses over is treated as two distinct logical locations.

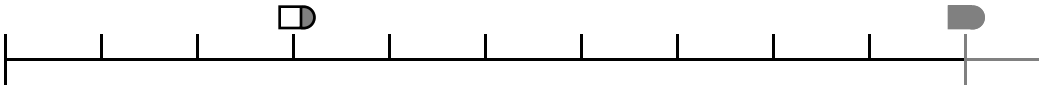

Fig. 3. *Discrete representation of the continuous space.*

Each unit-length section of track is called a segment and is divided into ten intervals, closed on the left and open on the right (Fig. 3). A cart always stops at the left hand side of the intervals. Because the length of the cart is less than one tenth of a unit, the cart can be safely treated as a mathematical point that advances one or two tenths of a unit at a time. As explained later, two carts are not allowed to share a segment. This means that a cart must stop at the beginning of the last interval if some other cart is present on the next segment—a safe distance is guaranteed to remain between them at all times.

In general, restrictions on how such a location variable is accessed and updated must reflect the mobility characteristics of the computation. In a cellular network, for instance, the location of the mobile unit 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. In a robot application it is also conceivable that a program may actually have the ability to control the movement of its 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.

Our choice of UNITY over similar models of concurrency such as TLA [14] was dictated largely by our broad experience with UNITY and the simplicity with which programs are represented. Also, we will see later that it is convenient to have the proof logic depend on the text of the program with only a few basic constructs such as **co** and **ensures**. Our treatment of location as a distinguished variable was inspired by [15].

In the following sections, we describe two communication mechanisms which use this notation and provide programs with the ability to interact with each other.

3. Transient Variable Sharing

The kinds of ad-hoc network 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 consider the data stored on each individual cart. It consists of three values denoting some unique bag identifier along with its source and destination. Station identifiers are used to identify the source and destination of bags. The bag identifier, *bagid*, is needed in the proofs where it acts as an auxiliary variable and at the loaders which may keep a log of arriving bags. The origin of the bag being transported, *source*, is needed in order to allow unloaders to alter a cart's destination in accordance with some global service policy. The presence of the symbol $\perp$ in *bagid* and *source* indicates the absence of a bag, henceforth called a null bag. The cart has a valid destination, *dest*, at all times, the next station to be serviced for loading or unloading. When the cart is full, the destination is that of the bag. In all other cases it refers to the next loader to be serviced. Finally, with 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, the cart requires information about whether it arrived at a particular station. This is made known through the variable *stationid*. By comparing the station identifier with its destination, the cart is able to determine if it must wait at a station or move on.

The variable *stationid* is declared in the **always** section as a **reflective** variable, i.e., a variable whose value is determined by the context in which the program finds itself at the time. The variables *bagid, source,* and *dest* are declared as coop variables by using the keyword **shared** in the **declare** section. Their values are affected by the interactions with loaders and unloaders. The resulting program assumes the following form which, being parameterized by a cart identifier, we treat as a program type definition

```
Program Cart(i) at λ
    declare
            bagid, source, dest : shared integer
    initially
            λ > 0
        []  bagid, source, dest = ⊥, ⊥, FirstLoadingStation(i)
    always
            smallstep    = 0.1
        []  largestep    = 0.2
        []  stationid :  reflective integer
    assign
            λ := λ + smallstep    if stationid ≠ dest
        []  λ := λ + largestep    if stationid ≠ dest
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 condition that it must stop at its current destination. 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.

We now turn our attention to the loaders and unloaders. A loader has one action, the transfer of the first-in-line bag to a waiting cart, along with the source and destination for that bag—waiting bags are represented by a queue of bag identifiers called *cargo*. If there is no first-in-line bag, the loader assigns a null bag and uses the *NextLoader* function to tell the cart which loader it should visit next. The loader acts only in the presence of a cart destined for it. As explained later, this will be possible because the *dest* variable present on the cart will become visible to the loader when the *Cart(i).dest* and *Loader(j).dest* become one and the same variable.

Program Loader(i) **at** λ
  declare
    bagid, source, dest : **shared** integer
  [] cargo : queue of integer
  assign
    cargo, bagid, source, dest
      := cargo.tail, cargo.head, i, *Destination*(cargo.head) **if** dest = i ∧ cargo ≠ ϕ
      ~ cargo, ⊥, ⊥, *NextLoader*(i) **if** dest = i ∧ cargo = ϕ
end

Fig. 4 shows a possible implementation of the transient variable sharing. A cart matching the loader's identification stops at the loader. A piece of luggage is moved onto the cart and its label is scanned for its identifier and destination. This information together with the identification of the current loader is shared with the cart. The latter, detecting a new destination value, terminates the sharing protocol and starts moving. The formal specification for this interaction is given later in this section.

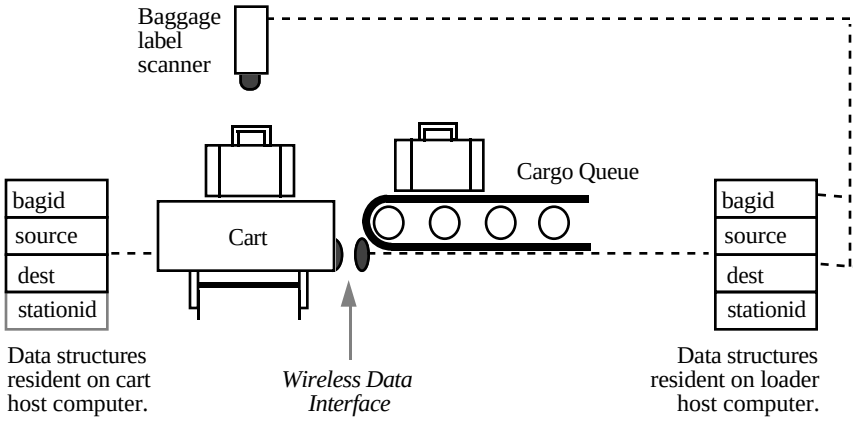

Fig. 4. *A possible realization of the transient data sharing between a cart and a loader.*

Each unloader is also limited to a single action, the transfer of baggage from a waiting cart to its own outgoing queue *cargo*. This transfer is accomplished by an assignment to the shared variable *bagid.* By assignment to other shared variables, the action also supplies a new destination to the cart, telling the cart which loader to visit next. Here again the intent is to share the variables *bagid*, *source*, and *dest* in the unloader with the corresponding variables in the cart. The same function *NextLoader* is used here but the argument it evaluates is not the unloader's identifier but the identifier of the place where the bag originated.

Program UnLoader(i) **at** λ
  declare
    bagid, source, dest : **shared** integer
  [] cargo: queue of integer
  assign
    cargo, bagid, source, dest
      := cargo • bagid, ⊥, ⊥, *NextLoader*(source) **if** bagid ≠ ⊥
end

As with the Loader, assignment to the *bagid* variable models the physical action of moving a bag, this time from the cart to the outgoing *cargo* queue.

Next, we introduce the notation used to specify transient sharing. We start with the reflective variable *stationid*. This variable holds the identifier of the loader or unloader with which the cart is co-located. This interaction is specified apart from any program in a separate **Interactions** section. The value is treated as a function definition for proof purposes, since it is well-defined given the other state variables (including locations) of various components of the system. Here *i* ranges over all cart identifiers and *j* ranges over all loader and unloader identifiers.

```
⟨ ∀ i, j ::
    Cart(i).stationid  = j  if   Cart(i).λ = Loader(j).λ ∨ Cart(i).λ = UnLoader(j).λ
                            ~ ⊥ otherwise
⟩
```

When no loader or unloader is present *stationid* takes on the value ⊥. In general, the value of a 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.

The coop variables *bagid*, *source*, and *dest* are shared between a cart and either a loader or an unloader. The manner in which one can specify the interactions involving the *bagid* variable is shown below. As with the reflective variables, these coop interactions appear apart from the programs in a separate **Interactions** section.

```
⟨ ∀ i, j : 0 ≤ i ≤ NCarts  ∧  0 ≤ j ≤ NLoaders ::
    Cart(i).bagid ≈ Loader(j).bagid
        when       Cart(i).λ = Loader(j).λ ∧ Cart(i).dest = j
        engage     Cart(i).bagid
        disengage  current || current
⟩
```

The same pattern is shared by all the interactions associated with the other coop variables, whether they involve loaders or unloaders. For this reason we explain in detail only the interaction above.

The coop interactions specify not only the condition under which sharing takes place (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 (the **disengage** clause in which **current** is used to refer to the common value just prior to disengagement). For instance, the interaction specification appearing above states that *Cart*(*i*).*bagid* should be shared with *Loader*(*j*).*bagid* whenever cart *i* and loader *j* are at the same location and the loader happens to be the current destination of the cart. The newly created shared variable takes on the value of *Cart*(*i*).*bagid*, and when the transient period of sharing is over (after the loader performs its assignment statement, changing the value of *dest* and falsifying the **when** clause), each copy of the formerly shared variable retains the value it had when shared.

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 or filesystem. Our experience to date indicates that transient data sharing can contribute significantly to decoupling among the components of a mobile system. 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. Loaders and unloaders can freely change data representation and behavior as long as the same basic information is communicated to the cart through minor changes in the **Interactions** section; actually, a 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.

4. 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. For illustration purposes we turn again to the baggage system. Fig. 5 shows an intersection between two segments marked 1 and 2. Two carts, also labeled 1 and 2, are heading for the intersection or crossing. To avoid the possibility of collision, we propose to synchronize the movement of the two carts by having the cart closest to the intersection move faster.

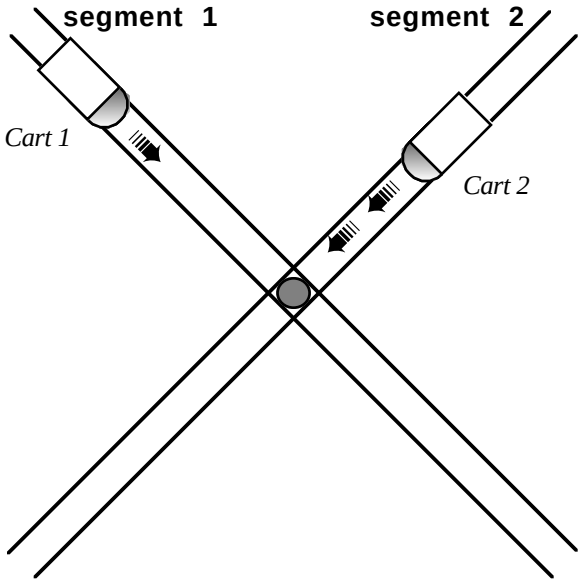

Fig. 5. *The cart closest to the intersection speeds up to avoid collision.*

Let us assume that the low speed of 0.1 units is the standard except in the situation described above. A mechanism must be found to inhibit the fast speed in all other circumstances, to assure that carts cannot approach the crossing at the same speed, and to force two carts that are on crossing segments to synchronize their movements. Our notation for transient synchronization facilitates easy specification of all these requirements.

The notation is very similar to that for transient variable sharing, except that now the objects that interact are statements instead of variables. Because of this we need to name statements in the same way that we name variables:

```
Program Cart(i) at λ
    declare
            bagid, source, dest : shared integer
    initially
            λ > 0
        []  bagid, source, dest = ⊥, ⊥, FirstLoadingStation(i)
    always
            smallstep     = 0.1
        []  largestep     = 0.2
        []  stationid :   reflective integer
    assign
            slow ::  λ := λ + smallstep    if stationid ≠ dest
        []  fast ::  λ := λ + largestep    if stationid ≠ dest
end
```

As in the case of sharing, synchronization is captured in the **Interactions** section of the system specification. To prohibit the use of fast speed outside crossings (i.e., on regular segments), and to prevent a cart from skipping the first interval of the next segment, we specify an interaction of the form

```
⟨ ∀ i ::
    inhibit Cart(i).fast
        when   RegularSegment(Cart(i).λ) ∨ RegularSegment(Cart(i).λ + largestep)
⟩
```

where the function *RegularSegment* returns true for any location not on a segment involved in a crossing. The operational semantics of the **inhibit** construct are such that the affected statement is inhibited from executing whenever the **when** predicate is true. This could be accomplished through the use of **if** clauses except for the fact 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. The result is a greater degree of modularity—the interface constraints are isolated to this one section and are removed from the code of the individual programs. The inhibit mechanism may also be used to restrict two carts from ever sharing the same segment:

```
⟨ ∀ i, j : i ≠ j ::
    inhibit Cart(i).slow
        when   Segment(Cart(i).λ + smallstep) = Segment(Cart(j).λ)

    inhibit Cart(i).fast
        when   Segment(Cart(i).λ + largestep) = Segment(Cart(j).λ)
⟩
```

When two carts are present at a crossing, the interleaving model assures us that one of them ends up closer to the intersection. It is at that point that we need be concerned with the synchronization. A new construct, called **coselect** and denoted by "||" is introduced to the **Interactions** section

```
⟨ ∀ i, j : i ≠ j ::
        Cart(i).fast || Cart(j).slow,  inhibit Cart(j).fast
        when   CrossingSegments(Cart(i).λ, Cart(j).λ)
                ∧ Distance_to_Crossing(Cart(i).λ) < Distance_to_Crossing(Cart(j).λ)
⟩
```

The function *CrossingSegments* is defined to be true if and only if the two locations passed to it are on segments that intersect and both locations are before the intersection, where "before" is defined according to the direction of movement along the track. The function *Distance_to_Crossing* returns the distance from a given location to the next intersection point. The **when** predicate above controls both the **coselect** and **inhibit** interactions. It is enabled when both carts are on crossing segments and cart *i* is closer to the intersection 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. Thus, cart *j* cannot catch up to cart *i* through some interleaving of statements that would have otherwise been perfectly fair and allowed by the scheduler. 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 prevented 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. Furthermore, **inhibit** interactions can prevent the execution of one member of a pair.

With the introduction of transient synchronizations one can think of the universe of actions as being expanded to include the set of actions resulting from the parallel execution of any pairs of actions present in distinct programs. Not all actions are actually enabled at all times. In the absence of any (synchronization) interactions, the actions corresponding to each individual program statement are enabled—this is the UNITY perspective. The coselection (a symmetric form of superposition) adds and removes, dynamically, actions corresponding to pairs of statements executed in parallel; the idea has its roots in the concept of dynamic synchrony [16] originally introduced in the Swarm model [17]. Similarly, an inhibition can be thought of as removing and restoring, dynamically, actions corresponding to singleton statements.

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 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. A critical evaluation of the practical and algorithmic implications of transient synchronization is currently underway. At this point, its main contribution to the field of formal methods is the ability to extend the concept of superposition in ways that offer dynamism and symmetry and which are meaningful to reasoning about mobility.

5. Proof Logic

UNITY [11] provides a formal framework for proving system properties from the text of a program. Basic safety (expressed as **unless** or **co**) and liveness (expressed as **ensures**) properties are proven by quantifying over the statements of the program text, which in standard UNITY correspond one-to-one with system actions (actual state transitions). These properties can then be reasoned about without reference to the original program text.

For Mobile UNITY, we have attempted in the sections above to generalize the mechanisms of UNITY composition (union and superposition) so as to encompass not only static but also transient program composition. This leads to a style of programming where the actions of the system no longer correspond one-to-one with the statements of the underlying programs, but rather, are made up of statements interacting under the rules given in the **Interactions** section. The rules for transient shared variables essentially specify that every statement in the system has the added side effect of keeping shared variables up-to-date, and the rules for transient synchronization essentially construct new actions (pairs of statements) under certain conditions. Proving basic safety and liveness properties requires now quantification over the set of system actions. Properties thus proven can be used to derive other properties without reference to the original program text or the new action system by employing established UNITY rules of inference.

The proof logic extension to Mobile UNITY presents several technical challenges. First, it is the definition of the Hoare triple for the actions resulting from the transient interactions. Once this is accomplished, proving basic safety properties follows directly from the definition of the **co** operator

$$\langle \forall a :: \{p\}\ a\ \{q\} \rangle$$

which states that any action a executed in a state satisfying p must result in a state satisfying q. Liveness is somewhat more complicated and involves a new definition for **ensures**, one that takes into account the interplay between the transient interactions and the weakly-fair scheduling of statements (not actions!). Second, it is the desire to continue to extricate proofs directly from the program text, as UNITY does. This requires one to favor proof rules whose quantifiers range over the set of statements or appropriate syntactic transformations of the basic program statements. Finally, it is the need to exercise control over the number and the complexity of actions one must consider in the proofs. The price of additional complexity in the verification process should be associated only with the use of transient interactions.

5.1 Notation

The proof rules for Mobile UNITY make reference to statements appearing in the mobile programs, to mechanical transformations of some of the statements, and to syntactic elements present in the **Interactions** section. Generally, we use s and t to denote statements and x and y for variables. Constructs that employ Greek symbols denote syntactic elements present in the **Interactions** section or easily derived from it:

$\iota(s)$
: The predicate constructed as the disjunction of all the predicates from the **inhibit** clauses that name statement s. (**inhibit** s **when** $\iota(s)$)

$\sigma(s, t)$
: The predicate controlling the coselection of statements s and t. ($s \parallel t$ **when** $\sigma(s, t)$)

$\nu(x, y)$
: The predicate controlling the sharing of variables x and y. ($x \approx y$ **when** $\nu(x, y)$)

$\delta_x(x, y)$
: The expression yielding the value for x when disengaging from y. (δ_y is defined similarly)

$\varepsilon(x, y)$
: The expression yielding the engagement value for the x, y sharing interaction.

This notation relies on the assumption that there is at most one clause governing synchronization of each pair of statements, and at most one clause governing sharing of each pair of variables. This condition can be mechanically checked. Also, if there is no clause relating a pair of statements or variables, $\sigma(s,t)$ and $\nu(x,y)$ are assumed to denote the predicate **false**. Similarly, $\iota(s)$ returns **false** if there is no **inhibit** clause for the given statement.

5.2 Basic Safety

In the context of a system resulting from the composition of several mobile programs under the constraints specified by the interactions, each statement has side effects which correspond to assignment to shared variables that are not explicitly referenced by that statement and each statement has the potential of being synchronized with statements from other programs. Let s denote an arbitrary statement and $(s \parallel t)$ denote an arbitrary pair of statements. Let s^* and $(s \parallel t)^*$ denote a mechanical transformation of the respective statement and statement pair, one that captures the potential effects of these constructs not only on the variables that they access explicitly but also on those affected by the sharing interactions. We will return later to the manner in which to build such transformed statements. For the time being we merely assume their existence. In our model, we preserve the weak fairness assumption of UNITY: in an infinite execution, each statement (not action!) is selected infinitely often. The result may be a skip, a singleton, or the synchronous execution of a pair of statements, depending upon which guards present in the interactions hold at the time. A single statement s executes when scheduled and has the effect captured by s^* if (1) it is not inhibited, and (2) for any statement t with which s is coselected under some condition $\sigma(s, t)$, either $\sigma(s, t)$ is untrue or the statement t is inhibited. Furthermore, a pair of statements $(s \parallel t)$ executes synchronously when either s or t is scheduled having the effect $(s \parallel t)^*$ if (1) neither statement is inhibited, and (2) the condition $\sigma(s, t)$ controlling the coselection is true. Thus, the following UNITY program is operationally equivalent to the result of composing some set of programs P_1 to P_N under transient interaction:

$$\langle [] \; s, n : s \text{ in } P_n :: s^* \;\; \textbf{if } \neg\iota(s) \wedge \langle \forall t :: \neg\sigma(s, t) \vee i(t) \rangle \rangle$$
$$[]$$
$$\langle [] \; s, t, n, m : s \text{ in } P_n \wedge t \text{ in } P_m \wedge n \neq m :: (s \parallel t)^* \;\; \textbf{if } \sigma(s, t) \wedge \neg\iota(s) \wedge \neg\iota(t) \rangle$$

It is important to note that in this operational model coselection is not transitive, thus one cannot execute more than two statements simultaneously. If one statement is coselected explicitly with two other statements, the two coselections are viewed as distinct. To allow arbitrary numbers of statements to execute synchronously would cause an exponential growth of our proof obligations as the size of programs increases. Because the constraint that coselections be pairwise cannot be enforced syntactically, this operational model will actually enable us to prove the pairwise restriction as a semantic property of the composed system.

For a given state transition system, p **co** q states that any action initiated in a state satisfying p terminates in a state satisfying q. To prove this, one must enumerate all possible actions a of the system and prove that

$$\langle \forall a :: p \Rightarrow \text{wp}(a, q) \rangle$$

where *wp* is the weakest precondition predicate transformer. In this case the actions are transformed statements s^* and $(s \parallel t)^*$ guarded as shown above. Using standard *wp*-calculus this results in the definition

$$p \textbf{ co } q \;\equiv\; \mathit{Single}(p, q) \wedge \mathit{Double}(p, q)$$

where

$$\mathit{Single}\,(p, q) \equiv \quad \langle \forall s :: \neg\iota(s) \wedge \langle \forall t :: \neg\sigma(s, t) \vee \iota(t) \rangle \wedge p \;\Rightarrow\; \text{wp}(s^*, q) \rangle$$

$$\mathit{Double}(p, q) \equiv \quad \langle \forall s, t :: \sigma(s, t) \wedge \neg\iota(s) \wedge \neg\iota(t) \wedge p \;\Rightarrow\; \text{wp}((s \parallel t)^*, q) \rangle$$

$\mathit{Single}(p, q)$ states that every singleton action executed in a state satisfying p terminates in a state satisfying q while $\mathit{Double}(p, q)$ states that every possible pair of coselected statements executed in a state satisfying p terminates in a state satisfying q. It is important to note that the quantifications range over the original statements of the underlying programs.

We turn next to considering the mechanical transformation from w to w^*, where w denotes singleton statements such as s or pairs such as $(s \parallel t)$. First, we characterize the side effects due to updating shared variables during normal assignment, ignoring engagement and disengagement. These can be given by transforming statement w into w^{α}

$$w^{\alpha} \equiv \quad w \;\parallel\; \langle \parallel x, y : x \in \text{lhs}(w) :: w[y / x] \textbf{ if } \nu(x, y) \rangle$$

where the notation $w[y / x]$ represents the syntactical replacement of all occurrences of variable x with variable y. Second, w^* is defined as a sequence of actions that first check the value of each sharing predicate, execute w^α, and finally disengage or engage variables as necessary.

$$\begin{aligned} w^* \equiv\ & \langle \| \ x, y :: \rho_{xy} := \nu(x, y) \rangle; \\ & w^\alpha ; \\ & \langle ; \ x, y : \neg\nu(x, y) \wedge \rho_{xy} :: x, y := \delta_x(x, y), \delta_y(x, y) \rangle; \\ & \langle ; \ x, y : \nu(x, y) \wedge \neg\rho_{xy} :: x, y := \varepsilon(x, y), \varepsilon(x, y) \rangle \end{aligned}$$

The ";" quantifier means to execute the assignments in some fixed sequence. The variables ρ_{xy} do not appear anywhere in the original program. Operationally, this is the sequence of actions where we first evaluate all the sharing conditions, store their current state in temporary variables, perform the action w^α, and finally perform disengagement and engagement where necessary. This allows us to apply the *wp* semantics to a known sequence of actions.

As with transient synchronization, this operational model is well-defined even for cases where the interaction is not pairwise. In cases where a given variable (say x) is shared with two other variables (say y and z), any assignment to x would propagate to y and z, but an assignment to y would propagate only to x. Thus, transient sharing follows a "nearest neighbors" semantics where only variables that are directly named as being shared with x are updated when an assignment to x is made. The engagement and disengagement semantics are also well-defined in these cases. The third and fourth lines of the formula use the ";" quantifier rather than the "||" quantifier. This is to avoid undefined behavior in the case of non-pairwise sharing. A problem would arise if a given variable (say x) is shared with two other variables (say y and z), and both sharing conditions ($x \approx y$ and $x \approx z$) are disabled simultaneously. Each sharing clause would specify its own disengagement value, and a simultaneous assignment would have undefined semantics.

Once we have a definition for the basic safety properties of the new action system, we can apply all of the inference rules from standard UNITY to derive more complicated properties. For example, an invariant can be expressed as:

$$\textbf{invariant}\ p \ \equiv\ \text{Init} \Rightarrow p \ \wedge \ p \ \textbf{co}\ p$$

where *Init* is a predicate that constrains the initial conditions of the program. Other safety and progress properties can be specified as in UNITY.

5.3 Well-Formed Programs

In UNITY, multiple simultaneous assignments to the same variable are prohibited and the malformed programs can be detected via a simple syntactic check. These kinds of restrictions can no longer by checked syntactically in Mobile UNITY. This is why, it is important for the semantics of each action to be well-defined even in the case of non-pairwise sharing and synchronization. This requirement is the source of much of the complexity of the proof rules given above. Disjointness of the interaction conditions becomes a system property that must be proven. Once we have proven disjointness, however, a lot of the complexity in the proof rules disappears. For instance, we know that only one disengagement or engagement value will be assigned to each variable (eliminating the need for the complicated sequential assignment). Disjointness is a safety property that can be expressed as

$$\begin{aligned} \textit{Disjointness} \ \equiv\ & \langle \forall \ x, y, z :: \neg \ (\nu(x, y) \wedge \nu(x, z)) \rangle \\ \wedge\ & \langle \forall \ s, t, u :: \neg \ (\sigma(s, t) \wedge \sigma(s, u)) \rangle \end{aligned}$$

This can be proven with the aid of a sufficiently strong invariant. It states that each variable x is shared with at most one other variable, and each statement s is synchronized with at most one other statement. Once this has been proven, we know that at most one engagement and disengagement of each variable will take place after any single action. This simplifies the application of the *wp* semantics to the statement sequence simulating engagement and disengagement.

As in UNITY, it is also important to avoid undefined behavior in the case of simultaneous assignment to a shared variable. Invalid code would result if two synchronized actions s and t assigned different values to variables x and y while x and y were shared. For this reason, we rule out assignment to an even potentially shared variable inside a potentially synchronized statement:

$$\textit{Noassign} \equiv \langle \forall \ s, t, x, y : \tau(s, t) \wedge \pi(x, y) :: x \notin \text{lhs}(s) \wedge x \notin \text{lhs}(t) \rangle$$

where

$\tau(s, t)$ is **true** if there exists a coselection clause referencing s and t, **false** otherwise.

and

$\pi(x, y)$ is **true** if there exists a sharing clause referencing x and y, **false** otherwise.

5.4 Progress

Progress properties follow from a proper redefinition of the UNITY **ensures** relation. The added complexity is the result of the need to guarantee that weak fairness does actually result in actual progress, i.e., when the right statement is eventually selected for execution it will have the desired effect alone or as part of a coselection. Formally, this can be expressed as

$$
\begin{aligned}
p \textbf{ ensures } q \;\equiv\; & (p \wedge \neg q) \textbf{ co } (p \vee q) \\
& \wedge \langle \exists\ s : (p \wedge \neg q) \Rightarrow \neg i(s) :: \\
& \qquad ((p \wedge \neg q) \Rightarrow \neg\langle \forall\ t :: \neg\sigma(s, t) \vee i(t)\rangle \vee \mathrm{wp}(s^*, q)) \\
& \qquad \wedge \langle \forall\ t :: (p \wedge \neg q) \Rightarrow \neg(\sigma(s, t) \wedge \neg i(s) \wedge \neg i(t)) \vee \mathrm{wp}((s \parallel t)^*, q)\rangle\rangle
\end{aligned}
$$

which states that no statement falsifies p unless q is established and, in addition, there exists a statement s which is un-inhibited in every state satisfying $(p \wedge \neg q)$, and for which the following proof obligations hold: (1) $(p \wedge \neg q)$ implies that s cannot execute alone or that s, when executed alone, does establish q, and (2) for every other statement t, $(p \wedge \neg q)$ implies that t does not execute simultaneously with s, or that s, when executed simultaneously with t, does establish q.

This progress rule may seem cumbersome, but in practice, the quantifications over t contain only those statements that could be coselected with s, usually a small number. For each of these statements, the rule provides for two alternatives: prove that the statement does not execute with s, or show that such an execution establishes q.

6. Sample Proofs

In this section we illustrate the manner in which proofs about mobile systems can be constructed. We continue to use the baggage delivery program introduced in the earlier sections. Proof outlines are provided for several program properties: pairwise interaction, collision avoidance, and bag delivery. The complete text of the program is presented first.

6.1 Program

A complete system consists of definitions for prototypical programs and interactions plus a list of component and interaction instances that make up the system. The programs and interactions given below match the ones given in Sections 3 and 4. The *Cart* module holds variables representing a piece of baggage and is responsible for movement along the track. The *Loader* module sits at a fixed location and has one assignment statement that models the transfer of a bag onto a waiting cart. Appropriate use of transient sharing (given by the *ShareLoader* and *ShareUnLoader* interactions) allows this assignment to affect the variables of the waiting cart. The *UnLoader* module performs a similar assignment to unload baggage from the cart. The interaction *UpdateStation* keeps the reflective variable *stationid* up to date for each cart so as to force the cart to wait when present at its current destination. Finally, the interactions *StopCollide* and *SynchCrossing* make use of the transient synchronization mechanism to coordinate movement among carts. The former prevents a cart from entering an already occupied next segment while the latter avoids collisions when carts are present on overlapping segments, i.e., pass through the same intersection. The three parts of the overall system are visible below:

Program definitions

```
Program Cart(i) at λ
    declare
            bagid, source, dest : shared integer
    always
            smallstep     = 0.1
        []  largestep = 0.2
        []  stationid : reflective integer
    assign
    slow ::     λ := λ + smallstep      if stationid ≠ dest
    fast :: []  λ := λ + largestep      if stationid ≠ dest
```

end

Program Loader(i) **at** λ
  declare
    bagid, source, dest : **shared** integer
  [] cargo : queue of integer
  assign
    cargo, bagid, source, destination
      := cargo.tail, cargo.head, i, *Destination*(cargo.head) **if** dest = i ∧ cargo ≠ ϕ
      ~ cargo, ⊥, ⊥, *NextLoader*(i) **if** dest = i ∧ cargo = ϕ
end

Program UnLoader(i) **at** λ
  declare
    bagid, source, dest : **shared** integer
  [] cargo : queue of integer
  assign
    cargo, bagid, source, dest := cargo • bagid, ⊥, ⊥, *NextLoader*(source) **if** bagid ≠ ⊥
end

Interaction definitions

ShareLoader(i, j, α) ≡
  Cart(i).α ≈ Loader(j).α
    when Cart(i).λ = Loader(j).λ ∧ Cart(i).dest = j
    engage Cart(i).α
    disengage current || **current**

ShareUnLoader(i, j, α) ≡
  Cart(i).α ≈ UnLoader(j).α
    when Cart(i).λ = UnLoader(j).λ ∧ Cart(i).dest = j
    engage Cart(i).α
    disengage current || **current**

UpdateStation(i, j) ≡
  Cart(i).stationid = j **if** Cart(i).λ = Loader(j).λ ∨ Cart(i).λ = UnLoader(j).λ
    ~ ⊥ **otherwise**

StopCollide(i, j) ≡
  inhibit Cart(i).slow
    when *Segment*(Cart(i).λ + smallstep) = *Segment*(Cart(j).λ)
  inhibit Cart(i).fast
    when *Segment*(Cart(i).λ + largestep) = *Segment*(Cart(j).λ)

SynchCrossing(i, j) ≡
  Cart(i).fast || Cart(j).slow, **inhibit** Cart(j).fast
    when *CrossingSegments*(Cart(i).λ, Cart(j).λ)
      ∧ *Distance_to_Crossing*(Cart(i).λ) < *Distance_to_Crossing*(Cart(j).λ)

SlowRegular(i) ≡
  inhibit Cart(i).fast
    when *RegularSegment*(Cart(i).λ) ∨ *RegularSegment*(Cart(i).λ + largestep)

Components

⟨[] i : 0 ≤ i ≤ NCarts :: Cart(i) **at** *InitialCartPosition*(i)⟩
[] ⟨[] i : i∈Lids :: Loader(i) **at** *InitialLoaderPosition*(i)⟩

[]	⟨[]	i :	i∈ ULids ::	UnLoader(i) **at** *InitialUnLoaderPosition*(i)⟩
[]	⟨[]	i, j :	0 ≤ i ≤ NCarts ∧ j∈ Lids ::	ShareLoader(i, j, bagid), ShareLoader(i, j, source), ShareLoader(i, j, dest)⟩
[]	⟨[]	i, j :	0 ≤ i ≤ NCarts ∧ j∈ ULids ::	ShareUnLoader(i, j, bagid), ShareUnLoader(i, j, source), ShareUnLoader(i, j, dest)⟩
[]	⟨[]	i, j :	0 ≤ i ≤ NCarts ∧ 0 ≤ j ≤ NCarts ∧ i ≠ j ::	StopCollide(i, j), SynchCrossing(i, j)⟩
[]	⟨[]	i, j :	0 ≤ i ≤ NCarts ::	SlowRegular(i)⟩
[]	⟨[]	i, j :	0 ≤ i ≤ NCarts ∧ (j∈ Lids ∨ j∈ ULids) ::	UpdateStation(i, j)⟩

The **Components** section defines the initial locations of both mobile and stationary components. The declarations are quantified over the sets of identifiers for carts ($0 \le i \le NCarts$), loaders (*Lids*), and unloaders (*ULids*). In addition, all interactions in effect are explicitly listed. The system description is complete except for the initialization section and the definition of an assortment of functions that appear italicized above. For the sake of brevity, we omit these aspects of the program and provide the reader with a list of assumptions the missing definitions are meant to enforce:

Tracks

- **T 1** The track layout is fixed and divided into *NSegments* track segments.
- **T 2** The layout configuration is circular with one segment connected to the next.
- **T 3** Pairs of segments can intersect to form crossings—the crossing point is the center of the two segments.
- **T 4** No two crossings are immediately adjacent.

Space

- **L 1** The space is linear with each segment labeled by successive integers from *0* to (*NSegments* - *1*).
- **L 2** Each track segment corresponds to a half-open interval of ten locations, closed on the left.

Movement

- **M 1** A single move is shorter than the length of a whole segment, actually 0.1 or 0.2 units.
- **M 2** A slow move (*smallstep*) is shorter than a fast move (*largestep*).

Carts

- **C 1** There are fewer carts than track segments (*NCarts* < *NSegments*).
- **C 2** Each cart has a unique identifier *i* in the range $0 \le i <$ *NCarts*.
- **C 3** Initially, no carts are located on segments that cross.
- **C 4** Initially, no two carts are located on the same segment.
- **C 5** Initially, all carts are empty and have destinations that are valid loader identifiers from the set *Lids*.

Bags

- **B 1** Initially, each bag identifier *i* in the range $0 \le i <$ *NBags* appears in at most one loader's queue.
- **B 2** The function *Destination*(*i*) maps bag identifiers to valid unloader identifiers from the set *ULids*.

Loaders and UnLoaders

- **U 1** The sets of loader identifiers and unloader identifiers are constant and disjoint.
- **U 2** Loaders and unloaders have fixed locations.
- **U 3** No loaders or unloaders appear on segments that cross.
- **U 4** At most one loader or unloader appears on any given segment.
- **U 5** The function *NextLoader*(*i*) defines a cycle among loader identifiers.
- **U 6** There are at least two loaders.

U 7 Initially, all unloaders' queues are empty.

6.2 Pairwise Interaction

The first property we prove is the fact that all program interactions are actually pairwise. To show this we prove a stronger condition: no two carts are ever present on the same segment. If this property holds, two carts can interact only when they are on a crossing at which time no loaders or unloaders are present on either of the two segments involved in crossing. Furthermore, a cart that is not on a crossing can interact with no other carts and only with one loader or unloader at a time by the assumptions we made about the placement of loaders and unloaders.

Formally, we wish to prove:

$$\textbf{invariant}\ i \neq j \Rightarrow \text{Segment}(\text{Cart}(i).\lambda) \neq \text{Segment}(\text{Cart}(j).\lambda) \qquad \text{(PI)}$$

The reader is reminded that all free variables are assumed to be universally quantified, by convention. It is easy to see that this property holds initially. What remains to be shown is the fact that the property is preserved by the execution of either singleton statements or pairs, under the appropriate circumstances outlined in the proof logic rules. However, several simple observations can significantly reduce the magnitude of the verification task. We first note that the only statements that can affect the variables referenced in the invariant are the movement statements in *Cart(i)* and *Cart(j)* and that the variables are never shared. Without loss of generality, we can concentrate on the movement statements of *Cart(i)*. There are three cases to consider. They correspond to statements of *Cart(i)* executing alone, in synchrony with statements of *Cart(j)*, or in synchrony with statements of other cart programs, say *Cart(k)* where $k \neq j$. Actually, the first and the third cases collapse into one because the presence of statements from *Cart(k)* does not affect the weakest precondition with respect to the predicate appearing in (PI), call it *Exclusive(i,j)*.

A proof which initially appeared to be quite complex is now reduced to two simpler lemmas. First, we show that the invariant is not violated by the execution of statements of *Cart(i)* when not inhibited and not synchronized with any other statement. Here we only consider synchronization with *Cart(j).fast*, because synchronization with other statements is either impossible (no synchronization interaction is specified) or does not affect the *wp* calculation for the reasons given above. The proofs for the slow and the fast movements are similar and assume the form below (shown for the slow movement case only):

$$\begin{aligned} & \text{Exclusive}(i,j) \wedge \neg\iota(\text{Cart}(i).\text{slow}) \wedge \neg\sigma(\text{Cart}(i).\text{slow}, \text{Cart}(j).\text{fast}) \\ \Rightarrow\ & \text{wp}(\text{Cart}(i).\text{slow}, \text{Exclusive}(i,j)) \end{aligned}$$

which translates to

$$\begin{aligned} & \text{Exclusive}(i,j) \wedge \langle \forall\ k : k \neq i :: \text{Segment}(\text{Cart}(i).\lambda+\text{smallstep}) \neq \text{Segment}(\text{Cart}(k).\lambda)\rangle \\ & \wedge \neg(\text{CrossingSegments}(\text{Cart}(i).\lambda, \text{Cart}(j).\lambda) \wedge \text{dist}_j < \text{dist}_i) \\ \Rightarrow\ & (\text{Cart}(i).\text{stationid} \neq \text{Cart}(i).\text{dest} \Rightarrow \text{Segment}(\text{Cart}(i).\lambda+\text{smallstep}) \neq \text{Segment}(\text{Cart}(j).\lambda)) \\ & \wedge (\text{Cart}(i).\text{stationid} = \text{Cart}(i).\text{dest} \Rightarrow \text{Segment}(\text{Cart}(i).\lambda) \neq \text{Segment}(\text{Cart}(j).\lambda)) \end{aligned}$$

where $dist_k$ = *Distance_to_Crossing(Cart(k).λ)*. This, in turn, reduces to proving:

$$\begin{aligned} & \text{Exclusive}(i,j) \wedge \langle \forall\ k : k \neq i :: \text{Segment}(\text{Cart}(i).\lambda+\text{smallstep}) \neq \text{Segment}(\text{Cart}(k).\lambda)\rangle \\ \Rightarrow\ & \text{Segment}(\text{Cart}(i).\lambda+\text{smallstep}) \neq \text{Segment}(\text{Cart}(j).\lambda) \end{aligned}$$

Second, we show invariance under the execution of pairs of statements from *Cart(i)* and *Cart(j)*. The only viable combination is one in which one cart moves fast and the other moves slow. Because of the symmetry, it suffices to prove only one of the two cases:

$$\begin{aligned} & \text{Exclusive}(i,j) \wedge \sigma(\text{Cart}(i).\text{fast}, \text{Cart}(j).\text{slow}) \wedge \neg\iota(\text{Cart}(i).\text{fast}) \wedge \neg\iota(\text{Cart}(j).\text{slow}) \\ \Rightarrow\ & \text{wp}((\text{Cart}(i).\text{fast} \parallel \text{Cart}(j).\text{slow}), \text{Exclusive}(i,j)) \end{aligned}$$

which translates into

$$
\begin{aligned}
& \text{Exclusive(i,j)} \wedge \text{CrossingSegments}(\text{Cart(i).}\lambda, \text{Cart(j).}\lambda) \wedge \text{dist}_i < \text{dist}_j \\
& \wedge \langle \forall\ k : k \neq i :: \text{Segment}(\text{Cart(i).}\lambda\text{+largestep}) \neq \text{Segment}(\text{Cart(k).}\lambda) \rangle \\
& \wedge \neg\text{RegularSegment}(\text{Cart(i).}\lambda) \wedge \neg\text{RegularSegment}(\text{Cart(i).}\lambda + \text{largestep}) \\
& \wedge \langle \forall\ k : k \neq j :: \text{Segment}(\text{Cart(i).}\lambda\text{+smallstep}) \neq \text{Segment}(\text{Cart(k).}\lambda) \rangle \\
\Rightarrow\ & (\text{Cart(i).stationid} \neq \text{Cart(i).dest} \wedge \text{Cart(j).stationid} \neq \text{Cart(j).dest} \\
& \quad \Rightarrow\ \text{Segment}(\text{Cart(i).}\lambda\text{+largestep}) \neq \text{Segment}(\text{Cart(j).}\lambda\text{+smallstep})) \\
& \wedge (\text{Cart(i).stationid} \neq \text{Cart(i).dest} \wedge \text{Cart(j).stationid} = \text{Cart(j).dest} \\
& \quad \Rightarrow\ \text{Segment}(\text{Cart(i).}\lambda\text{+largestep}) \neq \text{Segment}(\text{Cart(j).}\lambda)) \\
& \wedge (\text{Cart(i).stationid} = \text{Cart(i).dest} \wedge \text{Cart(j).stationid} \neq \text{Cart(j).dest} \\
& \quad \Rightarrow\ \text{Segment}(\text{Cart(i).}\lambda) \neq \text{Segment}(\text{Cart(j).}\lambda\text{+smallstep})) \\
& \wedge (\text{Cart(i).stationid} = \text{Cart(i).dest} \wedge \text{Cart(j).stationid} = \text{Cart(j).dest} \\
& \quad \Rightarrow\ \text{Segment}(\text{Cart(i).}\lambda) \neq \text{Segment}(\text{Cart(j).}\lambda))
\end{aligned}
$$

which, finally, reduces to showing

$$
\begin{aligned}
& \text{Exclusive(i,j)} \wedge \text{CrossingSegments}(\text{Cart(i).}\lambda, \text{Cart(j).}\lambda) \\
\Rightarrow\ & \text{Segment}(\text{Cart(i).}\lambda\text{+largestep}) \neq \text{Segment}(\text{Cart(j).}\lambda\text{+smallstep})
\end{aligned}
$$

This property holds as a consequence of the unidirectionality of the track layout. The two segments involved in a crossing can not be followed by the same single segment.

6.3 Collision Avoidance

The second property we want to prove deals with the synchronized movement through the crossings designed to ensure that the two carts do not collide. As explained earlier, the cart nearest the intersection point moves twice as fast than the one farther away. The carts move simultaneously until one of them clears the intersection point assumed to be at the center of the crossing track segments. Formally, the property we need to prove is

$$\textbf{invariant}\ i \neq j \wedge \text{CrossingSegments}(\text{Cart(i).}\lambda, \text{Cart(j).}\lambda) \wedge \neg(\text{dist}_i = \text{dist}_j = 0.5) \Rightarrow (\text{dist}_i \neq \text{dist}_j) \qquad \text{(CA)}$$

which states that, when two carts are present at a crossing and at least one moved past the start of its respective segment, they are at unequal distances from the intersection and thus in no danger to collide.

The invariant holds initially because we assumed that no carts are present on any crossing segments. Verifying that all actions preserve (CA) requires us to consider three cases: (1) the left hand side is false because we do not have two carts at the crossing; (2) two carts are present at the start of the crossing segments and equidistant from the intersection; and (3) two carts are present on the crossing but at unequal distances from the intersection point.

Case (1). The movements of *Cart(i)* and *Cart(j)* are not synchronized and at least one is outside the crossing, e.g., cart *i*. The only action that can make the left hand side of the invariant true is *Cart(i).slow* executed in a state in which cart *i* ready to enter the intersection. The result is case (2) or case (3), if *Cart(j)* is already present at the crossing, and case (1) otherwise.

Case (2). The movement of the two carts is not synchronized yet. As soon as either cart moves, regardless of its speed, case (3) is established.

Case (3). Let's assume that *Cart(i)* is nearest the intersection. The only feasible action is *Cart(i).fast* ‖ *Cart(j).slow*. Cart *j* moves one step closer to the intersection while cart *i* moves two steps closer, maybe passing beyond the intersection point. In either case the invariant is preserved.

It should be noted at this point that a much stronger version of (CA) is actually provable, one that shows that, at the time the fast cart clears the intersection, the slow cart is actually no closer than half way to the intersection. The proof method, however, is basically the same.

6.4 Bag Delivery

In this section we sketch out a proof regarding the eventual delivery of a bag loaded on some cart. To accomplish this we need to consider two properties: a safety property which states that the cart does not lose the bag and a progress property which states that a cart is bound to reach any one of the positions on the track layout. Formally, the two properties take the form

$$\begin{array}{rl} & \mathrm{Cart(i).}\lambda{\neq}\delta \wedge \mathrm{Cart(i).bagid}{=}\beta \wedge \beta{\neq}\perp \wedge \mathrm{Cart(i).dest}{=}\delta \\ \textbf{unless} & \mathrm{Cart(i).}\lambda{=}\delta \wedge \mathrm{Cart(i).bagid}{=}\beta \wedge \mathrm{Cart(i).dest}{=}\delta \end{array} \quad (1)$$

$$\mathrm{Cart(i).}\lambda{=}k \wedge \textit{RegularSegment}(k') \textbf{ leads-to } \mathrm{Cart(i).}\lambda{=}k' \quad (2)$$

Using a UNITY inference rule called PSP (Progress-Safety-Progress)

$$\frac{p \textbf{ leads-to } q,\ r \textbf{ unless } b}{p \wedge r \textbf{ leads-to } (q \wedge r) \vee b}$$

and an appropriate choice for *k* and *k'*, one can deduce the desired result

$$\begin{array}{rl} & \mathrm{Cart(i).}\lambda{\neq}\delta \wedge \mathrm{Cart(i).bagid}{=}\beta \wedge \beta{\neq}\perp \wedge \mathrm{Cart(i).dest}{=}\delta \\ \textbf{leads-to} & \mathrm{Cart(i).}\lambda{=}\delta \wedge \mathrm{Cart(i).bagid}{=}\beta \wedge \mathrm{Cart(i).dest}{=}\delta \end{array}$$

The proof for (1) is a direct consequence of the fact that no statement can affect the bag identifier and the destination of a cart except the statement in the loader, i.e., change can occur only when the destination is reached. The proof of property (2) is more complicated. It requires the repeated application of general disjunction (case analysis) and induction. We start by separating the proof into cases that can be composed through the transitivity of the **leads-to** relation, when *k'* is just ahead of *k* on the same regular segment and when it is not. Without loss of generality, we replace *k'* by a constant *K'* satisfying the property *RegularSegment*(*K'*). Here the predicate *JustBefore*(*k,K'*) is true if and only if the segment *k* immediately preceeds segment *K'*.

$$\mathrm{Cart(i).}\lambda{=}k \wedge \textit{JustBefore}(k,K') \textbf{ leads-to } \mathrm{Cart(i).}\lambda{=}K' \quad (2.1)$$

$$\mathrm{Cart(i).}\lambda{=}k \wedge \neg\textit{JustBefore}(k,K') \textbf{ leads-to } \textit{JustBefore}(\mathrm{Cart(i).}\lambda,K') \quad (2.2)$$

The proof for (2.1) follows directly from some of the lemmas needed to prove (2.2). Regarding (2.2), we simply need to show that eventually *Cart(i)* enters the segment where *K'* is located. This fact combined with a safety property (omitted) which establishes the fact that carts always enter at the start of the segment allows us to establish the RHS of (2.2). The obvious metric to use here is the number of segments between the current location and the destination—the metric is zero if the cart is just before K' but the total number of segments less one if the cart is on the same segment with K' but already passed it. At this point, one simply needs to show that a cart eventually departs the current segment and enters the next one given by the function *NextSegment*.

$$\mathrm{Cart(i).}\lambda{=}k \textbf{ leads-to } \textit{Segment}(\mathrm{Cart(i).}\lambda){=}\textit{NextSegment}(k) \quad (2.2')$$

Two cases need to be considered next, moving along the segment until the last position is reached and moving on to the next segment.

$$\mathrm{Cart(i).}\lambda{=}k \textbf{ leads-to } \mathrm{Cart(i).}\lambda{=}\textit{End\_of\_Segment}(k) \quad (2.2.1)$$

$$\mathrm{Cart(i).}\lambda{=}k \wedge k{=}\textit{End\_of\_Segment}(k) \textbf{ leads-to } \mathrm{Cart(i).}\lambda{=}k{+}\mathrm{smallstep} \quad (2.2.2)$$

The proof of (2.2.1) entails showing that progress is made along the segment with the eventual result being the arrival at the last location of the segment given by the function *End_of_Segment*. The cases we need to distinguish are as follows

(1) Movement along the crossing segments—it is always possible and it is forced to terminate at the end of the segment by the fact that *Cart(i).fast* is inhibited whenever the cart tries to enter the next segment.

(2) Movement along the regular segments when the current cart destination has not been reached—this is always possible and the low speed forces the cart to "pass" through every location of the segment.

(3) Arrival at a destination point—the loader/unloader at the same location eventually changes the cart's destination (an **ensures** property referencing to transiently shared variables) thus establishing the condition (2) above.

The proof for (2.2.2) is complicated by the fact that the move may be blocked by the presence of some other cart on the next segment. A separate progress proof is needed here to show that eventually the blocking cart departs. The proof relies on the fact that a chain of blocking carts is finite since the number of segments is larger than the number of carts, i.e., some cart in front eventually moves. The metric used is the number of carts in a

blocking chain. As the front cart moves to a new segment, the chain decreases by one. We omit the remainder of the proof.

The most striking thing about these proof sketches is the lack of specificity to mobile computing. They look and feel very much like traditional UNITY proofs. This is precisely the point we wanted to make in this paper. The impact of introducing location and transient interactions is felt mostly in the proof of basic **unless** and **ensures** properties. The entire proof infrastructure associated with UNITY is preserved, both in terms of inference rules and proof strategies. In other words, most of the tools needed to reason about mobility are already in place.

7. Conclusions

The starting point for this investigation was the search for a model of mobile computing able to capture the kinds of applications we expect to see emerging in ad-hoc networks. A decoupled and opportunistic style of computing demanded a highly-modular system organization. We accomplished this by separating component implementation from program interfacing concerns—the latter were delegated to the **Interactions** section. Location-sensitive behavior and communication led to explicit representation of location and movement and to the concept of transient interactions controlled by the components' relative positions in space. Transient variable sharing and transient synchronization, however, turned out to be direct generalizations of already established UNITY constructs. This fact enabled us to extend, in a straightforward manner, the existing UNITY proof logic to the verification of mobile computations and allowed us to address the important issue of dependability in the development of mobile applications. The example used in this paper also showed that, through the use of appropriate invariants, proofs can be kept simple even in the presence of mobility. Despite these encouraging results, we view our investigation into Mobile UNITY mostly as a feasibility study with much more work remaining to be done. Future research will be directed towards a more comprehensive assessment of the practical implications of transient interactions and the associated proof logic. Verification of existing mobile applications and protocols, ease of programming, and novel algorithms for use with ad-hoc network applications are some of our most immediate concerns. In the longer range, we seek to find a model that exhibits parsimony and captures the very essence of mobile computing. Research to date supports the conjecture that Mobile UNITY, with its novel ways to interface mobile programs, is a reasonable starting point.

References

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

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

[3] B. R. Badrinath and G. Welling, "Event Delivery Abstractions for Mobile Computing," Rutgers University, New Brunswick, NJ 08903, Technical Report LCSR-TR-242, 1995.

[4] B. D. Noble, M. Price, and M. Satyanarayanan, "A programming interface for application-aware adaptation in mobile computing," *Computing Systems*, vol. 8, no. 4, pp. 345-63, 1995.

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

[6] C. D. Tait and D. Duchamp, "An Efficient Variable Consistency Replicated File Service," *Proceedings of the USENIX File Systems Workshop*, Ann Arbor, MI, pp. 111-126, 1992.

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

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

[9] A. Acharya, B. R. Badrinath, T. Imielinski, and J. C. Navas., "A WWW-based Location-Dependent Information Service for Mobile Clients," *submitted to the Fourth International WWW Conference*, 1995.

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

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

[12] R. Milner, J. Parrow, and D. Walker, "A calculus of mobile processes I," *Information and Computation*, vol. 100, no. 1, pp. 1-40, 1992.

[13] W. D. Clinger, "Foundations of Actor Semantics," MIT Artificial Intelligence Laboratory AI-TR-633, 1981.

[14] L. Lamport, "The temporal logic of actions," *ACM Transactions on Programming Languages and Systems*, vol. 16, no. 3, pp. 872-923, 1994.
[15] M. Abadi and L. Lamport, "An old-fashioned recipe for real-time," in *Lecture Notes in Computer Science*, vol. 600, J. W. d. Bakker, C. Huizing, W. P. Roever, and G. Rosenberg, Eds.: Springer-Verlag, 1991, pp. 1-27.
[16] H. C. Cunningham, G.-C. Roman, and J. Y. Plun, "Assertional Reasoning about Dynamic Systems," in *Parallel Computing: Paradigms and Applications*, A. Y. Zomaya, Ed. London, UK: Chapman & Hall, 1994.
[17] G.-C. Roman and H. C. Cunningham, "Reasoning about Synchronic Groups," in *Research Directions in High-Level Parallel Programming Languages*, vol. 574, *Lecture Notes in Computer Science*, J. P. Banâtre and D. L. Métayer, Eds. New York, NY: Springer-Verlag, 1992, pp. 21-38.