Report Number: WUCS-96-15

1996-01-01

Mobile UNITY Coordination Constructs Applied to Packet Forwarding for Mobile Hosts

Peter J. McCann and Gruia-Catalin Roman

With recent advances in wireless communication technology, mobile computing is an increasingly important area of research. A mobile system is one where independently executing components may migrate through some space during the course of the computation, and where the pattern of connectivity among the components changes as they move in and out of proximity. Mobile UNITY is a language and logic for specifying and reasoning about mobile systems, the components of which must operate in a highly decoupled way. In this paper it is argued that Mobile UNITY contributes to the modular development of system specifications because of the... **Read complete abstract on page 2.**

Mobile UNITY Coordination Constructs Applied to Packet Forwarding for Mobile Hosts

Peter J. McCann and Gruia-Catalin Roman

Complete Abstract:

With recent advances in wireless communication technology, mobile computing is an increasingly important area of research. A mobile system is one where independently executing components may migrate through some space during the course of the computation, and where the pattern of connectivity among the components changes as they move in and out of proximity. Mobile UNITY is a language and logic for specifying and reasoning about mobile systems, the components of which must operate in a highly decoupled way. In this paper it is argued that Mobile UNITY contributes to the modular development of system specifications because of the declarative fashion in which coordination among components is specified. The packet forwarding mechanism at the core of the Mobile IP protocol for routing to mobile hosts is taken as an example. A Mobile UNITY specification of packet forwarding and the mobile system in which it must operate is developed. Mobile hosts are the components that can disconnect from one location in the network and reconnect to another at any point during system execution. Finally, the role of formal program verification in the development of protocols like Mobile IP is discussed.

Mobile UNITY Coordination Constructs Applied to Packet Forwarding for Mobile Hosts

Peter J. McCann
Gruia-Catalin Roman

WUCS-96-15

17 December 1996

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

Mobile UNITY Coordination Constructs Applied to Packet Forwarding for Mobile Hosts

Peter J. McCann
Gruia-Catalin Roman

Abstract

With recent advances in wireless communication technology, mobile computing is an increasingly important area of research. A mobile system is one where independently executing components may migrate through some space during the course of the computation, and where the pattern of connectivity among the components changes as they move in and out of proximity. Mobile UNITY is a language and logic for specifying and reasoning about mobile systems, the components of which must operate in a highly decoupled way. In this paper it is argued that Mobile UNITY contributes to the modular development of system specifications precisely because of the decoupled and declarative fashion in which coordination among components is specified. The packet forwarding mechanism which is at the core of the Mobile IP protocol for routing to mobile hosts is taken as an example. A Mobile UNITY specification of packet forwarding and the mobile system in which it must operate is developed. Mobile hosts are the components that can disconnect from one location in the network and reconnect to another at any point during system execution. Finally, the role of formal program verification in the development of protocols like Mobile IP is discussed.

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

Acknowledgement: This paper is based upon work supported in part by the National Science Foundation under Grant No. CCR-9217751. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation. The authors thank Toni Reiss for the illustrations in figures 1 and 3.

Correspondence: All communications regarding this paper should be addressed to

Dr. Gruia-Catalin Roman	office: (314) 935-6190
Department of Computer Science	secretary: (314) 935-6160
Washington University	fax: (314) 935-7302
Campus Box 1045	
One Brookings Drive	roman@cs.wustl.edu
Saint Louis, MO 63130-4899	http://www.cs.wustl.edu/~roman/

1. Introduction

Mobile computing represents a major point of departure from the traditional distributed computing paradigm. The potentially very large number of independent program 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 for this environment. These services are characterized by more dynamic binding and weaker consistency than traditional distributed applications. For example, the components needed to carry out a service are often not determined until runtime, as in the location-dependent services provided by a mobile web browser [12]. Other work has pointed out the importance of context other than location [9], such as the presence or absence of other components. Weak consistency protocols for filesystems and databases [8, 10, 11] are motivated by the low bandwidth and frequent disconnections typical of a wireless network with mobile nodes. These systems trade consistency for availability under the assumption that in some cases, dealing with the consequences of inconsistencies is cheaper than denying access to a resource.

Some researchers have focused on toolkits and abstractions for building mobile applications. Badrinath and Welling [1] describe a C++ abstraction for delivering events such as bandwidth variations, disconnections, and battery measurements to applications. Noble, Price, and Satyanarayanan [6] 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. Both emphasize the need to present information about connectivity directly to applications, which violates traditional notions of abstractions and encapsulation of the network. Such information is necessary, however, to build applications that behave properly under changing circumstances, such as responding to diminished connectivity by changing to a lower-resolution video stream.

In this paper we focus on new kinds of abstractions for interprocess communication in the mobile setting. Mobile UNITY [5] provides a notation for mobile system components and a coordination language for expressing interactions among the components. Once expressed in our notation, a system can be subjected to rigorous formal verification against a set of requirements expressed as temporal properties of executions. Mobile UNITY is based on the UNITY model of Chandy and Misra [2], with extensions to both the notation and logic to accommodate specification and reasoning about mobile programs. In Mobile UNITY, each program is a unit of mobility and all variables are locally owned. We capture movement by augmenting the program state with a location attribute whose change in value is used to represent motion. The new language supports a declarative style of communication that allows a component program to be written in a modular fashion, without regard to the identities of the other components with which it must later interact. This is accomplished with a novel construct, transient variable sharing. This 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. This implies that a value written to one variable of such a pair must be propagated to the other variable as a side-effect of an assignment. The basic constructs of Mobile UNITY that allow us to express this idea also allow for other coordination constructs, such as transient statement synchronization. However in this paper we deal only with transient sharing, which suffices for the examples presented.

Mobile UNITY is designed to accommodate mobile applications and services that exhibit dynamic reconfiguration and weak consistency, like the ones discussed earlier. Perhaps the most basic service that can be provided in the mobile setting is simple packet routing. The Mobile IP protocol [7] is designed to deliver this service to mobile hosts that are transiently connected to the Internet. In this paper we give a formal description of the packet forwarding mechanism at the core of the Mobile IP protocol and begin to investigate some of its formal properties. Eventually, we hope to formally describe other packet routing algorithms, such as those for ad-hoc networks [4], which provide routing services to a group of mobile hosts that may be completely disconnected from the Internet. Such situations may arise at a conference, for instance, where it would cost too much to install a fixed routing infrastructure for a short-duration event, or in an urban setting after a natural disaster had wiped out the fixed infrastructure. These kinds of networks must rely on the individual hosts for control and routing functions. For now, however, we concentrate on the current version of Mobile IP, which assumes that mobile hosts connect to the Internet at specific points designated by network administrators.

Section 2 gives a simple description of a network that allows hosts to disconnect but not to move to another point of connectivity. This serves as an introduction to the Mobile UNITY notation and offers an example of how mobile, transiently-connected systems may be specified. Section 3 refines the example to model packet forwarding inside the fixed network, which allows mobile nodes to migrate from one point of connectivity to another and to receive packets there. Section 4 shows how UNITY-style program verification is still feasible in Mobile UNITY. Several correctness properties of the packet forwarding system are stated and outlines their proof are discussed—the presentation style is accessible to a broad audience. Finally, conclusions appear in Section 5.

2. A Notation for Mobile Programs

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.

In this paper we use the UNITY [2] 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 first consider a standard UNITY program which models a network. A picture of the network is shown in Fig. 1.

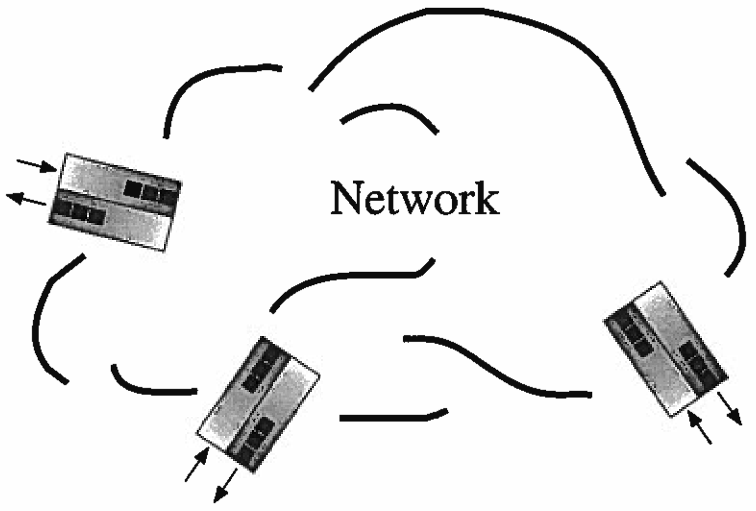

Fig. 1. The simple packet router modeled by the program *network*, below.

The simplified network consists of *NPorts* input queues and *NPorts* output queues. The index of each queue is a network address that can be assigned to some host. The state of the network is completely defined by the contents of the queues. Each queue contains one or more packets (the *msg* type in the program below) which are assumed to be tuples of the form (*A*, *D*), where *A* is the destination address of the packet and *D* is the data contents. The destination address of a message *m* can be accessed by writing *m.A*. Execution of the network corresponds to taking some packet out of an input queue and delivering it to the appropriate output queue. Execution of the network continues forever.

The program below expresses these ideas formally. The **declare** section contains variable declarations, here the two arrays of queues. The **always** section defines convenience functions to be used later in the program text and in proofs. Here it contains the definition of the parameterized function *dest*(*i*) which returns the address of the packet on the head of queue *i*. The **initially** section is a set of state predicates defining the allowed set of initial states. The predicates are separated by the symbol [] which is used as the quantifier in three-part notation[1].

[1] The three-part notation ⟨ **op** quantified_variables : range :: expression ⟩ used throughout the text is defined as follows: The variables from *quantified_variables* take on all possible values permitted by *range*. If *range* 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,

```
program network
  declare
    in, out : array[NPorts] of queue of msg
  always
    dest(i) = in[i].head.A
  initially
    ⟨[] i : 0 ≤ i < NPorts :: in[i] = InitialPort(i)⟩
  assign
    ⟨[] i : 0 ≤ i < NPorts :: out[dest(i)], in[i] := out[dest(i)]•in[i].head, in[i].tail if in[i] ≠ ϕ⟩
end
```

Each initialization predicate makes use of the function *InitialPort(i)*, which simply returns some initial state for input queue *i*. The network will execute as a closed system for now, performing routing only for those packets which are initially in the input queues. The **assign** section contains the "code" for the program. It contains a set of assignment statements, one for each port, each guarded with an **if**. Each assignment is a multiple assignment, that is, each has an equal number of left and right hand sides. When a statement is selected for execution, all right hand sides (and left hand side array indices) are completely evaluated before any assignment takes place. Concurrent execution is modeled as a non-deterministic fair interleaving of the statements from this set. In the assign section, the notation *q.head* is used to refer to the element on the front of queue *q*, and *q.tail* refers to a queue formed by removing the first element of *q*. The operator '•' appends an element to a queue; the expression (*q•e*) is equivalent to the queue *q* with element *e* added to the end. Thus, if input port *i* is non-empty ($in[i] \neq \phi$) the assignment takes the packet from the head of input queue *i* and places it on the end of the output queue given by the packet's destination address.

This model of the network has no interaction with the outside world. The initial conditions place packets on the input links which are routed to the appropriate output link where they remain forever. Once all packets have been routed, the program is said to have reached *Fixed Point*, i.e., no further state changes take place. However, we would like to compose this program with others that model hosts connected to the network. These hosts would insert packets into input queues and remove packets from output queues. In standard UNITY, this composition would be accomplished with statically shared variables. Any two variables with the same name are considered shared, and the relationship is not allowed to change over the course of execution. However, we would like to model the intermittent types of system relationships characteristic of mobile computing. The first step towards this goal is to isolate the namespaces of each program. Thus, in Mobile UNITY variables are not shared unless explicitly allowed to by the circumstances of the current state; this leads to a less tightly coupled system where relationships between components change dynamically. The program *mobile-node* contains two queues, one for input to the node and one for output from the node. The first two statements in the **assign** section consume and generate messages on these links. Messages are generated from the function *NextMessage*(*address*) which is simply assumed to return the next message that the host wants to transmit.

```
program mobile-node(k) at λ
  declare
    in, out : queue of msg
  [] address : integer
  initially
    address = k
  assign
    in := in.tail  if in ≠ ϕ
  [] out := out •NextMessage(address)
  [] λ := Move(λ)
end
```

The factors leading to interaction are likely to include location, so we add this as a distinguished variable λ which models the physical placement of the node in space. This program contains one assignment statement that modifies λ, which uses the function $Move(\lambda)$ to determine the new location. Implicit in our notational conventions is the notion that a program and its variables are co-located and move as a single unit. Assignments to λ are not compiled as assignment statements but instead model physical movement. In this

yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, the value of the three-part expression is the identity element for *op*, e.g., *true* when *op* is $\forall$.

case, the statement simply models the fact that the program may move at some point in its execution, most likely due to a user walking away from the network. Such statements must represent a correct reflection of the physical world, accurate enough to facilitate reasoning about both functional and mobility aspects of the program's behavior. 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 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 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.

To compose this program with the *network* program given earlier, we need to formally declare the components and specify the interactions between them using a specialized abstract coordination language built on top of three primitive structures not discussed in this paper (see [5] for details). In the following system specification, we declare one *network* component, one *mobile-node* component for each port, parameterized by the port number, and one sharing interaction for each mobile node input and output queue.

```
System network-and-nodes
  Components
    network
  [] ⟨[] i : 0 ≤ i < NPorts :: mobile-node(i) at λ_i⟩
  Interactions
    ⟨[] i : 0 ≤ i < NPorts ::
          mobile-node(i).in ≈ network.out[i]
            when       mobile-node(i).λ = λ_i
            engage     mobile-node(i).in
            disengage  current, φ

        [] mobile-node(i).out ≈ network.in[i]
            when       mobile-node(i).λ = λ_i
            engage     network.in[i]
            disengage  φ, current
    ⟩
end
```

Consider the first sharing interaction. It names two variables, *mobile-node*(*i*).*in* and *network.out*[*i*]. The first is a variable from an instance of a parameterized program, while the second is an array element from the fixed program *network*. The two variables are treated as shared **when** the mobile node i is co-located with port i at λ_i. By "shared" we mean that any change to one copy is atomically propagated to the other copy, just as if the variables were statically shared as in standard UNITY. In this system, only node i can be connected to port i (a node has no connectivity when away from its home port). The **engage** expression is used to specify what value the newly shared variable should take when the sharing predicate transitions from *false* to *true*. In this case, we use the value *mobile-node*(*i*).*in* because the shared queue is actually physically located on some network interface hardware present on the mobile node (we assume that the network interface has on board a queue of messages that have not yet been processed by the mobile node). We must also specify what values the shared variables should take when the sharing predicate transitions from *true* to *false*. This is the **disengage** value, which in this case states that the variable *mobile-node*(*i*).*in* should retain the current shared value, while the variable *network.out*[*i*] should be cleared. Packets might be placed on this output queue during the period of disconnection, but they will be dropped by the **engage** specified here once the connection is re-established. We share the mobile node's output queue with the appropriate network input queue in a similar fashion.

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. Changes in the design of individual components often have effects limited only to the definition of the interactions. The same programs work in a multitude of configurations using varying numbers of components. 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.

3. Packet Forwarding in Mobile IP

In the previous section, we modeled the network as a simple packet router, and the hosts were mobile but could only be connected when present at the home location. In this section, we would like to refine this abstract program so that it captures important aspects of the Mobile IP protocol [7]. This is an extension to IP version 4 that attempts to deal with host mobility at a network level by forwarding those packets that arrive at the mobile node's home address to the foreign subnetwork on which the host happens to be. The goal of Mobile IP is to accomplish this without introducing changes to the bulk of the Internet routing fabric or nodes that want to communicate with mobile hosts. In what is expected to be the most common mode of operation, implementation of the protocol requires a *home agent*, which accepts packets on behalf of the mobile node and performs forwarding, a *foreign agent*, which receives the forwarded packets and delivers them to the mobile node, and a *mobile node* that detects movement and initiates the appropriate registration with the home agent so that packets can be forwarded. Fig. 2 illustrates a mobile node away from home and the "tunnel" through which the home agent sends packets. The mobile node is allowed to transmit packets normally as long as it can find an appropriate outgoing router (often the foreign agent itself).

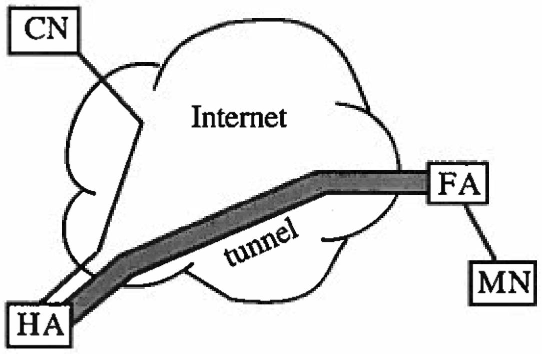

Fig. 2. A high-level picture of the Mobile IP protocol. Packets arriving at the home agent (HA) from some correspondent node (CN) are tunneled using some form of packet encapsulation to the foreign agent (FA), which then delivers them to the mobile node (MN).

In this section, we introduce some modifications to the abstract program presented earlier that let it capture the packet forwarding idea from Mobile IP. To keep the presentation short, the protocol is modeled at a very high level and we make many simplifying assumptions. We make no modifications to the *mobile-node* program; this program corresponds to the interface presented by the operating system to applications. Our modifications will thus be confined to the *network* program and the system declarations. The resulting new system is named *mobile-ip*. Because our modifications are to the *network* program, this is where we are modeling the functionality provided by home and foreign agents. As before, each port *i* is considered to be "at" location λ_i, and a mobile node can be connected only when it is co-located with some port—*any* port in this refined example. The mapping from agents to ports is not defined; each port could correspond to a separate mobility agent or one mobility agent could manage several ports. The function of the agents is completely encapsulated inside the network. Fig. 3 illustrates this away-from-home connectivity.

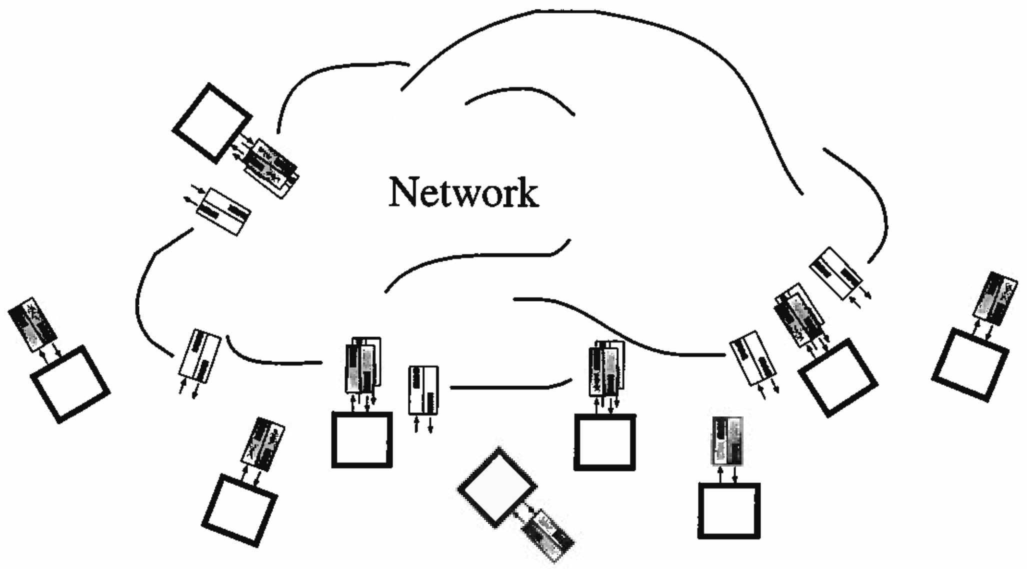

Fig. 3. A network supporting mobile hosts in the style of Mobile IP. The packet forwarding scheme is modeled as a modification to the routing functionality so that packets at any input are routed immediately to the correct output.

Our modified *network* program is shown below. It now contains two additional arrays, *address* and *binding*.

```
program network
  declare
    in, out : array[NPorts] of queue of msg;
    address, binding : array[NPorts] of integer
  always
    dest(i) = binding[in[i].head.A]
  initially
    ⟨[] i : 0 ≤ i < NPorts :: binding[i] = i⟩
  [] ⟨[] i : 0 ≤ i < NPorts :: address[i] = i⟩
  assign
    ⟨[] i : 0 ≤ i < NPorts :: out[dest(i)], in[i] := out[dest(i)]•in[i].head, in[i].tail if in[i] ≠ φ⟩
  [] ⟨[] i : 0 ≤ i < NPorts :: binding[address[i]] := i     if address[i] ≥ 0⟩
end
```

The value in *address*[*i*] is set to the address of the mobile node that is currently connected to port *i*, or -1 if no mobile node is present. The system specification given later will accomplish this. The value in *binding*[*i*] will be set to the forwarding address of node *i*. This will be accomplished by the program *network* itself when it detects a mobile node at some port (using the value of *address*[*i*]). The code for the network now has two sets of quantified actions. The first accomplishes routing and is very similar to the routing action of the previous version, except that it now uses the forwarding address of the destination, instead of the destination itself, to determine the final endpoint of the communication. This models correct execution of the home agent and foreign agent as they collaborate to deliver a packet to the mobile node. The second set of actions establishes a forwarding address for a mobile node when it is connected away from home and also clears this forwarding address when it is re-connected to the home port. This models registration of the mobile node as it moves from subnetwork to subnetwork. As the reader can readily observe, the program required very few modifications to capture the essential nature of the packet forwarding scheme. The basic structure of the program remains the same. We simply modified the routing to account for forwarding, and added one statement to model remote registration.

The system *mobile-ip* is shown below. In it, we rely on interactions given in the system specification to update the *address* array. As before, the system is composed of one *network* program and as many *mobile-nodes* as there are ports.

```
System mobile-ip
  Components
    network
  [] ⟨[] i : 0 ≤ i < NPorts :: mobile-node(i) at λ_i⟩
  Interactions
    ⟨[] i, j : 0 ≤ i < NPorts ∧ 0 ≤ j< NPorts ::
            mobile-node(i).address ≈ network.address[j]
              when        mobile-node(i).λ = λ_j ∧ Alone(λ_j)
              engage      mobile-node(i).address
              disengage   current, -1

          [] mobile-node(i).in ≈ network.out[j]
              when        mobile-node(i).λ = λ_j ∧ Alone(λ_j)
              engage      mobile-node(i).in
              disengage   current, φ

          [] mobile-node(i).out ≈ network.in[j]
              when        mobile-node(i).λ = λ_j ∧ Alone(λ_j)
              engage      network.in[j]
              disengage   φ, current
    ⟩
end
```

The transient sharing conditions now support away-from-home interaction, however, so there are now $O(NPorts \times NPorts)$ interactions. For each pairing of mobile node to port, there are three interactions. The first shares the mobile node's address with the appropriate network port when connected. This models the acquisition of a foreign agent upon entering a new subnetwork. The actual registration with the home agent was modeled by assignment to the *binding* array given previously, based on this address. The second and third interactions are the same as the ones given earlier that share the input and output queues of the current port with the mobile node. As before, the actual state of any queue is considered to reside on the receiving side, as reflected by the **engage** values. Because a mobile node can now be connected to any port, not just the port with the same index, we must be careful that no two nodes are connected to the same port. This would lead to undefined behavior of the queues as both nodes tried to access them. We can think of the locations as corresponding to physical network connections that can accommodate only one node at a time. Therefore, even if two mobile nodes are co-located, neither has connectivity to the port at that location. This is reflected in the semantics of the *Alone* predicate which is true if and only if there is only one node at the given location. This system is an abstract program that ignores many of the details of the Mobile IP protocol. However, it does capture the essence of the protocol at a similar level of abstraction to the single point of connectivity example presented earlier. The modifications did not affect the *mobile-node* program in the least, which is indicative of the amount of decoupling among components provided by our notation.

Throughout the examples, the manner in which we have modeled the links among programs (as shared message queues) is an abstraction from the physical reality of a wired or wireless data link between hardware devices where each receiver maintains a queue of unprocessed messages. In reality, each program has "append-only" access to a remote queue. This restriction is in fact satisfied by the above programs as they are composed here, but our notation does not explicitly protect the queues from remote access. Mobile UNITY is intended to be a general model helpful in the design, modeling, and verification of mobile computations and not a programming language supporting implementation. Consequently, it allows one to model undesirable structures and subject them to analysis.

In the next section we discuss the appropriateness of the level of abstraction presented so far and the role of formal verification in the design process.

4. Verification Strategy and Issues

The brevity of this paper does not allow us to give a complete description of the Mobile UNITY notation or its proof logic, e.g., we omitted the transient action synchronization mechanism among others. Neither can we provide a complete formal verification of the Mobile IP protocol, even at this abstract level. The purpose of this section is to show how one might approach this task and the benefits one can expect from such an exercise. We hope to convince the reader that Mobile UNITY makes the task manageable even for programs involving mobility. Formally, the key to accomplishing this is the fact that Mobile UNITY inherits most of the proof logic of UNITY, along with its inference rules. This is because we were able to encapsulate the impact of mobility to very few low level axioms, e.g., a reformulation of the Hoare triple and of the concept of basic progress.

Typically, a designer starts out with informal expectations for what a protocol is expected to accomplish, and then proposes an operational solution (a program) to meet those goals. The process of protocol verification entails formalizing the correctness criteria and verifying that the proposed program meets these conditions. The program that is verified is usually an abstraction of the actual implementation—it contains fewer details than a running implementation would. This simplifies the verification process and does not adversely affect the correctness of a final implementation, as long as a correctness-preserving mechanical transformation of the abstract program into an implementation can be performed. Of course, the abstract program must contain enough detail so that all interesting and difficult aspects of the protocol are captured within it.

The Mobile UNITY program notation offers a simple means to specify operational solutions, provides a means to verify properties of such programs, and is sufficiently abstract so as to be compatible with many implementation architectures. In formalizing the correctness expectations of a protocol, the UNITY logic can be used to write down properties that will later be proven of a specific program. These properties are called assertions. They are of two kinds, *safety* and *liveness* properties. In the UNITY logic, which is based on a restricted form of temporal logic, an assertion must hold in every state along every execution sequence. This allows us to distance ourselves from reasoning about execution sequences explicitly. Basic properties are derived from the program text and are combined using a battery of inference rules.

Simple program properties are built from state predicates and simple relations on them. A state predicate is constructed using boolean algebra, the names of variables from the program text, quantification, and ordinary relational operators, as in

$$network.address[3] = 5 \wedge network.binding[5] = 3$$

which states that the program variable *network.address*[3] (the third element of the *address* array from program *network*) has the value 5, and the variable *network.binding*[5] (the fifth element of the *binding* array from program *network*) has the value 3. For any given state (an assignment of values to the program variables), this predicate might be *true* or *false*. Those states for which the predicate is true are said to be the set of all states that "satisfy" the predicate. Note that there are many such states that satisfy the above predicate, because it fixes only two of the program variables, while the others can take on any value.

An execution is a sequence of states the program passes through starting from an acceptable initial state. All executions are assumed to be infinte in length and weakly fair. Relations over predicates can be used to express properties of executions. For example, the relation **co** (short for *constrains*)

$$p \ \mathbf{co} \ q$$

asserts that if a state satisfying p occurs anywhere in the execution, the next state must satisfy q. This expression is now a predicate over state sequences (executions) and defines a set of executions that satisfy it. This kind of property is called a *safety* property because if it is not true, there is a finite place in the execution where it breaks down. A safety property states that "bad things do not happen." The **co** relation for two predicates p and q with respect to some program F can be proven by considering all the statements of F and showing that if each is executed in a state satisfying p, it will terminate and leave the program in a state satisfying q. Well known methods from sequential programming can be used to carry out this proof [3]. Invariant properties, those that are true throughout execution, can be expressed simply as

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

which states that the initial conditions satisfy p and also every action preserves p if executed from a state satisfying p.

For example, consider one desirable property of the Mobile-IP protocol, NO-MISDELIVERY, which states that a message *(A,D)* is never delivered to the wrong node, i.e., to a node i different from the address A. We can express this property as a predicate over the input queue for node i, stating that the queue can only contain messages addressed to node i.

$$\begin{aligned}&\text{NO-MISDELIVERY} \equiv \\ &\qquad \mathbf{invariant}\ (\text{A, D}) \in \text{mobile-node(i).in} \Rightarrow \text{A} = \text{i}\end{aligned}$$

By convention, all free variables (i.e., A, D and i) are assumed to be universally quantified over the appropriate ranges.

In a standard UNITY program, we would prove the above by showing first that it was satisfied by the initial conditions, and then that every statement preserved it. In Mobile UNITY, we follow the same basic procedure, except that some of the statements have side-effects due to the variable sharing. These side-effects propagate changes to variables that are currently shared and also establish **engage** and **disengage** values whenever a state change causes a transition in the value of a sharing predicate. A formal proof logic for the transient sharing construct as well as other coordination constructs can be found in [5]. For the purposes of this short paper, we reason about the packet forwarding system somewhat informally, at a level that might be undertaken by a system designer while thinking about the code presented so far. This is done by enumerating the system transitions that may affect the truth of the invariant and proving that the invariant is preserved in each case. There are two actions that could potentially modify the variable *mobile-node(i).in*. The easiest to consider is the statement inside *mobile-node* that removes messages from the queue. Clearly, this statement preserves the invariant, because if no errant messages are in the queue before a removal there will be none afterwards.

The other statement that modifies the queue is the routing statement inside *network*, under the condition that the mobile node is attached and the queue is shared. Note that we cannot prove that this action preserves the above invariant: the routing statement might be directing packets meant for another node *j* (which hasn't yet updated its binding address *binding*[*j*]), to the outgoing queue currently shared with node *i*. To prevent this from happening, we need to strengthen the guard on the routing statement so that the network checks to see that the right node is currently attached. That is, we should change the statement to read

$$\text{out[dest(i)], in[i]} := \text{out[dest(i)]}\bullet\text{in[i].head, in[i].tail} \ \mathbf{if}\ \text{in[i]} \neq \phi \wedge \text{address[dest(i)]} = \text{in[i].head.A}$$

This is an example of how formal thinking helps reveal program error even when proof outlines are substituted for the actual formal treatment.

The movement statements might also affect the value of the variable *mobile-node*(*i*).*in*, because it is a transiently shared variable that could be affected by engagement and disengagement. Careful inspection of the **Interactions** section reveals that the input queue is always preserved intact for both engagement and disengagement.

In most cases, safety properties are not sufficient to fully specify the desired behavior of a program. In addition to not doing a bad thing, a program is usually expected to accomplish some good thing. Without a *liveness* property as part of the specification, the empty program would trivially satisfy the correctness criteria. In UNITY, we can express basic liveness properties using **ensures**. Again, this is a relation over state predicates:

$$p\ \mathbf{ensures}\ q$$

is written to mean that if a state satisfying p occurs anywhere in the execution, and q is not also satisfied in this state, then p will continue to be true from that state forward until a state satisfying q occurs. In addition, there is some statement that is guaranteed to establish q if selected for execution. Due to the weak fairness assumption of the UNITY statement scheduler, this statement cannot be denied forever and will eventually establish q. More complicated progress properties such as **leads-to** can also be expressed. The property

$$p\ \mathbf{leads\text{-}to}\ q$$

means that if a state satisfying p occurs, then eventually a state satisfying q will occur. Here there is no requirement that p is maintained until q is established. This property is usually proven using induction on a well-founded integer metric that is shown to eventually decrease (by inductive use of **ensures** or another **leads-to**) to zero, at which point q is established.

For example, consider the property EVENTUAL-DELIVERY, which states that a message on an output queue of some node is eventually delivered to the input queue of the node to which it was addressed.

$$\begin{array}{l}\text{EVENTUAL-DELIVERY} \equiv \\ \qquad \text{(A, D)} \in \text{mobile-node(i).out}\ \mathbf{leads\text{-}to}\ \text{(A, D)} \in \text{mobile-node(A).in}\end{array}$$

where again, free variables are quantified over the appropriate ranges. To prove this property, we might break it up into separate **leads-to** conditions and then chain these together using the standard transitivity rule for **leads-to** which states

$$\frac{p\ \mathbf{leads\text{-}to}\ q\ \wedge\ q\ \mathbf{leads\text{-}to}\ r}{p\ \mathbf{leads\text{-}to}\ r}$$

We can take p to be $(A, D) \in$ *mobile-node(i).out*, r to be $(A, D) \in$ *mobile-node(A).in*, and q to express the property $(A, D) =$ *mobile-node(i).out.head*. Then, we have broken the overall property into two parts. The first part says that any message in the output queue of a mobile node eventually reaches the head, and the second part says that a message on the head of an output queue is eventually delivered to the correct mobile node.

Note that we can prove neither of these from the text of the program as it was given. If a mobile node becomes disconnected, its output queue is not shared with the network and any packets placed there will be lost. Similarly, if the destination mobile node becomes disconnected, it will miss packets directed towards it. The protocol satisfies EVENTUAL-DELIVERY only with very strong guarantees on the movement patterns and connectivity of nodes, e.g., if mobile nodes are connected to the same foreign agent for "long enough." A UNITY mechanism called *conditional property* allows us to formalize such assumptions and to carry out the necesssary proofs. This illustrates another benefit of formal reasoning, the identification and formal characterization of the assumptions made by the protocol.

5. Conclusions

Mobile UNITY consists of a notation for specifying abstract programs and an assertional style proof logic, both aimed at the specification and verification of mobile computations. Coordination among components is specified separately from the components involved. This can be viewed as a way to manage connectivity changes in a declarative fashion. Our transient sharing abstraction is closely related to weak consistency policies in filesystems and databases, and allows explicit specification of re-integration values for disconnected variables. By adding location explicitly to the formal model (but not constraining the semantics or type of this location except on an as-needed basis) we can reason about context-dependent applications as well. Despite its stylized treatment, the packet forwarding example is representative of the verification strategies we inherited from UNITY and of the decoupled style of programming promoted by Mobile UNITY. Using Mobile UNITY, abstract programs that provide an operational characterization of some mobile computing task can be verified against formally stated assertions. We hope that, after further evaluation of the approach, some of the lessons we learned so far will make their way into the programming practice in the form of packages supporting high level coordination languages for mobile systems. The potential is here to simplify the application development effort by providing the right programming abstractions along side a semantic model supportive of formal verification.

References

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

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

[3] D. Gries, *The Science of Programming*. Springer-Verlag, 1987.

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

[5] P. J. McCann and G.-C. Roman, "Mobile UNITY: A Language and Logic for Concurrent Mobile Systems," Washington University in St. Louis, Technical Report WUCS-97-01, 1997.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

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