Modeling Mobile IP in Mobile UNITY

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 notation and proof logic for specifying and reasoning about mobile systems. 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... **Read complete abstract on page 2.**

Modeling Mobile IP in Mobile UNITY

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 notation and proof logic for specifying and reasoning about mobile systems. 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 model of packet forwarding and the mobile system in which it must operate is developed. Proofs of correctness properties, including important real-time properties, are outlined and the role of formal verification in the development of protocols like Mobile IP is discussed.

Washington

WASHINGTON·UNIVERSITY·IN·ST·LOUIS

School of Engineering & Applied Science

Modeling Mobile IP in Mobile UNITY

Peter J. McCann and Gruia-Catalin Roman

WUCS-98-03

March 30, 1998

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

Modeling Mobile IP in Mobile UNITY

Peter J. McCann and Gruia-Catalin Roman

August 17, 1998

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 notation and proof logic for specifying and reasoning about mobile systems. 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 model of packet forwarding and the mobile system in which it must operate is developed. Proofs of correctness properties, including important real-time properties, are outlined and the role of formal verification in the development of protocols like Mobile IP is discussed.

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, decoupled computing style, frequent disconnections, continuous position changes, and location-dependent nature of the behavior and communication patterns present designers with unprecedented challenges in the areas of modularity and dependability.

Mobile UNITY [11] provides a notation for mobile system components, a coordination language for expressing interactions among the components, and an associated proof logic. 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 [4], with extensions to both the notation and logic to accommodate specification of and reasoning about mobile programs. Mobile UNITY is designed to accommodate mobile applications and services that exhibit dynamic reconfiguration.

Perhaps the most basic service that can be provided in the mobile setting is simple packet routing. The Mobile IP protocol [15] is designed to deliver this service to mobile hosts that change the point at which they are attached to the Internet. In this paper we give a formal description of Mobile IP at a level of abstraction that exhibits more detail than our previous work [10]. This allows for specification and verification of several real-time properties upon which the protocol relies for correct behavior.

Because the protocol serves as a bridge from a location-dependent model of the world to a location-independent one, it must be expressed and reasoned about using location-dependent abstractions for communication. For instance, a message sent by a mobile host while it is at one attachment point will have a different effect from one sent at another, simply because the host is physically connected to a different set of real machines. The basic state transitions allowed by the model must reflect this reality, and a proof of correctness must make assumptions about the manner in which components move, because not all patterns of movement will allow for successful delivery of messages by the protocol. The Mobile UNITY model is well suited to this task.

In Section 2, we give a high-level overview of Mobile IP, including the assumptions about the environment in which the protocol is designed to operate. In Section 3 we present a Mobile UNITY model of the protocol. The construction of such a model serves two purposes. First, it illustrates the concepts and notation of Mobile UNITY on a well-known problem in mobile computing. Second, it formalizes and highlights the assumptions made by the protocol about patterns of movement and timing constraints among the components. In Section 4 we examine issues involved in formal verification of correctness of the protocol. Important issues here are what properties constitute correctness and how the proofs of such properties make assumptions about patterns of movement, message delivery, and timing constraints. Conclusions and related work are presented in Section 5.

2 Mobile IP

Mobile IP is intended to provide mobile hosts with Internet connectivity when they are away from their home networks. It allows such nodes to use the same Internet address regardless of their point of attachment to the network, so that higher layer protocols and applications can continue operation without interruption. Mobile IP presumes the existence of a fixed infrastructure, namely, the current Internet. Because most Internet routers are unaware of mobility, The philosophy behind the protocol is to reduce routing of packets directed at mobile hosts to well-understood methods of routing between fixed nodes.

The abstraction provided by the protocol is the same as for standard IP – each node is given a fixed home address to which packets can be sent. One of the goals is to keep the bulk of the Internet routing fabric completely unaware of mobility, to provide backwards compatibility with the existing network and facilitate gradual deployment. The sender of the packet is similarly unaffected by mobility; such a transmission looks like an ordinary packet. The protocol describes the collaboration that must take place between the mobility agents and the mobile node to successfully deliver a packet to the mobile host.

A high-level picture of the Mobile IP protocol is shown in Fig. 1. In what follows, we use the term "subnet" to refer to a group of nodes that are connected without traversing an IP router, and the term "network" to refer to the entire Internet routing fabric. In what is expected to be the most common mode of operation, a mobile node (MN) will contact a foreign agent (FA) upon arriving at a foreign subnet. This foreign agent will forward a registration request message to the mobile node's home agent (HA) to inform the home agent of the mobile node's new location. The registration message must contain a care-of address (COA), which in this case is the foreign agent's own address. The home agent then responds with a registration reply message confirming the registration. If packets arrive on the home network from some correspondent node (CN), they are intercepted by the home agent, which encapsulates them inside packets addressed to the foreign agent, and forwards them along. The foreign agent, upon receipt of an encapsulated packet, looks inside for the home address of any mobile nodes that are currently registered with it. If it finds one, the foreign agent decapsulates the packet and delivers it to the link-layer address of the mobile node. In
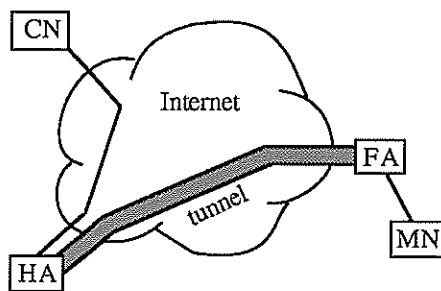

Figure 1: A high-level picture of the Mobile IP protocol.

order to intercept packets, the home agent must be located on the same subnet as the mobile node's home address. In order to contact a foreign agent, a mobile node must appear on the same subnet as the foreign agent. The mechanism used to discover a foreign agent is called an *agent advertisement* message, which is broadcast to all nodes on the local subnet. These advertisements are not propagated by IP routers and are therefore confined to the subnet on which they originate. Thus, the mobile node and foreign agent must be able to communicate at the data link layer.

Once a registration has been processed, it is only valid for a limited lifetime. Therefore, a mobile node must transmit periodic re-registrations in order to continue receiving service at its current location. These re-registrations must be sent in a timely manner to avoid interruptions in service.

The next section develops a detailed Mobile UNITY model of Mobile IP. Disconnection and reconnection of the mobile nodes, as well as transmission of registration messages and packet forwarding, are all captured by the model. Real-time properties, such as the expiration of registrations, are also captured by the addition of timers to some components. Many details are not captured, however. For example, link-layer addresses are not a part of the model, and so abstractions for communication are developed that use only network-layer IP addresses. Also, alternative modes of Mobile IP operation, such as the use of co-located care-of addresses where the mobile node performs decapsulation instead of the foreign agent, are not modeled for the sake of simplicity.

3 Model

This section presents a Mobile UNITY system that models Mobile IP at a more detailed level than our previous work [10]. It consists of a set of programs of type *mobile-node*, parameterized by subnet and node number; a set of *home-agent* and *foreign-agent* programs, parameterized by subnet number; and a *network* program that models standard IP routing. These components are illustrated in Fig. 2. There is assumed to be one location λ_s for each subnet s, corresponding to the gray circles in Fig. 2. We assume that if a mobile node is at a location λ_s, it may communicate with the mobility agents at the same location without resorting to IP routing. In reality, this might represent a region or set of locations which are in range of a transmitter or where the mobile node may be "plugged in" to a wired network. In the model given below, the components will communicate using a novel construct called *transient sharing*.
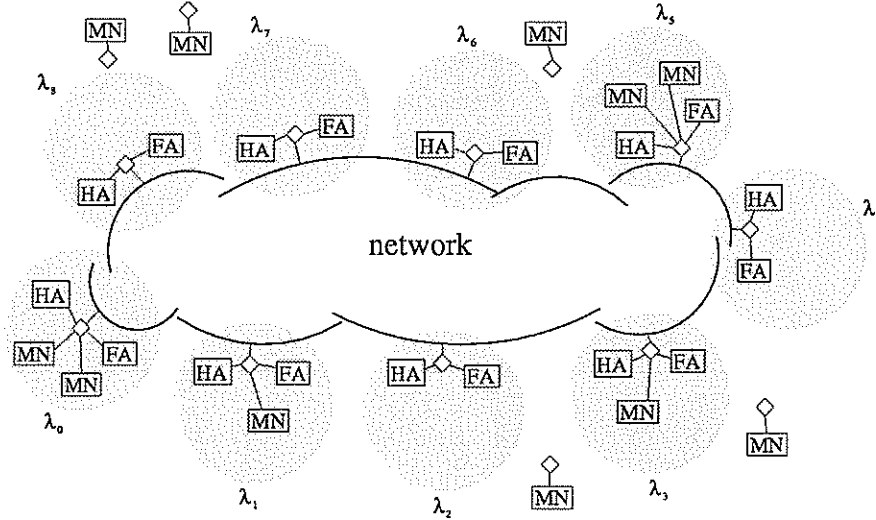

Figure 2: Mobile IP system components.

That is, when a mobile node is at a given location λ_s, it may communicate with the other components at that location by writing to a variable which is shared only for the duration of co-location. This is illustrated in Fig. 2 with the overlapping diamonds. Each such variable is assumed to hold one IP packet and is given the name *ether* in each component, although this is not intended to suggest any specific communication medium.

IP packets in the Mobile IP protocol will be specified as type *message*, which is defined to be a 4-tuple of (*source, dest, type, data*). For a given message *msg*, these fields will be referenced as *msg.source, msg.dest, msg.type*, and *msg.data*. The first two fields are the IP source and destination addresses. Each address is assumed to be of the form (*subnet, node*). Again, we will reference the fields of an address with the notation *address.subnet* or *address.node*. Here we assume that all addresses are of the same class rather than attempt to model addresses of several classes or classless addresses as in [7]. This makes for a simpler specification of the fixed IP routing network, which is not our focus here. We assume that messages with destination address set to (-1, -1) are subnet-directed broadcasts. These will be used for agent advertisement messages.

Each type of component (*network, mobile-node, home-agent*, and *foreign-agent*) will be modeled as one Mobile UNITY program. Each program comprises a **declare, always, initially**, and **assign** section. The **declare** section contains a set of variables that will be used by the program. Each is given a name and a type. The **always** section contains constant definitions that may be used for convenience in the remainder of the program or in proofs. The **initially** section contains a set of state predicates which must be true of the program before execution begins. Finally, the **assign** section contains a set of assignment statements. In each section, the symbol '[]' is used to separate the individual elements (declarations, definitions, predicates, or statements).

Each assignment statement is of the form $\vec{x} := \vec{e}$ **if** p, where $\vec{x}$ is a list of program variables and $\vec{e}$ is a list of expressions, and p is a state predicate called the *guard*. When a statement is selected, if the guard is satisfied, the right-hand side expressions are evaluated in the current state and the resulting values are stored in the left-hand side variables. The standard UNITY execution model is a nondeterministic, fair interleaved selection of all statements from the **assign** section. The Mobile UNITY execution model is slightly different and three new kinds of statements

3

are added: the *reactive* statement, the *transaction* statement, and the *inhibition* statement. Each will be described at the point it is used in the programs below.

The program *network* is shown in Fig. 3. Conceptually, this program models the Internet routing fabric as well as the "edge routers" or "gateways" that appear on each subnet. It illustrates the strategy used to model communication between components of the system with its handling of the local variable *ether*. This is an array of message values, where each *ether*[i] denotes the message currently appearing at the gateway interface of subnet i. The **Interactions** section, defined later, will declare the sharing relationships between the *ether* variables of the mobile nodes and the elements of the array *network.ether*. In all that follows, we assume that there are *Nsubnets* subnets which the network must service.

> **program** *network* **at** λ
>   **declare**
>   *in, out* : array[*Nsubnets*] of queue of message
>   [] *ether* : array[*Nsubnets*] of message
>   **assign**
>   ⟨[] $i : 0 \leq i < Nsubnets$ ::
>   {Transfer messages from link to input queue.}
>   [] *in*[i], *ether*[i] := *in*[i] • *ether*[i], ⊥
>   **reacts-to** *ether*[i] ≠ ⊥
>   ∧ *ether*[i].*dest.subnet* ≥ 0 ∧ *ether*[i].*dest.subnet* ≠ i
>
>   {Transmit outgoing packets in each subnet.}
>   []⟨ *ether*[i], *out*[i] := *head*(*out*[i]), *tail*(*out*[i]) **if** *out*[i] ≠ ϵ ∧ *ether*[i] = ⊥ ;
>   *ether*[i] := ⊥ ⟩
>
>   {Sometimes drop messages.}
>   [] *out*[i] := *tail*(*out*[i]) **if** *out*[i] ≠ ϵ
>   ⟩
>   ⟨[] $i, j : 0 \leq i < Nsubnets ∧ 0 \leq j < Nsubnets$ ::
>   {Route packets internally.}
>   [] *out*[j], *in*[i] := *out*[j] • *head*(*in*[i]), *tail*(*in*[i]) **if** *in*[i] ≠ ϵ ∧ *head*(*in*[i]).*subnet* = j
>   ⟩
>   **end**

Figure 3: The program *network*, which performs simple routing of messages based on destinations.

The program *network* consists of two sets of quantified assignment statements[1]. The first set is quantified in one dimension over the subnets. The first statement of this set receives messages from *ether*[i] and places them on an input queue for the subnet i. Note that this statement is *reactive*, meaning that it executes immediately whenever the predicate following **reacts-to** is *true*. This construct is unique to Mobile UNITY; its precise definition and semantics may be found in [11]. Operationally, this statement can be thought of as an interrupt triggered by the presence of a valid packet on *ether*[i]. It executes exactly once because in doing so, it falsifies the trigger condition following **reacts-to**. In general, there may be many reactive statements, and each is allowed to execute until quiescence after each non-reactive statement. This particular reaction appends the packet on *ether*[i] to the end of the queue *in*[i] using the notation *in*[i] • *ether*[i]. With the use of the guard *ether*[i].*dest.subnet* ≠ i, the statement accepts any message whose destination subnet is different from the one on which it appears, and the additional guard *ether*[i].*dest.subnet* ≥ 0 ensures that only unicast messages are consumed. Because it is a reactive statement, any such message appearing on *ether*[i] will immediately be appended to the input queue, and deleted from *ether*[i]. This abstracts away from several details of this first hop made by a packet, including the decision to send a packet to the gateway, the discovery of the address of the gateway by the sender of a message, and the resolution of such an address to a hardware address. However, we deliberately ignore these details in order to focus attention on issues related to mobility.

The second statement transmits an already-routed message to the appropriate *ether*[i]. This statement is a

---

[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, 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 ∀ or the null element if *op* is [].

transaction consisting of two parts. Transactions were also introduced with Mobile UNITY, and their precise semantics may be found in [11]. Operationally, each part of the transaction executes in sequence, and reactive statements are given a chance to execute after each part. The first part of this particular transaction writes a value to *ether*[*i*], and the second sets the variable *ether*[*i*] to ⊥, a special value representing the condition that no message is currently being transmitted on the medium connected to the gateway. This models the fact that messages are present on the medium for only a short period of time, after which it returns to a quiescent state. The reactive statements that consume messages are given a chance to run between the two parts of the transaction. However, even if a packet appears on the medium that is not consumed by anyone, it will be discarded by the second part of the transaction.

The next statement models the dropping of packets in transit, which may happen for any number of reasons such as congestion or network outage. IP offers no guarantee that any particular message will eventually be delivered. The final statement models the routing performed internal to the network. Note that there is no upper bound on the time it takes to eventually deliver a message; message delivery is modeled by nondeterministic, fair interleaving of the atomic actions present in the program. These assumptions will present several interesting challenges to the verification of properties of the system as a whole.

The program *mobile-node* is shown below, parameterized by subnet and node number. It is at this point that we need to be concerned with message contents, not just the source and destination addresses. The *type* field of a message can be any of *Regular, Advertisement, Request, Reply*, or *Encapsulated*. The final field, *data*, will have a different interpretation for each message type. For a message of type *Regular*, it will be an opaque value. For a message of type *Advertisement*, it will be a pair of boolean values (*homeflag, foreignflag*) indicating whether the advertisement is for a home agent or a foreign agent. If it is a home agent advertisement, the *homeflag* will be set to *true* (we will sometimes use the keyword *home* in place of *true*), and if it is a foreign agent advertisement, the *foreignflag* will be set to *true* (we will sometimes use the keyword *foreign* in place of *true*). For a message of type *Request, data* will be a triple (*home, foreign, ID*), where *home* represents the home address (*s, n*) of the mobile node making the request, *foreign* represents the care-of address under which the mobile node is requesting to be registered, and *ID* represents a unique identifier assigned to the request that will be used for matching replies to requests. Each address is again a (*subnet, node*) pair. For a type *Reply* the *data* field will be a response of the form (*home, statusflag, ID*), where *home* is the home address of the mobile node that initiated the request, *statusflag* is *Ok* or *Denied*, and *ID* is the identifier from an earlier request. For a message of type *Encapsulated*, the *data* field will be another message.

On any subnet *s* we assume that the home agent has address (*s, HA*) and the foreign agent has address (*s, FA*). Thus, the mobility agents for each subnet appear at well-known addresses. In an implementation of the protocol, the true home agent address would be known to the mobile host and transmitted with its registration request, or discovered when needed using multicast techniques. Also, the foreign agent would make its address known when it broadcasts advertisements. The well-known addresses are used to simplify the presentation and keep the message format simple.

The first statement of *mobile-node* models movement of the node from one location to another. We assume the existence of an external function *Move* which returns the next location of the node. The other components with which the mobile node may communicate will be dictated by its current location, as specified in the **Interactions** section given later.

The modeling of unicast message reception with the first reactive statement is very similar to the way this was accomplished in program *network*, except that the mobile-node must match the destination address to its own address. There is some additional complexity, however, in the way a mobile node receives broadcast messages. The mobile node is the only program that must do so, since it must receive agent advertisement messages as specified by Mobile IP. We ignore all other forms of broadcast messages for simplicity, and do not model the forwarding of broadcast packets to mobile nodes, which is allowed by the Mobile IP specification. The agent advertisement messages are indicated by a destination address of (-1, -1), and every mobile node present at a location λ_s must receive such an advertisement. Therefore, the message cannot be consumed as in the case of a unicast packet, where such consumption models the fact that the unicast IP address of a host was resolved through the use of mechanisms such as ARP [16] to a unicast hardware address which is ignored by all recipients except the intended one. Because a broadcast message must be added to the input queue only once, the boolean *gotflag* is needed to ensure that the statement modeling such reception is idempotent without actually overwriting the value of *ether*, which must be seen and reacted to by other mobile nodes that may be listening on the same subnet. When the message is removed from *ether* by the component which sent the message in the first place, *gotflag* is reset by the third reactive statement.

Once messages have been added to the input queue, they are processed one at a time in the order received. The predicates in the **always** section are very important to this task. For each predicate, there is a corresponding statement

designed to handle messages that match the predicate and to take appropriate action. For instance, the predicate *IsForeignAd* is true whenever the first message in the input queue is a foreign advertisement and the mobile node is not currently registered with the foreign agent that sent the message. Because there is only one foreign agent on each subnet, receipt of such a message indicates that the mobile node has moved to a new subnet and should generate a new registration request. The statement that processes the advertisement generates such a request, placing it in *transbuffer* and recording the time at which it did so. The next statement processes messages satisfying *IsHomeAd*, and generates a similar de-registration request, which is also placed in *transbuffer*. The *transmit* statement then transmits this value repeatedly, until an *OkReply* is processed, which sets the boolean flag *confirmed* to *true*. This statement increments *transcount* with each transmission, and the guard on the transmit statement ensures that a certain minimum time has elapsed since the last transmission, modeling the exponential delay increments called for by the specification.

The statement labeled *reregister* is enabled whenever the local clock indicates that almost *Lifetime* seconds (specifically, *Lifetime* − *D* seconds) have elapsed since the registration message was first generated. This statement generates a new request ID and sets the *confirmed* flag to *false*, which re-enables transmission of the request message. The constant *D* reflects assumptions about variabilities in the propagation delay suffered by different request messages as they are routed through the program *network*. These assumptions are not explicitly represented in the text of the system for the sake of simplicity. However, a validation proof could explicitly assume such properties, and indeed this may be necessary to prove certain correctness conditions. The next section will return to this issue in more detail.

The next two statements model the sending and receiving of messages from higher layers in the protocol stack. The reception of such a message is simply modeled by discarding it, and the transmission of messages assumes the existence of the functions *NewDest* and *NewData*, which return the next destination and contents to be transmitted.

The last three statements are intimately involved in the specification of timing constraints discussed earlier. The first, *timer*, simply increments the locally declared integer *clock* by one. In this system, we assume that *clock* represents a free-running count of elapsed seconds. In the **Interactions** section, this clock will be linked to the clocks of other components to express the fact that all clocks run at about the same rate. When considered in isolation, however, all timing constraints of the mobile-node can be expressed in terms of this local clock. For example, the guards on some statements prohibit their execution until the clock has reached a certain minimum value. This expresses a lower bound on the time at which the action may execute. The **inhibit** clauses at the end of the program, in contrast, express upper bounds on the time at which actions may execute. By inhibiting the *timer* statement under those conditions in which these constraints would be violated, the inhibitions force the execution of those statements that falsify the inhibiting condition. Of course, a separate proof that the timer eventually makes progress is required for complete correctness. This corresponds to the notion of non-Zenoness in [1].

Once the *mobile-node* program is understood, the remaining programs for the home-agent and foreign-agent follow in a straightforward manner from the Mobile IP specification [15]. The code for the program *home-agent* is shown below.

Note that the home agent reacts to any message whose destination is in its *InterestSet*, because it must encapsulate packets for the mobile nodes that are away from home. The *InterestSet* models mechanisms such as proxy ARP [17] and gratuitous ARP [21] for mapping these packets to the hardware address of the home agent.

The home agent must also process request messages from mobile nodes that are away from home. The next statement accomplishes this and, like most of the remaining statements, in addition to consuming a message from the input queue it also transmits a message on the *ether*. As with the programs *network* and *mobile-node*, each transmission is the first part of a two-part transaction, the second phase of which clears the value written. In this case the home agent consumes a request message and transmits a response, setting the forwarding address in the array *home_forward* and recording the time at which the request was processed in *home_rtime*. In each array there is one element for each mobile node served by the home agent, the number of which we assume is *Nnodes*. In addition to setting up the forwarding address and sending a reply, the mobile node's home address is added to *InterestSet* so that packets intended for the mobile node can be intercepted by the home agent.

The next statement is responsible for de-registering the mobile nodes that arrive home. Upon receiving a de-registration request, the home agent deletes the appropriate *home_forward* address, removes the mobile node's address from its *InterestSet*, and sends a reply confirming the de-registration. The statement labeled *advertise* is executed periodically, sending out a broadcast packet on *ether*. This advertisement is received by all mobile nodes that are currently at home, letting them know that they should send a de-registration request if they have not already done so.

The statement labeled *timeout-fwd* is responsible for deleting a mobility binding when *Lifetime* seconds have elapsed since the registration was received. The guard on this statement ensures that it will not execute before the

6

lifetime has expired. Note that unlike the *mobile-node* program, there are no **inhibit** statements representing deadlines by which actions must occur. Thus, the *timeout-fwd* action may not execute until long after the lifetime has expired, but fairness constraints require that it will execute eventually. Like the *mobile-node* program, the *home-agent* has a timer statement responsible for incrementing the local clock which will be linked to other clocks in the system by the **Interactions** section.

The code for the foreign-agent is shown below. Of all the programs, its message reception takes on the simplest form, because it may only receive unicast packets directed at its own address.

The foreign agent receives registration requests from mobile nodes, and forwards them on to the appropriate home agent. Each foreign agent contains state large enough to handle registrations from up to *Nvisitors* mobile nodes. The statement *process-request* removes a request message from the input queue and forwards a request to the appropriate home agent. Also, it allocates a new location i in the *visitor_assign* array to track which mobile node sent the request (old locations are re-used in the case of a re-registration). The element *visitor_state[i]* is set to *Pending*, and *visitor_rtime[i]* records the local time at which the request was processed. If all locations in *visitor_assign* are full of registrations from other mobile nodes, the agent immediately transmits a *Denied* response. Otherwise, the foreign agent waits for a reply from the home agent, which it then forwards to the mobile node, deleting its record of the registration if the home agent denied the request, and resetting the *visitor_rtime[i]* value if the request was successful. This value is later used to allow the registration to expire, which is carried out by the statement labeled *timeout-visit(i, t, n)*. As was the case with the home agent, the foreign agent must periodically advertise itself on the local network and maintain a local clock.

The complete system modeling Mobile IP is shown in Fig. 11. It consists of the program definitions, omitted here to avoid repetition, the **Components** section which declares instances of the programs along with their initial positions, and the **Interactions** section which defines how the components share state and expresses the constraints that the local clocks of each component must run at approximately the same rate.

The various programs communicate by sharing their *ether* variables. The interactions given in Fig. 11 specify that the home agents and foreign agents are connected to the network at all times, while the mobile nodes are only connected to the network when they are co-located with some subnet and are then only connected to that subnet's ether. Because this sharing is transitive, a mobile node is also connected to that subnet's foreign agent and home agent. Transient sharing is defined in terms of reactive statements and the introduction of auxiliary system variables:

$A.x \approx B.y$ **when** $p \equiv$
  $B.y,\ B.y_{A.x},\ A.x_{B.y} := A.x,\ A.x,\ A.x$
    reacts-to $A.x \neq A.x_{B.y} \wedge p$
  $A.x_{B.y} := A.x$
    reacts-to $\neg p$
  $A.x,\ A.x_{B.y},\ B.y_{A.x} := B.y,\ B.y,\ B.y$
    reacts-to $B.y \neq B.y_{A.x} \wedge p$
  $B.y_{A.x} := B.y$
    reacts-to $\neg p$

Here auxiliary variables $A.x_{B.y}$ and $B.y_{A.x}$ are introduced which track the last state of each variable, and reactive statements immediately propagate changes if any difference between the previous and next states occurs. If the components are connected (p is *true*), then changes are propagated to the remote variable as well as the local history variable. If the components are disconnected (p is *false*), then changes are propagated only to the local history variable.

Clock synchronization is expressed with the **synch** construct, which is defined as:

$A.clock$ **synch** $B.clock$
  when r
  within $D \equiv$
    $A.clock_B,\ B.clock_A := A.clock,\ B.clock$ **reacts-to** r
    inhibit A.timer **when** $A.clock - A.clock_B > D \cdot (B.clock - B.clock_A)$
    inhibit B.timer **when** $B.clock - B.clock_A > D \cdot (A.clock - A.clock_B)$

The predicate in the **when** portion defines a condition that demands the clocks proceed with bounded drift from that point forward. This does not constrain the actual values of the clocks but only the rate at which they advance.

The **within** value specifies the drift ratio: in this case, neither clock of any pair will advance more than one percent more than the other. The **when** predicates given above constrain the clocks of two components whenever one component of a pair sends a message of any type to the other (dashes are used to represent don't-care conditions). This may seem to be action-at-a-distance, because it affects both parties as soon as the packet is transmitted, before any information could propagate to the receiver. However, it is simply an expression of the constraint that the two clocks run at about the same rate, a perfectly reasonable and implementable assumption without communication, assuming of course that the two components are not moving so quickly that relativistic effects play a role. Even though much stronger guarantees could have been expressed, such as bounded drift from any point in system execution, the constraints represented here are sufficient to express the guarantees relied on by the protocol. For instance, the mobile node only requires that its local clock and that of its home agent run at approximately the same rate after it has sent a request. Note that no timing constraints are expressed for the program *network* or for the time spent by a message in the input queues of components, although they could be added using techniques already introduced. Assumptions about these processing times will be important to the correct functioning of the protocol, and the next section examines how these assumptions can be expressed during the verification process.

The program as given models many aspects of the Mobile IP protocol, but abstracts away from many others. Authentication, link-layer identifiers, and other modes of operation (e.g., co-located care-of addresses from [15]) are all aspects that are not directly modeled by the system. However, the modeling of location-dependent interaction among components clearly captures many interesting aspects of the various dependencies among components and assumptions made by the protocol designers.

4 Verification Strategy

It would be naive to claim that a proof of a particular set of formal properties of the above system would demonstrate the correctness of the Mobile IP specification or any particular implementation of it. This is true for several reasons. First, there is no precise and accepted definition of correctness for the protocol, because it makes no guarantees that a message will ever be delivered to a mobile node. In fact, depending on the pattern of movement undertaken by a mobile node, it may never receive forwarded packets. Second, there are simply too many details that have been abstracted out of the above system. Any errors or oversights in the draft specification of Mobile IP or implementations of it are likely to be subtle feature interactions between this protocol and any of the myriad of other Internet protocols, all of which are outside the scope of this paper.

However, attempts at proofs can still serve useful purposes. First, they can help to clarify informal notions of correctness that make up expectations about the way the protocol should behave under most conditions. Second, they can help to identify under what conditions and assumptions the protocol is expected to behave properly. This paper outlines one possible approach to this task. In addition to clarifying expectations and assumptions specific to Mobile IP, the verification exercise will also illustrate the proof techniques available in Mobile UNITY.

We begin by presenting formal correctness criteria for Mobile IP that are justified informally. Then, the assumptions underlying the protocol are formalized with conditional properties, a standard UNITY mechanism for handling such assumptions in proofs. Proof outlines illustrate the basic steps required to carry out a proof in Mobile UNITY.

4.1 Correctness

Like non-mobile Internet routing protocols, Mobile IP offers no strong guarantee of message delivery. Because it relies on an existing, lossy infrastructure, any message may be lost in transit. Also, given a mobile node that is moving from subnet to subnet sufficiently quickly, registration messages will be out of date by the time they reach the home agent, and no forwarded messages will ever reach the mobile node. Any correctness properties must take these issues into account.

Informally, we might require that if a mobile node stays at one location for "long enough," it will eventually have an up-to-date registration at its home agent and packets will eventually be forwarded to it. Formally, this might be represented with the property *REGISTER*, defined as

$REGISTER(s,\ t,\ n) \triangleq$
$\quad mobile\text{-}node(s,\ n).\lambda = \lambda_t \mapsto$
$\qquad home\text{-}agent(s).home\text{-}forward[n] = (t,\ FA) \vee mobile\text{-}node(s,\ n).\lambda \neq \lambda_t$

REGISTER(s, t, n) asserts that if a mobile node is at a given subnet, then eventually its registration is up to

date or it has moved to a different subnet. This uses the UNITY relation $\mapsto$ (read *leads-to*), described in [4], which expresses progress by requiring that if, at any point during execution, the predicate on the left-hand side is satisfied, then there is some later state where the predicate on the right-hand side is satisfied. In this definition and in what follows, the free variables s, t, and n are implicitly quantified over the appropriate ranges, with the restriction that $s \neq t$.

In addition to requiring that a registration eventually takes place, we might also desire that such a registration persist for as long as the mobile node is continuously present at the foreign network, as expressed by *PERSIST*:

$$PERSIST(s, t, n) \triangleq$$
$$\quad \textit{mobile-node}(s, n).\lambda = \lambda_t$$
$$\wedge \ \textit{home-agent}(s).\textit{home\_forward}[n] = (t, FA) \ \mathbf{co}$$
$$\qquad \textit{home-agent}(s).\textit{home\_forward}[n] = (t, FA) \vee \textit{mobile-node}(s, n).\lambda \neq \lambda_t$$

PERSIST(s, t, n) asserts that as long as the mobile node remains at its foreign location, the registration is kept up-to-date. This is specified with the UNITY relation **co**, described in [14], which constrains any state transition that takes place in a state satisfying the left-hand side to produce a state satisfying the right-hand side. *PERSIST* should be maintained by the periodic re-registration performed by the mobile node when it detects that its current registration is about to expire.

The constraint that packets are forwarded could be expressed with the property *DELIVERY*, defined as

$$DELIVERY(s, t, n) \triangleq$$
$$\quad \textit{mobile-node}(s, n).\lambda = \lambda_t$$
$$\wedge \ \textit{home-agent}(s).\textit{home\_forward}[n] = (t, FA)$$
$$\wedge \ \textit{head}(\textit{network.out}[s]) = (X, (s, n), Y, Z) \mapsto$$
$$\qquad \textit{head}(\textit{mobile-node}(s, n).in) = (X, (s, n), Y, Z)$$
$$\quad \vee \ \textit{mobile-node}(s, n).\lambda \neq \lambda_t$$

DELIVERY(s, t, n) asserts that if a home agent's registration is up to date, any packet that arrives for the mobile node at its home address will eventually arrive at the mobile node, unless the mobile node moves away while the packet is in transit. Here the dummy variables X, Y, and Z are assumed to be universally quantified, and so this property implies delivery of a message with any source, type, and contents to the mobile node named in its destination. Again, the notation $head(q)$ is used to denote the first element of queue q.

Although these properties seem to formalize very well our infomal notions of correctness for Mobile IP, none of these properties are true of the system *mobile-ip*, because they ignore the possibility of message loss in the network. For example, executions of the system are possible in which every registration request sent to the home agent is dropped by the network, and so *REGISTER* is not provable from the text of the system. Also, even if the network does not drop a packet, it may delay it for arbitrary periods of time and cause registrations to expire prematurely, falsifying *PERSIST*. However, the properties can be proven under some assumptions about the performance of the network that we choose to express with conditional properties, a standard technique from UNITY for reasoning about conditions that are not modeled directly by the program but that should be true of the environment in which the program runs.

For instance, it might be appropriate to assume that the network may not drop every packet, or that one of a set of retransmitted registrations eventually gets through to the home agent, within some reasonable period of time. This might suffice to prove *REGISTER* and *PERSIST*. To prove *DELIVERY*, however, we are interested in the correct delivery of every packet, and so we must assume that the network does not drop any packets. The property *NONDROP* states that a packet appears at the correct destination within a bounded period of time, according to the local clock of the destination component.

$$NONDROP(s, t, n) \triangleq$$
$$\quad \textit{last}(\textit{network.in}[s]) = (X, (t, n), Y, Z) \wedge DEST(t, n).\textit{clock} = k \mapsto$$
$$\qquad \textit{head}(\textit{network.out}[t]) = (X, (t, n), Y, Z)$$
$$\quad \wedge \ DEST(t, n).\textit{clock} < k + \textit{bound}$$

The notation $last(\textit{network.in}[s])$ is used to denote the message at the end of input queue s. Here $DEST(t, n)$ stands for *home-agent*(t) if n is *HA*, *foreign-agent*(t) if n is *FA*, and *mobile-node*(t, n) otherwise. Thus, the property states that a packet given to the network is delivered to its destination within *bound* seconds. Proof of *REGISTER* does not require a finite bound (eventual reliable delivery suffices), but such a bound is necessary to prove *PERSIST*.

This is because a registration at home may expire before a new one is sent, if the network is allowed to delay packets arbitrarily.

We must also assume that once a registration request packet is delivered to the head of the correct network output queue, it must be processed by the home agent within a bounded amount of time. This embodies assumptions made about the speed with which the home agent can process messages. For instance, *PERSIST* may be violated if a registration request message sits unprocessed for an unbounded period of time in the home agent's input queue. Such a property could be written as

$PROCESS(s, t, n) \triangleq$
    $head(network.out[s]) = ((t, FA), (s, HA), Request, ((s, n), (t, FA)))$
$\wedge$ *home-agent(s).clock* $= k \mapsto$
        home-agent(s).home_rtime[n] $> k$
    $\wedge$ *home-agent(s).home_forward[n]* $= (t, FA)$
    $\wedge$ *home-agent(s).clock* $< k +$ *bound*

$PROCESS(s, t, n)$ asserts that once a registration request message has arrived at a subnet, it is processed by the home agent for that subnet within a bounded period of time. For simplicity, we assume this bound is the same as the bound on transmission time in the network given by *NONDROP*. For complete correctness, similar bounds on the processing time at foreign agents and the number of mobile nodes present at each foreign agent should also be given. This is because an arbitrary delay at the foreign agent, or overcrowding of the foreign agent, can prevent registration from taking place in a timely fashion. We omit them for the sake of clarity; for the purposes of this presentation, assume they are contained within the property *PROCESS*.

With these definitions, we can express the formal correctness property for the protocol as a UNITY-style conditional property in hypothesis-conclusion form:

$$\frac{NONDROP(t, s, HA) \wedge NONDROP(s, t, FA) \wedge PROCESS(s, t, n)}{REGISTER(s, t, n) \wedge PERSIST(s, t, n) \wedge DELIVERY(s, t, n)}$$

This formula, which we take to be the correctness specification of system *mobile-ip* for the sake of this paper, can be read as, "given the network does not drop packets sent between a home agent on subnet s and a foreign agent on subnet t, and given that the home agent processes messages in a timely manner, the properties $REGISTER(s, t, n)$, $PERSIST(s, t, n)$, and $DELIVERY(s, t, n)$ can be proven from the text of the system."

4.2 Proof Outline

Consider the property $REGISTER(s, t, n)$. It states that a mobile home on a foreign subnet will eventually have an up-to-date registration at its home agent, unless it moves away first. This can be proven in a number of steps, using the transitive property of leads-to at each step. The first step is to prove that the mobile node eventually gets an advertisement message from the foreign agent. Operationally, this is accomplished in one step by the transaction in the program *foreign-agent(t)* that broadcasts advertisements. Formally, the property can be stated using the **ensures** operator, where s, t, and n are quantified over the appropriate domains, and $s \neq t$:

    mobile-node(s, n).$\lambda = \lambda_t$
      ensures
    mobile-node(s, n).$\lambda \neq \lambda_t$
    $\vee$ *(IsForeignAd(last(mobile-node(s, n).in))*
        $\wedge$ *last(mobile-node(s, n).in).source* $= (t, FA))$

Here *IsForeignAd()* is as defined inside *mobile-node(s, n)*, but may take any message in the system as its argument. The expression p **ensures** q states that if the program is in a state satisfying p, it remains in that state unless q is established, and, in addition, it does not remain forever in a state satisfying p but not q. According to [13], **ensures** can be defined with the more primitive operators **co** and **transient**:

$$p \text{ \textbf{ensures} } q \quad \triangleq \quad (p \wedge \neg q \text{ \textbf{co} } p \vee q)$$
$$\wedge \quad \textbf{transient} (p \wedge \neg q)$$

where p **co** q asserts that whenever the system is in a state satisfying p, its next state must satisfy q. Formally, this can be expressed with a Hoare triple [8] quantified over all state transitions of the system:

$$p \text{ co } q \stackrel{\Delta}{=} \langle \forall s \in \mathcal{N} :: \{p\}s^*\{q\}\rangle \wedge p \Rightarrow q$$

This is similar to the definition in [14], but in our case, the state transitions are defined as transformations s^* of each non-reactive statement s. Here the set $\mathcal{N}$ denotes all non-reactive statements of the system from all instantiated components, including multi-part transactions. The transformation into s^* will account for the reactive statements that are allowed to execute as a side-effect of s, and the possible inhibition of statement s.

We take as our basic notion of progress the **transient** operator defined in [13], again taking our quantification over all transformed non-reactive statements in the system:

$$\textbf{transient } p \stackrel{\Delta}{=} \langle \exists s \in \mathcal{N} :: \{p\}s^*\{\neg p\}\rangle.$$

Informally, **transient** p asserts that whenever the system is in a state satisfying p, it will eventually execute some statement that falsifies p.

We will present the proof of this property in its entirety in order to illustrate the basic Mobile UNITY proof logic, but in later proofs we will omit the low-level details for the sake of brevity. Proof of the above **ensures** must proceed in two parts. First, the safety part may be re-written as

$mobile\text{-}node(s,\ n).\lambda = \lambda_t$
$\wedge\ \neg(IsForeignAd(last(mobile\text{-}node(s,\ n).in))$
$\quad\ \wedge\ last(mobile\text{-}node(s,\ n).in).source = (t,\ FA))$
co
$mobile\text{-}node(s,\ n).\lambda = \lambda_t$
$\vee\ mobile\text{-}node(s,\ n).\lambda \neq \lambda_t$
$\vee\ (IsForeignAd(last(mobile\text{-}node(s,\ n).in))$
$\quad\ \wedge\ last(mobile\text{-}node(s,\ n).in).source = (t,\ FA))$

This asserts that if a mobile node is away from home and has not received an agent advertisement, in the very next state of the system, the mobile node must either (1) remain at the same location, or (2) move to a new location, or (3) receive an agent advertisement.

Because

$$mobile\text{-}node(s,\ n).\lambda = \lambda_t \vee mobile\text{-}node(s,\ n).\lambda \neq \lambda_t$$

is identically *true*, it is easy to see that no statement violates this property. However, we are still obligated to show that the reactive program $\mathcal{R}$ terminates in every case. The reactive program consists of all of the reactive statements from the text of the system, as well as those that result from the transient sharing relationships that are defined in terms of reactive statements. Clearly, no reactive statement modifies the position of the mobile node. Termination of the reactive program can be determined because each statement disables itself, and although some are re-enabled and may fire twice, none may fire three times. To see this it may be helpful to look back at the definition of transiently shared variables. The reactive statements that propagate updates to shared variables are triggered only when the variable holds a new value, a condition that may occur only at the start of the reactive program when a component writes a value to the shared ether, or during it when the destination component consumes the message and sets the ether to $\perp$.

The proof of the second part of the **ensures** is the demonstration of a particular statement that satisfies the right-hand side. This statement is, of course, the one in *foreign-agent(t)* that broadcasts an agent advertisement to the shared ether. Recall the central proof axioms from [11]:

$$\frac{p \wedge i(s) \Rightarrow q,\ \{p \wedge \neg i(s)\}s^{\mathcal{R}}\{q\}}{\{p\}s^*\{q\}} \tag{1}$$

$$\frac{\{r\}s\{H\},\ H \mapsto (FP(\mathcal{R}) \wedge q) \text{ in } \mathcal{R}}{\{r\}s^{\mathcal{R}}\{q\}} \tag{2}$$

These require the demonstration of a property H that is established by the non-reactive statement—in this case the broadcast of the agent advertisement—and that leads to termination of the reactive program in a state satisfying the postcondition. The advertisement statement is never inhibited. Using the following definition

11

$$Advert(s,\ n,\ t,\ msg) \triangleq mobile\text{-}node(s,\ n).IsForeignAd(msg)$$
$$\wedge\ msg.source = (t,\ FA)$$

we can take H to be

$H \equiv Advert(s,\ n,\ t,\ foreign\text{-}agent(t).ether)$

$\wedge\ foreign\text{-}agent(t).ether \neq foreign\text{-}agent(t).ether\ _{network.ether[t]}$

$\wedge\ network.ether[t] = network.ether[t]\ _{mobile\text{-}node(s,\ n).ether} = \perp$

$\wedge\ mobile\text{-}node(s,\ n).ether = \perp$

$\wedge\ mobile\text{-}node(s,\ n).gotflag = false$

Recall that transient sharing introduces history variables recording the last value transmitted in either direction. For example, the variable $foreign\text{-}agent(t).ether\ _{network.ether[t]}$ holds the last value transmitted from the foreign agent to the network. The proof of $\{r\}s\{H\}$ actually requires other invariants, omitted here, that state that all *ether* values are equal to $\perp$ at the end of every transaction.

The proof of the $\mapsto$ part can be proven with the aid of three **ensures** statements, corresponding to (1) the transfer of the message from the foreign agent to the network, (2) the transfer of the message from the network to the mobile node, and (3) the placement of the message on the mobile node's input queue. These are simply

$Advert(s,\ n,\ t,\ foreign\text{-}agent(t).ether)$

$\wedge\ foreign\text{-}agent(t).ether \neq foreign\text{-}agent(t).ether\ _{network.ether[t]}$

$\wedge\ network.ether[t] = network.ether[t]\ _{mobile\text{-}node(s,\ n).ether} = \perp$

$\wedge\ mobile\text{-}node(s,\ n).ether = \perp$

$\wedge\ mobile\text{-}node(s,\ n).gotflag = false$

ensures

    $Advert(s,\ n,\ t,\ network.ether[t])$

    $\wedge\ network.ether[t]\ _{mobile\text{-}node(s,\ n).ether} = \perp$

    $\wedge\ mobile\text{-}node(s,\ n).ether = \perp$

    $\wedge\ mobile\text{-}node(s,\ n).gotflag = false$

which asserts that a message broadcast by the foreign agent is transferred to the local subnet's *ether*; and

$Advert(s,\ n,\ t,\ network.ether[t])$

$\wedge\ network.ether[t]\ _{mobile\text{-}node(s,\ n).ether} = \perp$

$\wedge\ mobile\text{-}node(s,\ n).ether = \perp$

$\wedge\ mobile\text{-}node(s,\ n).gotflag = false$

ensures

    $Advert(s,\ n,\ t,\ mobile\text{-}node(s,\ n).ether))$

    $\wedge\ mobile\text{-}node(s,\ n).gotflag = false$

which asserts that such a message is transferred to the mobile node's ether; and finally,

$Advert(s,\ n,\ t,\ mobile\text{-}node(s,\ n).ether))$

$\wedge\ mobile\text{-}node(s,\ n).gotflag = false$

ensures

    $Advert(s,\ n,\ t,\ last(mobile\text{-}node(s,\ n).in))$

which asserts that a message on the mobile node's *ether* is appended to its input queue. The proof of the safety part of each **ensures** property relies on other safety properties, such as the fact that no other message is broadcast, which can be proven from the text of the reactive program.

The fixed point predicate, *FP*, can be calculated from standard techniques based on the assignment statements and guards of the reactive program, where a statement of the form $\vec{x} := \vec{e}$ **if** p translates into a predicate of the form $\neg p \vee \vec{x} = \vec{e}$. The relevant portion (several conjuncts are omitted) is:

$FP(\mathcal{R}) \equiv$
  $(\textit{foreign-agent}(t).\textit{ether} = \textit{foreign-agent}(t).\textit{ether}_{network.ether[t]}$
      $\vee\ \textit{foreign-agent}(t).\textit{ether} = \textit{foreign-agent}(t).\textit{ether}_{network.ether[t]} =$
          $network.ether[t] = network.ether[t]_{foreign-agent(t).ether})$
  $\wedge\ (network.ether[t] = network.ether[t]_{mobile-node(s,\ n).ether}$
      $\vee\ network.ether[t] = network.ether[t]_{mobile-node(s,\ n).ether} =$
          $\textit{mobile-node}(s,\ n).\textit{ether} = \textit{mobile-node}(s,\ n).\textit{ether}_{network.ether[t]})$
  $\wedge\ (\textit{mobile-node}(s,\ n).\textit{ether} = \bot$
      $\vee\ \textit{mobile-node}(s,\ n).\textit{ether.dest} \neq Bcast \vee \textit{mobile-node}(s,\ n).\textit{gotflag}$
      $\vee\ last(\textit{mobile-node}(s,\ n).in) = \textit{mobile-node}(s,\ n).\textit{ether})$

It can be seen from the collection of reactive statements that each conjunct is established in one step by execution of the corresponding statement, and that eventually, the entire predicate $FP(\mathcal{R}) \wedge Advert(s, n, t, last(\textit{mobile-node}(s, n).in))$ will be established.

The remaining steps in the proof of *REGISTER* have similar proofs at their core, but we will omit them here. It remains to be shown that the advertisement is eventually processed by the mobile node, which is a straightforward application of induction on the distance of the message from the head of the input queue. Then, the registration request that is produced must be propagated to the foreign agent. The required proof is very much like the above, except that communication proceeds in the reverse direction. The foreign agent must then process the message and forward the request. This involves another induction on the distance of the request from the head of the queue, as well as the assumption that the foreign agent has space available to hold the registration. Then, the request message must propagate through the network to the home agent. This proof will rely on the assumption *NONDROP*, which states that any message sent will not be dropped. Finally, the message is processed by the home agent, requiring another inductive argument, and the proof of *REGISTER* is complete.

Proof of the next property, *PERSIST*, will rely on the real-time restrictions that have been placed on the system. It will also require the use of bounds on communication time embodied in *NONDROP*, and bounds on processing time embodied in *PROCESS*. It will also require assumptions about the processing speed of foreign agents

Finally, the proof of *DELIVERY* will be very similar to *REGISTER*, except that now an actual message must be propagated instead of a registration request. Communication from the home agent to the network, from the network to the foreign agent, and the foreign agent to the mobile node will all look very similar to the above **ensures** proof.

These proofs may seem overly complicated, but many of the lower level details, such as the propagation of messages from one component to another, could be captured in lemmas that hide this complexity from higher level reasoning. Our presentation above was at a low level in order to demonstrate the fundamental proof axioms of Mobile UNITY.

5 Conclusions

This paper has demonstrated Mobile UNITY in the context of Mobile IP, a real protocol for routing packets to mobile hosts that are transiently connected to different attachment points on the Internet. This served to illustrate how Mobile UNITY may be used to create abstractions for communication in such protocols, and as an illustration for the Mobile UNITY proof logic, which was not exercised to the same degree in [11].

While formal models capable of expressing reconfiguration have been explored from the algebraic perspective [12] and from a denotational perspective [5, 2], very few state-based models can naturally express reconfiguration of components. Also, while algebraic models such as the π-calculus may be adequate for expressing reconfiguration, it is not so clear how to handle the issue of disconnection. Recent work has recognized the importance of introducing location and failures as concepts in mobile process algebras [3, 18, 6], but these do not directly address disconnection of components that continue to function correctly but independently.

In addition to directly modeling reconfiguration and disconnection, Mobile UNITY attempts to address design issues raised by mobile computing. These issues stem from both the characteristics of the wireless connection and the nature of applications and services that will be demanded by users of the new technologies. 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 determined by their runtime location, as in the location-dependent services provided by a mobile web browser [24]. Other work has pointed out the importance of context other than location [20], such as the presence or absence of other components. The reactive statement can be viewed as an attempt to capture implicit side-effects of non-reactive statements, and to take into account the global

context in which such a statement is executed, without modifying the text of such a statement directly.

Weak consistency protocols for filesystems and databases [19, 22, 23] 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. The transient sharing abstraction was motivated in part by these types of systems.

The abstraction of communication in this environment with transiently shared variables provided very strong atomicity guarantees on the way in which they are accessed. This may seem at first to be unrealistic, but it is perfectly reasonable to expect this kind of atomicity from lower-level carrier sense and retransmit algorithms such as that provided by ethernet. Such a protocol in fact guarantees atomic access to the shared medium in a very similar way to that modeled here. Message loss, rather than being modeled as a separate action that sometimes fires, could be accommodated in the **when** predicate with additional conjuncts representing interference or other forms of communication outage.

The treatment of real-time properties was also interesting for its lack of reliance on global clocks and the expression of only the minimum synchronization constraints needed among the components, rather than requiring all clocks to be tightly synchronized. This leaves open the possibility that different clocks may run at widely different rates unless synchrony is required. This reflects the fact that global time does not really exist in distributed systems, but is only an artifact of a particular frame of reference.

The example proof illustrated the logic underlying by Mobile UNITY. It provides a formal basis for specifying properties of a system and proving them correct. Implicit communication, e.g., transient variable sharing, is handled within a reactive program that conditionally propagates new information when state changes occur. Proof of a Hoare triple in the system must take into account the effects of the reactive program and prove that it terminates. The example showed how this can be accomplished with ordinary techniques from standard UNITY.

The choice of this example shows how Mobile UNITY might be used to reason about message routing protocols. While the current example assumes a fixed infrastructure (the Internet) which serves as a backbone, the communication abstractions developed do not depend on such an infrastructure. The ether variables just as easily could have been shared directly between components, such as two mobile nodes. This would allow the expression and verification of protocols for ad-hoc routing [9], where every mobile node is also a potential message router and no fixed infrastructure is assumed. Such a study was not attempted here because such algorithms are very new and are not widely known. Also, a much more careful examination of assumptions about patterns of movement would have been necessary, as these assumptions affect protocol correctness and performance to a much greater degree.

6   Acknowledgements

Authors' address: P. McCann, Lucent Technologies, Rm 2Z-305, 263 Shuman Blvd., Naperville, IL 60566-7050. G.-C. Roman, Department of Computer Science, CB 1045/Bryan 509, Washington University, One Brookings Dr., St. Louis, MO 63130-4899.

References

[1] Martín Abadi and Leslie Lamport. An old-fashioned recipe for real time. *ACM Transactions on Programming Languages and Systems*, 16(5):1543–71, 1994.

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

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

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

[5] William D. Clinger. Foundations of actor semantics. Technical Report AI-TR-633, MIT Artificial Intelligence Laboratory, 1981.

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

[7] V. Fuller, T. Li, J. Yu, and K. Varadhan. RFC 1519: Classless inter-domain routing (CIDR): an address assignment and aggregation strategy, September 1993. Obsoletes RFC1338.

[8] C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580,583, 1969.

[9] David B. Johnson. Routing in ad hoc networks of mobile hosts. In *Proceedings of the Workshop on Mobile Computing Systems and Applications*, pages 158–163, Santa Cruz, CA, 1994. IEEE.

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

[11] Peter J. McCann and Gruia-Catalin Roman. Compositional programming abstractions for mobile computing. *IEEE Transactions on Software Engineering*, 24(2):97–110, February 1998.

[12] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes. I. *Information and Computation*, 100(1):1–40, 1992.

[13] Jayadev Misra. A logic for concurrent programming: Progress. *Journal of Computer and Software Engineering*, 3(2):273–300, 1995.

[14] Jayadev Misra. A logic for concurrent programming: Safety. *Journal of Computer and Software Engineering*, 3(2):239–72, 1995.

[15] C. Perkins. RFC 2002: IP mobility support, October 1996.

[16] D. Plummer. RFC 826: Ethernet Address Resolution Protocol: Or converting network protocol addresses to 48.bit ethernet address for transmission on Ethernet hardware, November 1982.

[17] J. Postel. RFC 925: Multi-LAN address resolution, October 1984.

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

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

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

[21] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison-Wesley, Reading, Massachusetts, 1994.

[22] Carl D. Tait and Dan Duchamp. An efficient variable consistency replicated file service. In *Proceedings of the USENIX File Systems Workshop*, pages 111–126, Ann Arbor, MI, 1992.

[23] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. *Operating Systems Review*, 29(5):172–83, 1995.

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

program *mobile-node*(*s*, *n*) **at** λ
  declare
    care-of : address
  [] *in* : queue of message
  [] *ether* : message
  [] *clock, begintime, lasttime, transcount* : natural
  [] *transbuffer* : message
  [] *confirmed, gotflag* : boolean
  always
    Hdata $\equiv$ *head*(*in*).*data*
  [] *Bcast* $\equiv$ (-1, -1)
  [] *IsRegular*(*msg*) $\equiv$ *msg.type* = *Regular*
  [] *IsForeignAd*(*msg*) $\equiv$
          msg.type = *Advertisement* $\wedge$ *msg.data.foreignflag*
        $\wedge$ *msg.source.subnet* $\neq$ *s* $\wedge$ *msg.source* $\neq$ *care-of*
  [] *IsHomeAd*(*msg*) $\equiv$
          msg.type = *Advertisement* $\wedge$ *msg.data.homeflag*
        $\wedge$ *msg.source* = (*s*, *HA*) $\wedge$ (*s*, *HA*) $\neq$ *care-of*
  [] *IsDeniedReply*(*msg*) $\equiv$
          msg.type = *Reply* $\wedge$ *msg.data.flag* = *Denied*
        $\wedge$ *confirmed* = *false* $\wedge$ *msg.data.ID* = *transbuffer.data.ID*
  [] *IsOkReply*(*msg*) $\equiv$
          msg.type = *Reply* $\wedge$ *msg.data.flag* = *Ok*
        $\wedge$ *confirmed* = *false* $\wedge$ *msg.data.ID* = *transbuffer.data.ID*
  [] *IsGarbage*(*msg*) $\equiv$
          $\neg$ *IsRegular*(*msg*) $\wedge$ $\neg$ *IsForeignAd*(*msg*) $\wedge$ $\neg$ *IsHomeAd*(*msg*)
        $\wedge$ $\neg$ *IsDeniedReply*(*msg*) $\wedge$ $\neg$ *IsOkReply*(*msg*)
  initially
    in = ϵ [] $\lambda = \lambda_s$ [] *confirmed* = *true* [] *lasttime* = 0 [] *care-of* = (*s*, *HA*)
  assign
    {Move to a new location}
    λ := Move(λ)

    {Transfer appropriate messages from link to input queue}
  [] *in, ether* := *in* $\bullet$ *ether*, $\perp$ **reacts-to** *ether* $\neq$ $\perp$ $\wedge$ *ether.dest* = (*s*, *n*)

    {For broadcast, message is not consumed – need *gotflag* for idempotency}
  [] *in, gotflag* := *in* $\bullet$ *ether*, *true* **reacts-to** *ether* $\neq$ $\perp$ $\wedge$ *ether.dest* = *Bcast* $\wedge$ $\neg$ *gotflag*
  [] *gotflag* := *false* **reacts-to** *ether* = $\perp$

Figure 4: The first part of program *mobile-node*, which sends and receives messages.

{Generate away-from-home registration requests for later transmission}
⫿ *in, care-of, transbuffer, confirmed, begintime, transcount :=*
    tail(in),
    head(in).source,
    ((s, n), head(in).source, Request, ((s, n), head(in).source, NewID)),
    false,
    clock,
    0
        if *in ≠ ε ∧ IsForeignAd(head(in))*

{Generate at-home de-registration requests for later transmission}
⫿ *in, care-of, transbuffer, confirmed, begintime, transcount :=*
    tail(in),
    (s, HA),
    ((s, n), (s, HA), Request, ((s, n), (s, HA), NewID)),
    false,
    clock,
    0
        if *in ≠ ε ∧ IsHomeAd(head(in))*

{Transmit the buffer until we receive a reply}
⫿ *transmit ::*
⟨ *ether, lasttime, transcount := transbuffer, clock, transcount + 1*
        if *confirmed = false ∧ ether = ⊥ ∧ (clock - lasttime) > 2^{transcount}*
        ∧ (clock - begintime) < Lifetime;
    ether := ⊥ ⟩

{Process denied *Reply* messages}
⫿ *in, care-of := tail(in), ⊥*
        if *in ≠ ε ∧ IsDeniedReply(head(in))*

{Process ok *Reply* messages}
⫿ *in, confirmed := tail(in), true*
        if *in ≠ ε ∧ IsOkReply(head(in))*

{Enable re-registration when not at home and Lifetime is about to expire}
⫿ *reregister ::*
confirmed, begintime, transcount, transbuffer.data.ID := false, clock, 0, NewID
        if *care-of ≠ (s, HA) ∧ (clock - begintime) > (Lifetime - D)*

{Consume application messages}
⫿ *in := tail(in)* **if** *in ≠ ε ∧ IsRegular(head(in))*

{Transmit application messages}
⫿ ⟨ *ether := ((s, n), NewDest, Regular, NewData)* **if** *ether = ⊥* ;
ether := ⊥ ⟩

{Consume garbage messages}
⫿ *in := tail(in)* **if** *in ≠ ε ∧ IsGarbage(head(in))*

{Occasionally increment the clock}
⫿ *timer :: clock := clock + 1*

{Force re-registration when *Lifetime* is very close to expiring}
⫿ **inhibit** *timer* **when** *care-of ≠ (s, HA) ∧ (clock - begintime) > (Lifetime - D/2)*

{Force re-transmission within a reasonable delay}
⫿ **inhibit** *timer* **when** *confirmed = false ∧ (clock - lasttime) > 2^{transcount} + 1*

end

Figure 5: The program *mobile-node* (continued).

17

```
program home-agent(s) at λ
  declare
    in : queue of message
  [] ether : message
  [] InterestSet : set of address
  [] home_forward : array[Nnodes] of address
  [] home_rtime : array[Nnodes] of integer
  [] clock : integer
  always
    Hdata≡ head(in).data
  [] Addr ≡ head(in).dest
  [] Bcast ≡ (-1, -1)
  [] NoNode ≡ (0, -1)
  [] IsLocalRequest(i, msg) ≡
              msg.type = Request ∧ msg.source = (s, i)
          ∧  msg.dest = (s, HA)
  [] IsRemoteRequest(i, msg) ≡
              msg.type = Request ∧ msg.data.home = (s, i)
          ∧  msg.data.foreign.subnet ≠ s
  [] IsMessage(i, msg) ≡
              msg.dest = (s, i) ∧ home_forward[i] ≠ NoNode
  [] IsGarbage(i, msg) ≡
          ¬ IsLocalRequest(i, msg) ∧ ¬ IsRemoteRequest(i, msg)
          ∧  ¬ IsMessage(i, msg)
  initially
    InterestSet = {(s, HA)} ∧ ⟨ ∀ i : 0 ≤ i < Nnodes :: home_forward[i] =
NoNode ⟩
  assign
    {Transfer appropriate messages from link to input queue}
  [] in, ether := in • ether, ⊥
        reacts-to ether ≠ ⊥ ∧ ether.dest ∈ InterestSet
```

Figure 6: The first part of program *home-agent*, which processes registration requests on behalf of mobile nodes.

] ⟨ []$i : 0 \leq i < Nnodes$ ::
    {Remote request processing and reply generation}
    ⟨ $in, ether, home\_forward[Hdata.home.node], InterestSet, home\_rtime[i]$:=
        $tail(in)$,
        $((s, HA), Hdata.foreign, Reply, (Hdata.home, Ok, Hdata.ID))$,
        $Hdata.foreign$,
        $InterestSet \cup \{Hdata.dest\}$,
        $clock$
        if $in \neq \epsilon \wedge ether = \perp \wedge IsRemoteRequest(i, head(in))$;
      $ether := \perp$ ⟩

    {De-register those that arrive home}
  [] ⟨ $in, ether, home\_forward[i], InterestSet$:=
        $tail(in)$,
        $((s, HA), head(in).source, Reply, (Hdata.home, Ok, Hdata.ID))$,
        $NoNode$,
        $InterestSet - \{(s, i)\}$
        if $in \neq \epsilon \wedge ether = \perp \wedge IsLocalRequest(i, head(in))$;
      $ether := \perp$ ⟩

    {Encapsulated routing}
  [] ⟨ $in, ether := tail(in), ((s, HA), home\_forward[i], Encapsulated, head(in))$
        if $in \neq \epsilon \wedge ether = \perp \wedge IsMessage(i, head(in))$;
      $ether := \perp$ ⟩

    {Timeout a forwarding address}
  [] $timeout\text{-}fwd$ :: $home\_forward[i], InterestSet$:=
        $NoNode$,
        $InterestSet - \{(s, i)\}$
        if $(clock - home\_rtime[i]) > Lifetime$
⟩
    {Consume garbage packets}
] $in := tail(in)$ **if** $in \neq \epsilon \wedge$ ⟨ $\forall i : 0 \leq i < Nnodes :: IsGarbage(i, head(in))$ ⟩
    {Transmit periodic advertisements}
] $advertise$ ::
  ⟨ $ether := ((s, HA), Bcast, Advertisement, (home, false))$ **if** $ether = \perp$;
    $ether := \perp$ ⟩
    {Occasionally increment the clock}
] $timer$ :: $clock := clock + 1$
end

Figure 7: The program *home-agent* (continued).

19

program *foreign-agent(s)* **at** λ
  declare
    in : queue of message
  [] *ether* : message
  [] *visitor_assign* : array[*Nvisitors*] of address
  [] *visitor_state* : array[*Nvisitors*] of $\in\{Pending, Confirmed\}$
  [] *visitor_rtime* : array[*Nvisitors*] of integer
  [] *clock* : integer
  always
    $Hdata \equiv head(in).data$
  [] $Addr \equiv head(in).dest$
  [] $Bcast \equiv (-1, -1)$
  [] $NoNode \equiv (0, -1)$
  [] $IsNewRequest(i, t, n, msg) \equiv$
              $msg.type \equiv Request \wedge visitor\_assign[i] = NoNode$
        $\wedge\ msg.source = (t, n) \wedge t \neq s$
        $\wedge\ \langle\ \forall j : visitor\_assign[j] \neq (t, n)\ \rangle$
  [] $IsReRequest(i, t, msg) \equiv$
              $msg.type = Request \wedge visitor\_assign[i] = (t, n)$
        $\wedge\ msg.source = (t, n)$
  [] $IsBusyRequest(t, n, msg) \equiv$
              $msg.type = Request \wedge msg.source = (t, n) \wedge t \neq s$
        $\wedge\ \langle\ \forall j : visitor\_assign[j] \neq NoNode\ \rangle$
        $\wedge\ \langle\ \forall j : visitor\_assign[j] \neq (t, n)\ \rangle$
  [] $IsOkReply(i, msg) \equiv$
              $msg.type = Reply \wedge msg.data.flag = Ok$
        $\wedge\ visitor\_assign[i] = msg.data.home$
  [] $IsDeniedReply(i, msg) \equiv$
              $msg.type = Reply \wedge msg.data.flag = Denied$
        $\wedge\ visitor\_assign[i] = msg.data.home$
  [] $IsEncapsulated(i, msg) \equiv$
              $msg.type = Encapsulated \wedge visitor\_assign[i] = msg.data.dest$
  [] $IsGarbage(i, t, n, msg) \equiv$
              $\neg IsNewRequest(i, t, n, msg) \wedge \neg IsReRequest(i, t, n, msg)$
        $\wedge\ \neg IsBusyRequest(t, n, msg) \wedge \neg IsOkReply(i, msg)$
        $\wedge\ \neg IsDeniedReply(i, msg) \wedge \neg IsEncapsulated(i, msg)$
  initially
    $\langle\ [] i : 0 \leq i < Nvisitors :: visitor\_assign[i] = NoNode\ \rangle$

Figure 8: The first part of program *foreign-agent*, which receives registration requests from mobile nodes and forwards them to the appropriate home agent.

assign
  {Transfer appropriate messages from link to input queue}
▯ *in, ether := in • ether,* ⊥ **reacts-to** *ether* ≠ ⊥ ∧ *ether.dest =* (*s, FA*)

▯ ⟨ ▯*i, t, n :* 0 ≤ *i* < *Nvisitors* ∧ 0 ≤ *t* < *Nsubnets* ∧ 0 ≤ *n* < *Nnodes* ::
     {Registration request generation for visitors}
     process-request ::
     ⟨ *in, ether, visitor_assign*[*i*]*, visitor_state*[*i*]*, visitor_rtime*[*i*] :=
          tail(*in*),
          ((*s, FA*), (*Hdata.home.subnet, HA*)*, Request, Hdata.data*),
          (*t, n*),
          Pending,
          clock
          if *in* ≠ *ε* ∧ *ether =* ⊥
              ∧ (*IsNewRequest*(*i, t, n, head*(*in*)) ∨ *IsReRequest*(*i, t, n, head*(*in*)));
     ether := ⊥ ⟩

     {Deny registration if we're too busy}
▯ ⟨ *in, ether :=*
          tail(*in*),
          ((*s, FA*)*, (t, n), Reply, (Hdata.home, Denied, Hdata.ID*)),
          if *in* ≠ *ε* ∧ *ether =* ⊥ ∧ *IsBusyRequest*(*t, n, head*(*in*));
     ether := ⊥ ⟩

     {Ok reply processing}
▯ ⟨ *in, ether, visitor_state*[*i*]*, visitor_rtime*[*i*] :=
          tail(*in*),
          ((*s, FA*)*, visitor_assign*[*i*]*, Reply, Hdata*)
          Confirmed,
          clock
          if *in* ≠ *ε* ∧ *ether =* ⊥ ∧ *IsOkReply*(*i, head*(*in*));
     ether := ⊥ ⟩

     {Denied reply processing}
▯ ⟨ *in, ether, visitor_assign*[*i*] :=
          tail(*in*),
          ((*s, FA*)*, visitor_assign*[*i*]*, Reply, Hdata*),
          NoNode
          if *in* ≠ *ε* ∧ *ether =* ⊥ ∧ *IsDeniedReply*(*i, head*(*in*));
     ether := ⊥ ⟩

     {Decapsulation and delivery}
▯ ⟨ *in, ether := tail*(*in*)*, Hdata*
          if *in* ≠ *ε* ∧ *ether =* ⊥ ∧ *IsEncapsulated*(*i, head*(*in*));
     ether := ⊥ ⟩

     {Time-out visitor slots}
▯ *timeout-visit*(*i, t, n*)::
     visitor_assign[*i*] := *NoNode* **if** (*visitor_rtime*[*i*] - *clock*) > *Lifetime*
⟩

Figure 9: The program *foreign-agent* (continued).

21

{Consume garbage messages}
⟦ $in := tail(in)$ **if** $in \neq \epsilon \wedge \langle \forall i, t, n : 0 \leq i < Nvisitors$
$\wedge\ 0 \leq t < Nsubnets$
$\wedge\ 0 \leq n < Nnodes ::$
$IsGarbage(i, t, n, head(in)) \rangle$

{Transmit periodic advertisements}
⟦ ⟨ $ether := ((s,\ FA),\ Bcast,\ Advertisement,\ (false,\ foreign))$ **if** $ether = \bot$;
$ether := \bot$ ⟩

{Occasionally increment the clock}
⟦ $timer :: clock := clock + 1$
end

Figure 10: The program *foreign-agent* (continued).

System *mobile-ip*
...Program declarations from above...

Components
network
⟦ ⟨ ⟦$s : 0 \leq s < Nsubnets ::$ *home-agent*(s) **at** λ_s ⟩
⟦ ⟨ ⟦$s : 0 \leq s < Nsubnets ::$ *foreign-agent*(s) **at** λ_s ⟩
⟦ ⟨ ⟦$s, n : 0 \leq s < Nsubnets \wedge 0 \leq n < Nnodes ::$ *mobile-node*(s, n) **at** λ_s ⟩

Interactions
{Attach the network to the agents (transiently shared variables)}
⟨ ⟦$s : 0 \leq s < Nsubnets ::$
$network.ether[s] \approx home\text{-}agent(s).ether$
⟦ $network.ether[s] \approx foreign\text{-}agent(s).ether$
⟩

{Attach the network to the mobile nodes, when co-located}
⟦ ⟨ ⟦$s, t, n : 0 \leq s < Nsubnets \wedge 0 \leq t < Nsubnets \wedge 0 \leq n < Nnodes ::$
$network.ether[s] \approx mobile\text{-}node(t, n).ether$ **when** $mobile\text{-}node.\lambda = \lambda_s$
⟩

{Specify timing constraints}
⟦ ⟨ ⟦$s, t, n : 0 \leq s < Nsubnets \wedge 0 \leq t < Nsubnets \wedge 0 \leq n < Nnodes ::$
$home\text{-}agent(s).clock$ **synch** $mobile\text{-}node(s, n).clock$
when $mobile\text{-}node(s, n).ether = ((s, n), (s, HA), \text{-}, \text{-}, \text{-})$
within 1.01

⟦ $home\text{-}agent(s).clock$ **synch** $mobile\text{-}node(s, n).clock$
when $home\text{-}agent(s).ether = ((s, HA), (s, n), \text{-}, \text{-}, \text{-})$
within 1.01

⟦ $foreign\text{-}agent(s).clock$ **synch** $mobile\text{-}node(t, n).clock$
when $mobile\text{-}node(t, n).ether = ((t, n), (s, FA), \text{-}, \text{-}, \text{-})$
within 1.01

⟦ $foreign\text{-}agent(s).clock$ **synch** $mobile\text{-}node(t, n).clock$
when $foreign\text{-}agent(s).ether = ((s, FA), (t, n), \text{-}, \text{-}, \text{-})$
within 1.01
⟩

end

Figure 11: The complete system *mobile-ip*.

22