Reasoning About Active Network Protocols*

Samrat Bhattacharjee (Corresponding Author)
Kenneth L. Calvert
Ellen W. Zegura
Networking and Telecommunications Group
College of Computing
Georgia Tech
Atlanta, GA 30332-0280
{bobby,calvert,ewz}@cc.gatech.edu

Abstract

Active Networks allow users to "program" the network infrastructure, by injecting information that describes or controls a distributed algorithm to be executed for the user by the network infrastructure. The nature of the services that can be implemented with such a facility is determined by the programming interface to the active network, i.e. the set of abstractions it exposes to users. The complexity of this interface may range from a few simple parameters to a completely general programming language.

We present a formal model that supports reasoning independently about the correctness of both the underlying active network platform and the algorithms injected into it, in a manner that admits the full range of possible programming interfaces. The model is defined using the UNITY formalism, which facilitates reasoning about distributed computations in a platform- and language-independent way. The interaction between the underlying platform and the user-injected program is captured in a specialized form of program composition that allows properties of each to be preserved. The use of the model is illustrated via an example dealing with mobility.

1 Introduction

Active networks provide a programmable platform on which network services can be defined or altered by injecting code or other information into the nodes of the network. This paradigm offers a number of potential advantages, including the ability to develop and deploy new network protocols and services quickly, and the ability to customize services to meet the different needs of different classes of users.

Active networks also raise a number of interesting issues. For example, what programming model should the network support? What abstractions are available, and how can the programmer reason about the global correctness of a service implemented with them? How can the overall stability of the network be preserved? What are the mechanisms for injecting code into the network? These are important questions because distributed algorithms are notoriously difficult to get right; adding the ability to modify a node's behavior *on the fly* clearly adds another level of complexity.

In this paper we present a programming model for active networks that (i) constrains the degree to which an active node's behavior can be modified, and (ii) supports rigorous reasoning about the global behavior of the network. Our approach is to define —formally— a generic network node behavior that can be customized by means of simple instructions inserted into specific "slots" in that behavior. The slots define the interface to the generic node behavior and the degree to which that behavior can be modified.

*This work supported by DARPA under contract number N66001-97-C-8512.

This approach ensures that properties of the global network behavior are preserved, provided the injected code satisfies certain conditions.

We illustrate the approach in this paper using the UNITY formalism both to describe the generic node behavior (as a program) and to represent the injected code. We chose UNITY because of its simplicity: UNITY models interaction between different program components in the same way whether they are distributed (i.e. located at different nodes) or not. Thus, we use the same logical mechanisms to reason about the global network behavior as about the interaction between the generic node program and the injected code. Although this tends to complicate reasoning about local behavior, the ability to reason about global properties comes for free.

The rest of this paper is organized as follows. The next section defines the problem, and highlights the relationship between the network API and the possible reasoning methods. We also place our approach in the context of other active network research. Section 3 introduces our model of the active network, defining the general form of the fixed and variable parts of the network's behavior using the UNITY formalism. Section 4 presents an example illustrating the theory developed in Section 3, comprising a basic node program and an injected program that forwards messages toward a mobile resource that migrates throughout the network. Finally, Section 5 offers some conclusions.

2 Background and Related Work

The high-level goal of active networking is to define a dynamically-programmable network platform or "API" on which network services can be built. Here we consider how different approaches to that problem have different effects on our ability to reason about the problem. We lay out the problem in abstract form, and also describe some approaches under investigation elsewhere and how they relate to ours.

2.1 Problem Statement

We model the network itself as a collection of *nodes*, which communicate by sending packets over *channels*; the nodes and channels are arranged in some connected but otherwise unspecified topology. We assume for simplicity that each node of the network exports the same API, and furthermore each node exhibits the same basic behavior, which consists of repeatedly removing a packet from an incoming channel and then taking some action based upon the information contained in the packet and the current state of the node. As a result of this action the state of the node may be modified, and packets may be queued for transmission on outgoing channels.

We view the behavior of each active node as being made up of two components: a fixed part, which is the same for every packet; and a variable part, which is determined by the information carried in packets plus the node state. The fixed part, in effect, defines the "virtual machine" presented to the programmer, while the variable part consists of the program that is fed into that virtual machine, plus the input fed into that program-running-on-virtual-machine. In what follows, we refer to the fixed part of the node behavior as the *underlying program*, and the "program" portion of the variable part as the *injected program*.

Two key issues in the design of an active network API are: (i) the nature of the virtual machine defined by the underlying program, and (ii) the mechanisms for "injecting" the program defining the variable part of the behavior. These issues have a profound affect on our ability to reason about the global behavior of the network. If the underlying program defines a Turing-complete interpreter —as exemplified by, say, the Java Virtual Machine [2]— and we place no restrictions on the injected programs, then essentially all of the node behavior is determined by the injected program. In that case it is difficult to make any general statements about the global network behavior without complete knowledge of the injected program. Moreover, it is difficult or even impossible to reason simultaneously about the injection *process* (in which the injected program is treated by the network as data) *and* the global behavior of the network under the control of the injected program.

On the other hand, if the underlying program defines a fixed computation, to which the injected program merely supplies scalar parameters —say, menu selections which define a path through the program— then the node behavior is completely defined by the underlying program, and it is possible to make strong statements about the global behavior of the network. Moreover it is straightforward to reason about the injection process (at least in theory) because the number of possible node behaviors during the process is finite. The drawback at this end of the spectrum is that flexibility is taken away: the set of possible behaviors is completely defined by the underlying program.

Clearly an approach that achieves a middle ground between these two extremes is desirable. Such an approach defines part of the active node's behavior by the underlying program, and part by the injected program. The goal is to be able to make useful statements about the network's global behavior based on the fixed part of each node's behavior and on certain constraints or assumptions about the injected program. At the same time, it should be possible to extend the network's behavior in an infinite variety of ways via injected programs that satisfy the constraints. Ideally, the underlying program would be judiciously defined so that these constraints can be checked syntactically at the time the program is "injected". The approach described in this paper is of this middle-of-the-road type.

While this approach makes it possible to reason about global behavior independent of injected code, it does *not* solve the problem of reasoning about the injection process itself, i.e. how the injected program propagates through the network, and goes from being "data" to being "program". This is an interesting and important problem, which we do not consider in this paper, assuming instead that the active network is in a state where the injected program is in place at every node.

The problem of defining an active network API raises a number of other issues —security, scalability, and resource management, to name just a few— that we also assume away in the interest of brevity and separation of concerns. Thus our model posits a single active network user, and we do not consider performance or other real-time aspects.

2.2 Other Approaches

The SwitchWare active network architecture [5] developed at U. Penn and Bellcore defines two levels of programming, but in a somewhat different manner than what is described above. The "packet" level is a scripting language that provides for invocation and composition of lower-level services, but has little functionality of its own. Programs written in this scripting language are carried in packets and interpreted by the fixed part of node behavior.

The lower, or "service" level defines the functions invoked by packet level scripts. What we would call the "injected program" is a set of services defined in this language; once installed in nodes they can be invoked by scripts. The current version of the two-level architecture uses a new language called PLAN (Programming Language for Active Networks) as the packet level scripting language [3], and Java as the service level language. PLAN is a simple language based on typed lambda calculus. The language itself allows some generic properties (e.g., guaranteed termination, strong typing) to be asserted about individual node behaviors, independent of the injected program. However, PLAN depends on the injected (service-level) program to implement global functionality, and is in this sense a kind of "dual" of our approach.

The ANTS toolkit [6], developed at MIT, roughly corresponds to the "universal Turing machine" model of fixed behavior described above. In ANTS, both underlying and injected programs are specified in Java. Packets carry an identifier that indicates the method to be used in processing the packet and parameters specific to the referenced routine. Some methods are well-known and available at every active node (fixed part); other routines are injected using an in-band, on-demand code-loading mechanism. ANTS relies on mobile code techniques such as sandboxing to obtain generic guarantees on network behavior; however, it is unclear how one would reason formally about global properties of the network programmed in Java.

Both ANTS and PLAN feature a resource usage bound carried in each packet. The fixed behavior of each node includes decrementing this value upon receipt and discarding the packet if the result is zero. Any additional packets created during packet processing inherit a resource bound from the original packet. Precise policies, and associated guarantees, regarding the allocation of resources to these addtional packets have not been developed.

3 Formal Model

In this section we describe the relationship and interface between the underlying program and the injected program. This interface takes the form of one or more *slots*; for the purposes of this (general) discussion, slots are identified by natural numbers. Users can inject programs that *bind* to particular slots. The underlying program invokes the code bound to each slot at some point during its execution, by *raising* the slot. The semantics of the particular underlying program determine the exact conditions under which the slot is raised. Once a slot is raised, the underlying program suspends and the injected program runs until completion or until it exhausts its resources. All communication between the injected program and the underlying program is by shared variables.

In what follows we present our results using the UNITY program notation and logic. The reader who is unfamiliar with UNITY is referred to the appendix for a brief introduction.

We first describe the form of the underlying program, in terms of the program executed at an arbitrary node v. Then we describe the form required of an injected program, and the transformation that models the injection process itself. Finally, we state some general results about injection and property preservation.

3.1 Form of the Underlying Program

The underlying program is assumed to be uniform in the sense that every node of the network executes the same algorithm. Although this assumption can probably be relaxed, it does not seem a serious limitation and greatly simplifies the presentation. In what follows we describe the underlying program in terms of the program running at an arbitrary node v.

The underlying program is required to have a certain structure, namely, the union of two programs: $N \parallel DS$. Program N implements the main packet-processing algorithm (we give an example of such an algorithm later in the paper) and may also invoke (raise) one or more slots during its execution. Program DS is responsible for ensuring that the execution of the underlying program resumes after a slot has been raised.

In order to interface with injected programs, $N \parallel DS$ has certain variables. The variable $v.state$ defines the current node state. Setting this variable to $slot.i.raise$ raises slot i; only N sets $v.state$ to $slot.i.raise$. It is also possible for $v.state$ to be set to other values by N. We use the abbreviation $v.slot.i.raise$ for $v.state = slot.i.raise$.

Setting $v.state$ to $slot.i.complete$ indicates that slot processing for slot i has completed; program DS is responsible for setting $v.state$ to this value. We use the abbreviation $v.slot.i.complete$ for $v.state = slot.i.complete$. For each slot i, the natural number $v.rt.i.usage$ is a count of the resources used by the code (if any) bound to slot i, while $v.rt.i.bound$ is the usage bound, i.e. the maximum permissible value of $v.rt.i.usage$. The boolean variable $v.Progress.i$ is used in ensuring that slot processing does not deadlock (see below).

The program DS has the following form:

```
**Program** {**Default Slot**} $DS$
 **initially**
 **D0**    $\langle \parallel i :: v.rt.i.usage, v.rt.i.bound = 0, \beta_i \rangle$                { Initialization, $\beta_i \geq 0$ }
**always**
 **D1**    $\langle \parallel i :: v.SlotCondition.i = v.rt.i.bound > v.rt.i.usage \wedge v.slot.i.raise \rangle$
                                   { Default set of conditions for progress through slot }
 **D2**    $\langle \parallel i :: v.Progress.i = Q.i \rangle$                              { "default" predicate $Q$ }
**assign**
 **D3**    $\langle \parallel i :: v.rt.i.usage := v.rt.i.usage + 1$ if $v.SlotCondition.i \wedge v.Progress.i \rangle$
                                   { Increase resource usage if no other program active }
 **D4**  $\parallel$ $\langle \parallel i :: v.state := v.slot.i.complete$ if $v.slot.i.raise \wedge v.rt.i.bound = v.rt.i.usage \rangle$
                                   { Resource bound exhausted, slot processing complete }
```

The program DS provides a "default slot behavior" that —provided the predicate $Q.i$ is weak enough— ensures that slot processing completes, whether any injected program is bound to the slot or not. The form of DS is specified completely, except for the predicate $Q.i$; the "default" value for $Q.i$ (i.e. without any injected code) is *true*.

We say that program $N \,[\!]\, DS$ is *receptive* if it satisfies the following requirements:

(U0) The variables of the program can be partitioned into classes:

- C: variables related to the slot control mechanism. This class contains (only) $v.state$ and the natural numbers $v.rt.i.usage$, $v.rt.i.bound$, and the boolean $v.Progress.i$, for each slot i.
- R: variables that can be *read* (not written) by injected programs.
- W: variables that can be *read* or *written* by injected programs.
- X: all other variables of the program.

(Note that these classes relate to the accesses permitted to the *injected* program, not the underlying program.)

(U1) For each i, and any variable $r \in R$, the property

$$slot.i.raise \wedge r = k \ \mathit{unless}\ slot.i.complete$$

holds in the program. (This says that variables that can be read by the injected program do not change their values while a slot is raised.)

(U2) For each i, the property

$$\begin{array}{l} v.slot.i.raise \wedge v.rt.i.usage = k \wedge k < v.rt.i.bound \qquad \mathit{unless} \\ \quad v.slot.i.raise \wedge v.rt.i.usage > k \wedge v.rt.i.usage \leq v.rt.i.bound \end{array}$$

holds in N. (This says that as long as slot i is raised, the statements in N only increase the resource counter for i and do not increase it beyond its bound.)

(U3) For each i, the property

$$v.slot.i.raise \ \mathit{unless}\ v.slot.i.complete \wedge v.rt.i.usage = v.rt.i.bound$$

holds in N. (This says that slots "terminate" only when their resource allocations have been used up.)

(U4) For each i, the property

$$v.slot.i.raise \mapsto v.slot.i.complete$$

holds in $N \,[\!]\, DS$. (This says that each slot eventually completes.) A sufficient condition for this property to hold, given (U2) and (U3) and the above definition of DS, is $Q.i \equiv \mathit{true}$.

The property of being receptive ensures that the underlying program has a form that is compatible with the interface expected by injected programs; we define this latter interface next.

3.2 Injecting Programs

For each program J to be injected, a number designating the slot to which J is to be bound must be specified at injection time. More than one program may be bound to a given slot. A program J that is to be injected into underlying program $N \,[\!]\, DS$ is required to satisfy the following structural constraints:

(J0) All variables named in both N and J are in both R and W.

(J1) No variable in R occurs on the left-hand side of an assignment statement in J.

A program satisfying these constraints is said to be *acceptable* to N.

Our model postulates that the injection process "installs" the same code in every node of the network; during the process of injection, both N and J are transformed, by modifying their program statements as follows. Let the slot specified for J be i, and let each statement s of J be of the form:

$$\begin{array}{rcll} x_s & := & e_{s0} & \text{if } b_{s0} \\ & & \ldots & \\ & \sim & e_{sn} & \text{if } b_{sn} \end{array}$$

We define the injection of J into N, denoted by $Inj(N, J)$, to be the program $N \,[\!]\, DS' \,[\!]\, J'$, where DS' is DS with the statement **D2** for slot i modified to be:

$$v.Progress.i \;=\; Q.i \wedge \langle \forall\, s : s \in J : \neg b_{s0} \wedge \ldots \wedge \neg b_{sn} \rangle$$

and J' is obtained from J by modifying each statement s (assumed to have the form given above) to be:

$$\begin{array}{rrcll} & \bar{x} & := & \bar{e_{s0}} & \text{if } b_{s0} \wedge v.SlotCondition.i \\ & & & \ldots & \\ & & \sim & \bar{e_{sn}} & \text{if } b_{sn} \wedge v.SlotCondition.i \\ \| & v.rt.i.usage & := & v.rt.i.usage + 1 & \text{if } (b_{s0} \vee \ldots \vee b_{sn}) \wedge v.SlotCondition.i \end{array}$$

These modifications ensure that (i) each statement of the injected program increases the resource limit if it is effective, and (ii) the predicate $v.Progress.i$ is true whenever none of the statements of the injected program is effective, so that the resource usage counter will increase and the slot will eventually be completed, even if the injected program deadlocks.

We denote the program resulting from injecting J into the underlying program N by $Inj(N, J)$.

3.3 Properties of Injection

We now present some properties that follow from the foregoing definitions. In what follows let $N \,[\!]\, DS$ be a receptive program, and let I and J be acceptable to N.

Lemma 1. $Inj(N, J)$ is receptive.

Proof. We show that $Inj(N, J)$ satisfies conditions U0-U4:

Ad (U0): The variables of $Inj(N, J)$ can be partitioned as before, with the variables of J that are not in R or W being members of X.

Ad (U1): The statements of N are not changed, and none of the statements from J or DS writes to variables in R, so this property still holds.

Ad (U2): The original statements of N are not changed; the modified statements of J only increase the value of $v.rt.i.usage$, as required.

Ad (U3): This holds by a similar argument to the one for (U2).

Ad (U4): Because $N \,[\!]\, DS$ is receptive, there exists a proof of this property for the original $N \,[\!]\, DS$. The only steps of that proof that might have been invalidated are *ensures* properties that depend on the assignment statement **DS3**, whose guard was strengthened. The strongest claim any such property can make is that $v.rt.i.usage$ increases by 1. At any state where **DS3** is no longer effective because its guard was strengthened, some (modified) statement of J is enabled, and that statement will also increase $v.rt.i.usage$ by 1. Thus the original property p *ensures* q can be replaced by a set of properties $p \wedge b_{s0}$ *ensures* q, $p \wedge b_{s1}$ *ensures* q, etc., which, together with the modified ensures property based on the modified **DS3**, yield $p \mapsto q$ by disjunction. The rest of the proof proceeds unchanged.

The following result shows that our definitions are robust, and in some sense commute with union (parallel composition).

Theorem 2. Let $N \, [] \, DS$ be receptive, and let I and J be acceptable to N. Then any property P holds in $\mathit{Inj}(\mathit{Inj}(N, J), I)$ if and only if P holds in $\mathit{Inj}(N, I \, [] \, J)$.

Proof Sketch. Because injection does not modify the statements of N, and the modifications to DS due to repeated injections commute, it is possible to define a bijection between the statements of $\mathit{Inj}(\mathit{Inj}(N, J), I)$ and the statements of $\mathit{Inj}(N, I \, [] \, J)$ such that each assignment statement has the same weakest precondition operator as its corresponding statement. This can be used to show that any proof of P in $\mathit{Inj}(\mathit{Inj}(N, J), I)$ is a valid proof of P in $\mathit{Inj}(N, I \, [] \, J)$, and vice versa.

The next theorem says that any property of N that holds when N is composed with well-behaved programs is preserved by injection, if the injected program is also well-behaved.

Theorem 3. Let $N \, [] \, DS$ be receptive, and let J be acceptable to N. Let P be a set of safety properties that mention only variables in R and W, and let Q be any property that mentions only variables of $N \, [] \, DS$. If

Hypothesis: P in H
Hypothesis: properties in (U1–U4) in H
Conclusion: Q in $N \, [] \, H$

and

P in J

then

Q in $\mathit{Inj}(N, J)$.

Proof Sketch. The proof of Q in $N \, [] \, H$ whose existence is asserted in the hypothesis is valid for $\mathit{Inj}(N, J)$ as well, because the properties in the hypothesis hold in $\mathit{Inj}(N, J)$, and the statements of N are not modified by injection.

The next two results show how properties of the injected program are preserved by injection. These theorems require that properties be proved of the composite of the injected program and the underlying program, but allow abstraction from the mechanics of the slot mechanisms. For these theorems, we say a predicate is *pure* if it does not depend on any variables other than those in R, W, and variables of J.

Theorem 4. Let N be receptive, and let J be an acceptable program with respect to N. Let p and q be pure predicates. If p *unless* q in $N \, [] \, J$, then p *unless* q in $\mathit{Inj}(N, J)$.

Proof Sketch. Every statement in $\mathit{Inj}(N, J)$ either (i) does not affect pure predicates (if it came from DS) or (ii) has the same effect on pure predicates as some statement in $N \, [] \, J$ from N or J, possibly with a stronger guard. The conclusion follows.

Theorem 5. Let N be receptive, and let J be an acceptable program with respect to N. Let p, q and r be *pure* predicates. If $p \mapsto q$ in $N \, [] \, J$, then $p \mapsto q \vee \langle \exists\, i :: \neg v.\mathit{SlotCondition}.i \rangle$ in $\mathit{Inj}(N, J)$.

Proof Sketch. The proof is by induction on the length of the proof for $N \, [] \, J$. If p *ensures* q, the *unless* part follows by the previous theorem. The existence of a proof for p *ensures* q in $N \, [] \, J$ implies there is a statement s in $N \, [] \, J$ such that

$$\{p \wedge \neg q\} \quad s \quad \{q\}.$$

The injection process implies in $\mathit{Inj}(N, J)$ we have a statement s' (corresponding to s in $N \, [] \, J$) satisfying

$$\{p \wedge \neg q \wedge \langle \exists\, i :: v.\mathit{SlotCondition}.i \rangle\} \quad s' \quad \{q\}$$

which is what is required for p *ensures* $q \vee \langle \exists\, i :: \neg v.\mathit{SlotCondition}.i \rangle$ in $\mathit{Inj}(N, J)$. The transitivity and disjunction cases follow by the Cancellation theorem and predicate calculus.

`Variable`	Type	Denotes
$v.state$, $v.Msg$	$nodeState$, message	Node State, Current Message
$v.RouteTable$	Map $endpoint \rightarrow$ chan. index	Route Table
$v.inC[x]$, $v.outC[x]$	Sequence of messages	Channel indexed by x to, from v
$v.NH$, $v.LH$	$endpoints$	next hop, last hop
$errorCnt$, $discardCnt$	integers	# msgs rec'd in error, discarded

Table 1: Variables in underlying programs $Node$ and DS running at node v

4 Example

In this section, we present an example underlying program that supports processing slots to which injected programs can be bound to form composite network services. We present an injected program that, when composed with the underlying program, enables locating mobile resources in a network.

4.1 An Example Underlying Program $Node$

In the example underlying program, each node has a number of associated processes. We define the type *endpoint* to be a two-tuple consisting of a node identifier and a process identifier. The functions $node(x)$, $pr(x)$ return the node and the process associated with endpoint x. Each node v has a finite set $v.outC$ of outgoing channels and a finite set $v.inC$ of incoming channels. Individual channels in these sets are identified by indices: $v.outC[i]$ denotes a particular output channel of v. Each output channel is connected to an input channel of some other node or a process at the local node by unspecified means. The function $end(x)$ returns the identifier of the peer at the other end of a channel x. For simplicity, we assume that this connection is reliable, i.e. any message in $v.outC[i]$ eventually shows up in $u.inC[j]$ for the appropriate u and j.

The set of processes at a node v is represented by $v.process$. We assume that two processes $ErrProc$ and $NullProc$ are always present at each node. The $ErrProc$ process deals with error conditions, and messages received in error (e.g. messages for non-existent processes at a node) are directed to it. The $NullProc$ is the source of all messages from a node that does not originate at any other process, and discards all messages sent to it.

Messages in the network have the following structure:

$$m = \{s, d, body\}$$

i.e. messages have a source, a destination, and a body. The source ($m.s$) and destination ($m.d$) of a message are identifiers are of type *endpoint*.

The state of the $Node$ program is encoded in the $v.state$ variable which is of type $nodeState$. The members of type $nodeState$ are: $nodeState = \{idle, slot.i.raise, slot.i.complete, newPkt, routePkt, routeFound\}$. Note that the $slot.i.raise$ and $slot.i.complete$ states are quantified over all slots i. For α of type $nodeState$, we define the predicates $v.\alpha \overset{\text{def}}{=} v.state = \alpha$. Thus, $v.idle$ is equivalent to $v.state = idle$, etc.

The $Node$ program listed below identifies two processing slots. Each slot is *raised* when certain (slot-specific) conditions become true. Injected code, in an appropriate form, can be *bound* to slots. Code, bound to a slot, is executed when the slot is raised. The raising of a particular slot x at a node is signaled by setting the $v.state$ variable to $slot.x.raise$. The $Node$ program resumes when the $v.state$ variable is set to $slot.x.complete$. The current message being processed at node v is identified by the $v.Msg$ variable. We define the predicate $at(m, v)$ to be "message m is the current message at node v", i.e. $at(m, v) \overset{\text{def}}{=} v.Msg = m$. At each node, messages are routed according to a routing table (represented by $v.RouteTable$). The routing table is a map between endpoints and integers, and an entry in the routing table of the form $v.RouteTable(d)$ is the identifier for the index of the channel from node v to endpoint d. Let $\delta(i, j)$ denote the distance (in hops) between nodes i and j in the network. We assume that $\delta(i, j) > 0$, if $i \neq j$, and 0 else.

We assume that routing tables have the following properties (v is quantified over nodes, and d is quantified over endpoints in the following):

$$\langle \forall\, v, d : v \neq node(d) : v.RouteTable(d) = n \;\Rightarrow\; \delta(end(v.outC[n]), node(d)) < \delta(v, node(d)) \rangle$$

$$\langle \forall\, v, d : v = node(d) \wedge \langle \exists\, k :: v.outC[k] = pr(d) \rangle : v.RouteTable(d) = k \rangle$$

$$\langle \forall\, v, d : v = node(d) \wedge \neg\langle \exists\, k :: v.outC[k] = pr(d) \rangle : v.RouteTable(d) = i \wedge v.outC[i] = ErrProc \rangle$$

Thus, if the endpoint passed to the routing table at node v identifies a node i different from v, then an index to an outgoing channel to node x is returned such that the distance (in hops) from node x to node i is strictly smaller than the distance to i from the current node; x is the *next hop* to i from v. In case the endpoint passed to the routing table identifies a process on the current node to which an outgoing channel exists, the identifier for such a channel is returned. A channel to the error process (*ErrProc*) is returned in case the endpoint identifies a process on the current node to which no outgoing channel exists.

```
Program {Program at each active node $v$} $Node$
 initially
 N0   $v.state, discardCnt, errorCnt = idle, 0, 0$                              { Initialization }
assign
 N1  $\langle\langle [\!]\, x : v.inC[x] \in v.inC : v.state, v.inC[x], v.Msg, v.LH := newPkt, \mathrm{tail}(v.inC[x]), \mathrm{head}(v.inC[x]), x\rangle$
       $\|$  $\langle \forall\, i :: v.rt.i.usage := 0\rangle\rangle$ if $v.idle \wedge (v.inC[x] \neq \bot)$
                          { If channel is non-empty, read message and initialize usage counters }
 N2  $[\!]$ $v.state := slot.0.raise$ if $v.newPkt$                     { Raise message arrival event }
 N3  $[\!]$ $v.state, v.NH := routeFound, v.RouteTable(v.Msg.d)$ if $v.slot.0.complete$
                                                        { Route message to proper channel }
 N4  $[\!]$ $v.state := slot.1.raise$ if $v.routeFound$                  { Raise routing done event }
 N5  $[\!]$ $\langle v.state,\ v.outC[v.NH] := idle,\ v.outC[v.NH]; v.Msg$
       $\|$ $\langle$  $discardCnt := discardCnt + 1$  if  $end(v.outC[v.NH]) = NullProc$
          $[\!]$ $errorCnt := errorCnt + 1$  if  $end(v.outC[v.NH]) = ErrProc\rangle\rangle$
          if $v.slot.1.complete$                   { Send message on proper channel; Update Counters }

 end {$Node$}
```

It can be shown that $Node \,[\!]\, DS$ is receptive, using the following partition of variables:

- C: $v.state$, $v.rt.i.usage$, $v.rt.i.bound$, $v.Progress.i$, $i = \{0, 1\}$.
- R: $v.RouteTable$, $v.NH$, $v.LH$, $errorCnt$, $discardCnt$
- W: $v.Msg$, $v.outC[x]$ for all output channels
- X: $v.inC[x]$ for all input channels

4.2 Properties of $Node \,[\!]\, DS$

We now state some properties of the underlying program. Let

$$D(m) \stackrel{\text{def}}{=} \delta(i, node(m.d)) \text{ if } \; at(m,i) \vee m \in i.outC[j].$$

Thus, $D(m)$ is the *distance* in hops a message m is from its destination node. For each node v in the network executing the example underlying program, the following properties hold:

Progress Properties

NP0 $\neg v.idle \mapsto v.idle$ { Message Processing is bounded }

NP1 $m \in i.inC[j] \mapsto at(m, j)$ { Channels drain }

NP2 $at(m, v) \wedge D(m) > 0 \mapsto j = v.RouteTable(m.d) \wedge m \in v.outC[j]$ { Messages are routed to the correct next hop }

Property **NP0** states that processing incurred due to any message at any node is finite. Using the channel properties, and the fact that message processing is finite, we derive property **NP1**, which states that all messages in an channel to a node is eventually processed by the node. Property **NP2** states that messages are forwarded on the *current* output channel, as specified by the routing table.

From these local node properties, we can derive the following property global property for the network.

GP0 $at(m, i) \mapsto at(m, node(m.d))$ { All messages are delivered }

Property **GP0** states that all messages are eventually delivered to their destination.

4.3 An Example Injected Program

In this section, we present an acceptable injected program for locating mobile resources in the network.

4.4 Mobility

Unlike static resources within a network, the location of a *mobile* resource (i.e. the node at which the resource is currently available) can change dynamically and asynchronously. A mobile resource may be available at a particular node at a given time, and then migrate to another node in the network. During the migration period, the resource is not available at any node in the network. In this section, we present an injected program that tracks and locates such mobile resources in an active network.

Assumptions We make the following assumptions about the environment and the migration of the mobile resources:

- Channels do not lose or corrupt messages, and buffers are not bounded.
- Each mobile resource has an unique identifier and an associated *home* node where it is initially located. The identity of the home node for each mobile resource is known at all nodes.
- Resources do not migrate forever; i.e. each migration period is finite, and followed by a period when the resource is available at some node in the network. Further, resources are not modified during migration.
- A resource is available at only one node at a given time. Thus two different nodes cannot possess the same mobile resource at the same time.
- The node environment is responsible for update to state variables in the mobility algorithm to indicate the (un)availability of each mobile resource. The updates to the state variables by the node environment satisfy a set of rules specified by the mobility algorithm.

4.5 The mobility algorithm

In our exposition, we assume that there is only one mobile resource. However, the algorithm and programs presented readily work with any finite number of mobile resources.

In order to locate the mobile resource in the network, we associate a timestamp with the resource. Each node maintains a corresponding logical clock and a last known location for the mobile resource. The resource's timestamp is increased each time the resource arrives at a particular node. An update message about the resource arrival with the new timestamp is sent to the last node from which the resource migrated (the identity of the the last node is carried with the resource). Accesses to the resource are sent towards the most current location known to the source of the access. A timestamp in the access determines the *currency* of the resource location carried in the access. En-route, if the access encounters a node with more current information (i.e. the node's logical clock is higher than the timestamp carried in the access), it is redirected towards the new location. We show that this scheme results in accesses continually making progress towards the current location of the resource, either by finding the resource or by finding a newer update.

Details Initially the resource is located at its home node, its timestamp is zero, and this information is available at all nodes. When a resource migrates from a node, all subsequent accesses to the resource that reach this node are queued till a message with newer information about the resource's location arrives. This new update must arrive as the resource must eventually become current at some node, causing an update to be sent. Upon receipt of the update, the queued accesses are forwarded towards the new location of the resource. In case the resource migrates right back to the node at which it was *last* located —i.e. the identity of the last location carried in the resource is the new node where the resource is now located— only a new timestamp is generated and all queued messages queued at this node are forwarded to processes at this node.

Optimality Note that we do not guarantee that accesses will always find the mobile resource. This is a consequence of the very general model of mobility we have assumed. Even in a two node network, we can create a scenario in which the location of the resource is defined to be the *other* node each time the access arrives at some node. When an update arrives, the resource will be forwarded towards the other node. In this manner, the access will always chase the resource in the network but never actually locate the resource. However, our algorithm is optimal in the sense the access is bound to encounter more and more current information about the location of the resource (i.e. ever increasing timestamps at each node). There are well-known heuristics that are often used to enhance the average case performance of mobility algorithms: e.g. always sending an update to the home node of a resource when a resource arrives at a node, etc. These techniques are not essential to prove the correctness of the mobility protocol, and as such, we have not included them in our algorithm.

4.6 Implementation

Messages in the mobility algorithm correspond to the seven-tuple $\{s, d, r, loc, ts, type, body\}$ where s and d are the source and destination of the message. The resource is identified by r; loc, ts correspond to resource location and an associated timestamp. The field *type* encodes the type of the message: it can be one of $Access$ (for accesses to mobile resource) or $Update$ (for updates on resource location). Finally, the *body* is the "payload" of the message.

The algorithm described is implemented by maintaining the following set of variables at each node:

- The state of the resource at each node is encoded in the $v.rState$ variable which is of type $state$ and can assume one of the three values $Cur, Mig,$ or Fwd. The states $v.rState = Cur$, $v.rState = Mig$ and $v.rState = Fwd$ at node v are denoted by $v.Cur, v.Mig,$ and $v.Fwd$ respectively. $v.Cur$ detects if the resource is resident at node v. $v.Mig$ detects the state when the resource has migrated from v but an updated location information has not been received at v yet. $v.Fwd$ is true if the resource is not resident at v, and the node is not in Mig state. During certain state transitions, each node has to discharge certain obligations in order for the algorithm to be correct. We use the variable $v.rStable$ to detect whether such obligations have been fulfilled during these state transitions.
- The variables $v.rLC, v.rLoc$ store the value of logical clock and last known location for the resource at node v. Variable $v.rQ$ is a buffer where accesses are queued while $v.Mig$ holds.
- The *home* node for each resource r is denoted by $r.home$. Resources are initially available at their homes, and the initial timestamp is zero. Further, for each resource r, the identity of the home node is known at all nodes.

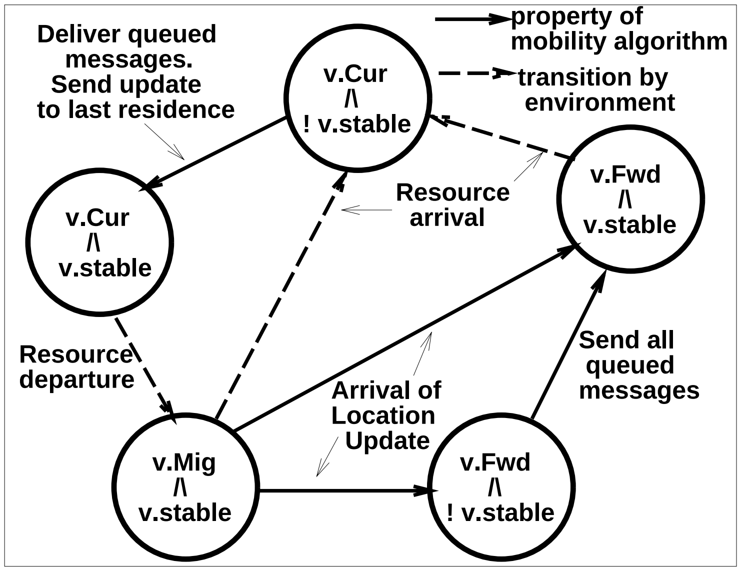

Figure 1: Possible transitions for node state

- If present (i.e. $v.Cur$ at node v), the resource can be accessed through the variable res. Specifically, $res.ts$ and $res.loc$ represent the timestamp and the location carried with the resource.

Resource Availability and Migration Figure 1 shows the valid transitions for the node state. Some transitions of the node state model the arrival and departure of the resource, and are triggered by the node environment. The predicate $v.Cur$ detects whether the resource is present at the node. The node environment must set the node state to Cur and the $v.rStable$ variable to $false$ to indicate the resource's arrival at v. Similarly, the environment must set the node state to Mig when the resource migrates. The resource may migrate only when $v.Cur \wedge v.stable$ holds. In general, the node environment may only update the value of $v.rState$ when $v.rStable$ is true. Thus, once the resource arrives at a node ($v.Cur \wedge \neg v.rStable$ holds), the resource can migrate only after the mobility algorithm has set the state to $v.Cur \wedge v.stable$.

We now formalize the rules for resource migration and the properties of the node environment.

Rules of resource migration

M0 $\langle \forall\, i, j : i \neq j : \neg i.Cur \vee \neg j.Cur \rangle$ { Resource resident at only one node }

M1 $v.Mig \wedge v.rLC = k \ \mathit{unless}\ \langle \exists\, i :: i.Cur \wedge \neg i.stable \wedge res.ts = k \wedge res.loc = v \rangle$
{ Resource does not change during migration }

M2 $v.Mig \wedge v.rLC = k \mapsto \langle \exists\, i :: i.Cur \wedge \neg i.stable \wedge res.ts = k \wedge res.loc = v \rangle$
{ Resource does not change during migration and migration terminates }

Properties of node environment

M3 $\neg v.Cur \wedge v.stable \ \mathit{unless}\ v.Cur \wedge \neg v.stable$ { Resource arrival }

M4 $v.Cur \wedge v.stable \ \mathit{unless}\ v.Mig \wedge v.stable$ { Resource departure }

M5 $\langle \forall\, s : \{Cur, Mig, Fwd\} : v.s \wedge \neg v.stable \ \mathit{unless}\ v.s \wedge \neg v.stable \rangle$
{ Environment may not change state unless $v.stable$ holds }

Variable	Type	Denotes
v.rState, v.rStable	*state*, boolean	Node state, whether state is stable
v.rLoc, v.rLC	node id, integer	Last known location, Logical clock at node
v.rQ	queue of type message	Buffer to queue accesses during migration

Table 2: Variables in the implementation of the mobility algorithm

4.6.1 UNITY **Program Specification**

For the sake of brevity, we define the following abbreviations. At each node v:

$$fwd\ (s,d,r,rd,ts,t,b) \ \overset{\text{def}}{=} \ \langle v.outC[v.RouteTable(d)] := v.outC[v.RouteTable(d)]; \{s,d,r,rd,ts,t,b\}\rangle$$

and

$$Qh \overset{\text{def}}{=} \text{head}(v.rQ)$$

Thus, $fwd\ (v,d,r,\alpha,k,Access,body)$ corresponds to sending a message towards the destination d from the current node (v), and the resource identifier, resource location, resource timestamp, message type, and message body in the message equal to r, α, k, $Access$, and $body$ respectively. Similarly Qh refers to the message (if any) at the head of the queue of messages in the queue $v.rQ$ at node v. Further, given endpoint d, we define $redir(x,d)$ to be the new endpoint $\{x, pr(d)\}$; thus, $redir(x,d)$ is an endpoint with the same process identifier as d but re-directed to node x. Finally, we define $nullPr(x)$ to be the endpoint $\{x, NullProc\}$.

```
Program {Mobility algorithm} $Mobility$
 initially
 MA0  $v.rState, v.rLC, v.rLoc, v.rStable, v.rQ = Fwd, 0, r.home, \text{true}, \perp$
        if $v \neq r.home \sim Cur, 0, v, \text{true}, \perp$ if $v = r.home$
                  { Resource $r$ is initially located at $r.home$; this is known to all other nodes }
 assign
 MA1  $v.Msg.d, v.Msg.loc, v.Msg.ts := redir(v.rLoc, v.Msg.d), v.rLoc, v.rLC$
        if $v.rLC > v.Msg.ts \wedge v.Msg.type = \text{Access} \wedge (v.Fwd \vee v.Cur) \wedge v.stable$
                  { Re-direct accesses containing stale information }
 MA2  $v.rLoc, v.rLC := v.Msg.loc, v.Msg.ts$ if $v.rLC < v.Msg.ts \wedge v.Fwd \wedge v.stable$
                  { Update local clock and forwarding information if message contains newer information }

 MA3  $\langle fwd\ (Qh.s, redir(v, Qh.d), Qh.r, v, res.ts+1, Qh.type, Qh.body)$
    || $v.rQ := \text{tail}(v.rQ)\rangle$ if $v.Cur \wedge \neg v.stable \wedge v.rQ \neq \perp$
                  { Resource arrives at node $v$; Deliver all queued messages }
 MA4  $\langle v.rLoc, v.rLC, res.ts, res.loc, v.rStable := v, res.ts+1, res.ts+1, v, \text{true}$
    || $fwd\ (nullPr(v), nullPr(res.loc), v, r, res.ts+1, Update, \perp)\ \rangle$
         if $v.Cur \wedge \neg v.stable \wedge res.loc \neq v \wedge v.rQ = \perp$
                  { Resource arrived from other node; Increment clock, send message to last known location }
 MA5  $v.rLoc, v.rLC, res.ts, res.loc, v.rStable := v, res.ts+1, res.ts+1, v, \text{true}$
        if $v.Cur \wedge \neg v.stable \wedge res.loc = v \wedge v.rQ = \perp$
                  { Resource migrated back to node $v$ from node $v$; Increment clock }

 MA6  $v.rQ, v.Msg.d := v.rQ; v.Msg, nullPr(v)$
          if $v.Mig \wedge v.Msg.ts \leq v.rLC \wedge v.Msg.type = Access$
                  { Access to migrating resource; Queue access, and redirect current message to $NullProc$ }

 MA7  $v.rState, v.rLoc, v.rLC, v.rStable := Fwd, v.Msg.loc, v.Msg.ts, (v.rQ = \perp)$
```

```
        if $v.Mig \wedge v.Msg.ts > v.rLC$                    { New update; $v.rStable$ detects empty $v.rQ$ }

  MA8  $\langle v.rQ, v.rStable := \text{tail}(v.rQ), (\text{tail}(v.rQ) = \perp)$
       $\|$ $fwd\ (Qh.s, redir(v.rLoc, Qh.d), Qh.r, v.rLoc, v.rLC, Qh.type, Qh.body)\ \rangle$
        if $\neg v.stable \wedge v.Fwd$          { forward queued messages to new location till queue is empty }

end {$Mobility$}
```

4.7 Composing *Mobility* with *Node*

The *Mobility* program is designed to be bound to slot 0 in the *Node* program. After processing by the *Mobility* program, the current message will be forwarded by the *Node* program. If the current message is at its destination, it will be forwarded to a process; otherwise, it will be forwarded a node corresponding to the next hop towards its destination. Specifically, *Update* messages are discarded at their destination as they are directed to the *NullProc* process. When the resource is available at the node, all *Access* messages are forwarded on channels to their respective processes. The guarantee about resource availability provided by the mobility algorithm is as follows: *Access* messages are forwarded to a process at a node v if and only if $v.Cur$ holds. This condition does not imply that $v.Cur$ must hold when the *Access* message is eventually read or processed by the terminating process at node v. Thus, a race condition may ensue if the resource migrates before the forwarded messages reach their individual processes. We do not model restrictions on resource migration that may be imposed by higher layer processes beyond the stability criteria specified by the mobility algorithm.

For composition with the *Mobility* program, the *Node* program specifies the following safety property for writing into the $v.Msg$ variable:

$$\textbf{stable}\, v.Msg.type \neq Access \wedge v.Msg = m \text{ in } Mobility.$$

Using this safety property, property **GP0**, and Theorem 3, we can derive:

GP1 $at(m,i) \wedge m.type \neq Access \ \mapsto\ at(m, node(m.d))$

All non-*Access* messages are delivered to their destination by $Inj(Node, Mobility)$

Property **GP1** states that all messages with message type different from *Access* eventually reach their destination node. Next we derive properties of messages of type *Access* using properties of *Mobility* in $Inj(Node, Mobility)$.

4.8 Properties of $Inj(Node, Mobility)$

We derive the following properties for each node in which $Inj(Node, Mobility)$ is executed:

Safety of clocks and timestamps

MP0 $res.ts = k \text{ unless } res.ts > k$ { Resource's timestamp monotonically increases }

MP1 $v.rLC = k \text{ unless } v.rLC > k$ { Logical Clocks monotonically increase }

Safety and Progress for unstable *Fwd* state

MP2 $v.Fwd \wedge \neg v.stable \text{ unless } v.Fwd \wedge v.stable$ { Unstable *Fwd* states must stabilize }

MP3 $v.Fwd \wedge \neg v.stable \mapsto v.Fwd \wedge v.stable$
{ Unstable *Fwd* states always terminate to stable *Fwd* states }

Safety and Progress for unstable *Cur* state

MP4 $v.Cur \wedge \neg v.stable \ unless \ v.Cur \wedge v.stable$ { Unstable Cur states must stabilize }

MP5 $v.Cur \wedge \neg v.stable \mapsto v.Cur \wedge v.stable$
{ Unstable Cur states always stabilize to stable Cur states }

MP6 $v.Cur \wedge \neg v.stable \wedge v.rLC = k \ unless \ v.Cur \wedge v.stable \wedge v.rLC > k$
{ Unstable Cur state must increase clock and stablize }

MP7 $v.Cur \wedge \neg v.stable \wedge v.rLC = k \mapsto v.Cur \wedge v.stable \wedge v.rLC > k$
{ Unstable Cur state always increases clock and stablizes }

Safety and Progress for Mig state

MP8 $v.Mig \wedge v.rLC = k \ unless \ (v.Fwd \wedge v.rLC > k) \vee (v.Cur \wedge \neg v.stable \wedge v.rLC = k)$
{ Mig state must be followed by Fwd or unstable Cur state }

MP9 $v.Mig \wedge v.rLC = k \mapsto \neg v.Mig \wedge v.rLC > k$
{ Mig state ends with an increment of the node's logical clock }

The properties **MP0**, **MP1** specify that the resource timestamp and logical clocks at each node can only increase. Properties **MP2**, **MP3**, **MP4**, **MP5** specify that periods during which a node is unstable (i.e. $\neg v.stable$ holds) is finite. Further, **MP6** says that each time the resource arrives at a node, the node's logical clock is increased. Property **MP8** specifies that a node in migration state must stay in migration state until an update with higher timestamp or the resource arrives at the node. Property **MP9** states that a migration phase at a node always comes to an end, i.e. the node always receives a message with a newer timestamp, or the resource migrates back to the same node. Thus, periods during which a node is in migration state is always finite. Further, **MP9** states that after the migration period, the logical clock at the node must increase. This must be so because either the update has a greater timestamp, or the clock at the node is incremented if the resource arrives at the node. In general, the safety properties prohibit transitions that are not present in the state transition diagram (Figure 1). The progress properties correspond to the arcs in the transition diagram — they represent all the legal state transitions.

A Global Property of $Node \,[\!]\, Mobility$ Define predicate $atq(m, v)$ as follows

$$atq(m, v) \stackrel{\text{def}}{=} at(m, v) \vee m \in v.rQ$$

Given that $Node$ program satisfies the conditions U0–U4, and the $Mobility$ program satisfies J0 and identifies slot 0 in $Node$ to be bound to, $Node \,[\!]\, Mobility$ program has the following global property:[1].

GP1 $at(m, v) \wedge m.type = Access \mapsto at(m, m.d) \vee \langle \exists\, i :: at(m, i) \wedge i.Cur \rangle$
$\vee \langle \exists\, j :: atq(m, j) \wedge j.rLC > m.ts \rangle$
{ Access Messages always find the resource, a new update, or are delivered to their destination }

Using Theorem 5, and **GP1**, we derive the following property for $Inj(Node, Mobility)$:

$at(m, v) \wedge m.type = Access \mapsto at(m, m.d) \vee \langle \exists\, i :: at(m, i) \wedge i.Cur \rangle$
$\vee \langle \exists\, j :: atq(m, j) \wedge j.rLC > m.ts \rangle \ \vee \langle \exists\, v :: \neg v.SlotCondition.0 \rangle$
{ Unless resource bounds are violated, the $Mobility$ properties are preserved }

5 Conclusion

We have described an abstract form of programming interface for active networks. The interface defines the interaction between a fixed, underlying node program and code which can be injected into the network.

[1]This proof of this property and some other properties of $Node \,[\!]\, Mobility$ are given in Appendix B

The interface constrains the functionality of the injected program to certain points in the execution of the underlying program. Using a formal model, we have shown how to reason about the correctness of the network + injected program starting from properties proved of each in isolation. We have illustrated the use of the techniques on a nontrivial example in UNITY dealing with mobility.

The UNITY notation is simple and easy to learn. This simplicity comes at a cost. There is no type theory, for example, nor any form of variable scoping—these must be handled outside the formalism. Any sequencing of statements must be handled explicitly by the programmer. On the other hand, UNITY programs naturally promote maximum parallelism, and treat global properties essentially the same as local ones. The UNITY syntax provides a powerful, yet simple means of encoding programs.

Our approach is designed to facilitate proofs that an active network's global behavior maintains certain correctness properties provided the injected programs satisfy certain restrictions. In this paper, some of these restrictions were syntactic (J0 and J1), but some were not (e.g. the hypotheses in several of the Theorems). The problem of checking injected programs for suitability remains an interesting one.

Appendix A: Overview of UNITY

The UNITY formalism was introduced by Chandy and Misra in 1987 [1], and has proven useful for reasoning rigorously about the correctness of distributed algorithms in a variety of contexts [4].

The Unity formalism consists of a program notation capable of representing any distributed algorithm, a specification notation for representing simple temporal properties (i.e. properties that express characteristics of a program's behavior as it evolves over time), and a proof system, which allows one to prove assertions of the form "P holds in F", where P is an expression in the program notation, and F is expressed in the specification notation. Such an assertion means that all possible computations of F have the characteristic represented by the the temporal predicate P.

A program in UNITY is represented as a finite set of concurrent assignment statements of the form:

$$\begin{array}{ccccc} \bar{x} & := & \bar{e_0} & \text{if} & b_0 \\ & \sim & \bar{e_1} & \text{if} & b_1 \\ & & \ldots & & \\ & \sim & \bar{e_k} & \text{if} & b_k \end{array}$$

where $\bar{x}$ is a list of variables and each $\bar{e}_i$ is a list of expressions of the same length. the semantics are that all the predicates b_i are evaluated; if one is true the corresponding expressions on the right-hand side are evaluated and the results are simultaneously assigned to the corresponding variables on the left-hand side. [The b_i are required to be mutually exclusive.] If no b_i evaluates to true, the values of the variables on the left-hand side of the statement remain unchanged, and the statement is said to be *ineffective*.

Large concurrent assignment statements can also be represented using the $||$ operator: the statement

$$\langle || \, i : 0 \leq i < 4 : x_i := y_i \text{ if } b \rangle$$

is equivalent to

$$x_0, x_1, x_2, x_3 := y_0, y_1, y_2, y_3 \text{ if } b$$

The different statements of a program are composed with the $[\!]$ operator

$$x := y \,\text{if}\, b \quad [\!] \quad u := v \,\text{if}\, c$$

which can also be used as a quantifier:

$$\langle [\!] \, i : i \in X : x_i, y_i := x_{f(x)}, y_{g(x)} \rangle$$

The above represents a program with as many statements as there are elements of X.

The execution model of a UNITY program has no explicit notion of control flow. The execution consists of an infinite number of steps, where at each step a statement is selected and executed. If the statement is ineffective, the step does not change the overall state of the program (i.e. the values of all the variables). The model requires that each statement be selected infinitely often, but offers no other guarantee on the scheduling, i.e. there is no bound on the number of steps between executions of any particular statement. This model obviously highlights the asynchrony between different parts of a distributed or concurrent program.

The *correctness properties* of a UNITY program are partitioned into assertions about safety and progress properties of a program. According to Lamport, safety properties ensure "bad things do not happen". Safety properties constrain the permissible states the program can be in by preventing a state transition into an invalid state. Informally, progress properties state that "good things do happen".

Safety Properties The fundamental safety property of UNITY programs is the **unless** property. It is a binary relation over program state predicates and is defined for a program P as follows:

$$p \textit{ unless } q \equiv \langle \forall \, s : s \text{ in } P : \{p \wedge \neg q\} \quad s \quad \{p \vee q\} \rangle$$

This means that if p is true at some point and q is not, then in all subsequent steps p remains true or q becomes true. Thus, a program in state where p can only falsify p by transitioning through a state where q holds. Consider the following example of an *unless* property:

$$x = k \ \textit{unless}\ x > k$$

This property states that in all executions of the program, the value of the variable x never decreases, i.e. x is monotonically increasing. Two important special cases of *unless* are **stable** and **invariant**, defined as follows:

$$\textbf{stable}\, p \equiv p \ \textit{unless false} \quad \text{and} \quad \textbf{invariant}\, p \equiv (\text{initial condition} \Rightarrow p) \land \textbf{stable}\, p$$

Thus, a stable predicate remains true forever once it becomes true. An invariant is always true—all states in the program always satisfy all invariants during any execution. Consider the example properties:

$$\textbf{stable}\, x > 0 \text{ and } \textbf{invariant}\, x > 0$$

The first property states that once the value of variable x becomes positive, it will always remain positive. If x were initially positive, then we can assert the second property — x is always positive in all states for all executions of the program.

Progress Properties The most basic progress of UNITY programs are specified in terms of the **ensures** property, which for a given program P is defined as follows:

$$p \ \textit{ensures}\ q \equiv (p \ \textit{unless}\ q \land \langle \exists\, s : s \text{ in } P : \{p \land \neg q\} \quad s \quad \{q\}\rangle)$$

This if p is true in some state, then p remains true as long as q does not hold. Further, there is a statement s that explicitly establishes q. Thus, once p becomes true, q must become true. The following is an example of an *ensures* property:

$$x = k \ \textit{ensures}\ x > k$$

which states that the value of x is non-decreasing and that x will increase. Thus, not only are there no statements that decrease the value of x, but there is at least one statement that increases the value of x. Almost all progress properties of UNITY programs are in terms of the **leads-to**($\mapsto$) operator. It is defined as the transitive, disjunctive closure of the *ensures* relation. A program has the property $p \mapsto q$ if and only if this property can be derived by a finite number of applications of the following inference rules:

$$\frac{p \ \textit{ensures}\ q}{p \mapsto q}, \quad \frac{p \mapsto r, r \mapsto q}{p \mapsto q}, \qquad \text{for any set W} : \ \frac{\langle \forall\, m : m \in W : p(m) \mapsto q\rangle}{\langle \exists\, m : m \in W : p(m)\rangle \mapsto q}.$$

Thus, once p becomes true, q is or will become true. For example, the property

$$x = k \mapsto x > k$$

states that once a program once the value of x is equal to k, at a subsequent state, the value of x is guaranteed to be greater than k.

Appendix B: Proofs of Properties of $\textit{Node} \,[\!]\, \textit{Mobility}$

We show **GP1**:
$at(m, v) \land m.type = Access \mapsto at(m, m.d) \lor \langle \exists\, i :: at(m, i) \land i.Cur\rangle$
$\lor \langle \exists\, j :: atq(m, j) \land j.rLC > m.ts\rangle$

Proof:
Let $p \equiv at(m, v) \land m.type = Access \land m.ts = k$.
Let $q \equiv at(m, m.d) \lor \langle \exists\, i :: at(m, i) \land i.Cur\rangle \lor \langle \exists\, j :: atq(m, j) \land j.rLC > k\rangle$.
Let $a \equiv at(m, m.d)$, $b \equiv \langle \exists\, i :: at(m, i) \land i.Cur\rangle$, and $c \equiv \langle \exists\, j :: atq(m, j) \land j.rLC > k\rangle$.

Thus, $q \equiv a \vee b \vee c$.
Note that:

$$p \equiv p \wedge (v.Cur \vee v.Mig \vee v.Fwd) \wedge (v.rLC > k) \vee (v.rLC = k) \vee (v.rLC < k)$$

We have to show that each of the nine cases
$p \wedge v.Cur \wedge v.rLC > k, p \wedge v.Mig \wedge v.rLC > k, p \wedge v.Fwd \wedge v.rLC > k$ etc. all lead-to q.

0. $p \wedge v.Cur \wedge v.rLC \geq k \Rightarrow b$ { $v.Cur$ holds }
1. $p \wedge v.Cur \wedge v.rLC < k \Rightarrow b$ { $v.Cur$ holds }
2. $p \wedge v.Cur \Rightarrow b$ { **0**,**1**, Thus, all Cur states lead-to q }

3. $p \wedge v.Mig \wedge (v.rLC > k) \Rightarrow c$ { $v.rLC > k$ }
4. $p \wedge v.Fwd \wedge (v.rLC > k) \Rightarrow c$ { $v.rLC > k$ }

Consider $p \wedge v.Mig \wedge v.rLC = k$:
5. $p \wedge v.Mig \wedge v.rLC = k$ *unless* $atq(m,v) \wedge v.Mig \wedge v.rLC = k$ { Program Text }
6. $p \wedge v.Mig \wedge v.rLC = k$ *ensures* $atq(m,v) \wedge v.Mig \wedge v.rLC = k$ { **MA6** }
Note that:
7. $atq(m,v) \wedge v.Mig \wedge v.rLC = k$ *unless* $atq(m,v) \wedge \neg v.Mig$ { Program Text }
8. $v.Mig \wedge v.rLC = k$ *unless* $(v.Fwd \wedge v.rLC > k) \vee (v.Cur \wedge v.rLC = k)$ { Program Text }
9. $atq(m,v) \wedge v.Mig \wedge v.rLC = k$ *unless* $(atq(m,v) \wedge v.Fwd \wedge v.rLC > k)$
$\vee(atq(m,v) \wedge v.Cur \wedge \neg v.stable \wedge v.rLC = k)$ { **7**, **8**, Conjunction }
10. $v.Mig \wedge v.rLC = k \mapsto (v.Cur \wedge \neg v.stable \wedge v.rLC = k) \vee (v.Fwd \wedge v.rLC > k)$ { Proved below }
Thus, from **9**, and **10**
11. $atq(m,v) \wedge v.Mig \wedge v.rLC = k \mapsto (atq(m,v) \wedge v.Fwd \wedge v.rLC > k)$
$\vee(atq(m,v) \wedge v.Cur \wedge \neg v.stable \wedge v.rLC = k)$ { **9**, **10**, PSP }
In each of the three ensuing cases:
12. $atq(m,v) \wedge v.Fwd \wedge v.rLC > k \Rightarrow c$ { $v.rLC > k$ }
13. $atq(m,v) \wedge v.Cur \wedge \neg v.stable \wedge v.rLC = k \Rightarrow b$ { $v.Cur$ holds }
14. $atq(m,v) \wedge v.Mig \wedge v.rLC = k \mapsto c \vee b$ { **11**, **12**, **13**, Implication }
Therefore:
15. $p \wedge v.Mig \wedge v.rLC = k \mapsto c \vee b$ { **6**, **15**, Transitivity }

Consider $p \wedge v.Mig \wedge v.rLC < k$:
16. $p \wedge v.Mig \wedge v.rLC < k$ *unless* $p \wedge v.Fwd \wedge v.rLC = k$ { Program Text }
17. $p \wedge v.Mig \wedge v.rLC < k$ *ensures* $p \wedge v.Fwd \wedge v.rLC = k$ { **MA7** }
We will show that $p \wedge v.Fwd \wedge v.rLC = k \mapsto q$ next.
Consider $p \wedge v.Fwd \wedge v.rLC \leq k$:
18. $p \wedge v.Fwd \wedge v.rLC \leq k \equiv p \wedge v.Fwd \wedge v.rLC \leq k \wedge D(m) \geq 0$
{ $D(m) \overset{\text{def}}{=} \delta(i, m.d)$ if $at(m,i) \vee m \in i.outC[j]$ }
There are two cases, $D(m) = 0$, and $D(m) > 0$
19. $p \wedge v.Fwd \wedge v.rLC \leq k \wedge D(m) = 0 \Rightarrow at(m, m.d)$ { Case I: $D(m) = 0 \Rightarrow at(m, m.d)$ }
20. $p \wedge v.Fwd \wedge v.rLC \leq k \wedge D(m) = l \wedge\ l > 0$
$\mapsto at(m,i) \wedge (i.Fwd \vee i.Mig \vee i.Cur) \wedge D(m) < l$
{ Case II: $D(m) > 0$, Property of $Node \,[\!]\, Mobility$ and $v.RouteTable$ }

Node i can now be in one of three states Cur, Fwd, Mig each of which lead-to q:
21. $at(m,i) \wedge i.Cur \wedge D(m) < l \Rightarrow b$ { **2** }

Consider the Mig states:
22. $at(m,i) \wedge i.Mig \wedge D(m) < l \equiv at(m,i) \wedge i.Mig \wedge D(m) < l$

$\wedge(i.rLC > m.ts \vee i.rLC = m.ts \vee i.rLC < m.ts)$ { Excluded Middle }

23. $at(m,i) \wedge i.Mig \wedge D(m) < l \wedge i.rLC = m.ts \mapsto c \vee b$ { **15** }
24. $at(m,i) \wedge i.Mig \wedge D(m) < l \wedge i.rLC > m.ts \mapsto c$ { **3** }
25. $at(m,i) \wedge i.Mig \wedge D(m) < l \wedge i.rLC < m.ts \mapsto at(m,i) \wedge i.Fwd \wedge D(m) < l \wedge i.rLC = m.ts$ { **17** }

And now the induction on the Fwd states:
26. $at(m,i) \wedge i.Fwd \wedge D(m) < l \wedge i.rLC > m.ts \Rightarrow c$ { **4** }
27. $p \wedge v.Fwd \wedge v.rLC \leq k \wedge D(m) = l \mapsto (p \wedge v.Fwd \wedge v.rLC \leq k \wedge D(m) < l) \vee (a \vee b \vee c)$
{ **19**, **21**, **23**, **24**, **25**, **26** }
28. $p \wedge v.Fwd \wedge v.rLC \leq k \mapsto (a \vee b \vee c)$ { **27**, Induction }
29. $p \wedge v.Mig \wedge v.rLC < k \mapsto q$ { **29**, **28**, Transitivity }
Thus:
30. $p \mapsto q$ { **3**, **4**, **2**, **15**,**28**,**29** }

We now prove **MP9**:
$v.rLC = k \wedge v.Mig \mapsto v.rLC > k \wedge \neg v.Mig$
Proof:
0. $v.Mig \wedge v.rLC = k \mapsto \langle \exists\, i :: i.Cur \wedge \neg i.stable \wedge res.ts = k \wedge res.loc = v \rangle$ { **M2** }

Let $len(q)$ be the number of messages in queue q,
and consider case when $i = v$. Let $p \equiv v.Cur \wedge \neg v.rStable \wedge res.ts = k \wedge res.loc = v$.
In this case, we have to show that resource arrival always increments the node's logical clock.
Before the clock increments, all the queued messages have to be forwarded.

1. $v.Mig \wedge v.rLC = k \mapsto p \vee \langle \exists\, i : i \neq v : i.Cur \wedge \neg i.stable \wedge res.ts = k \wedge res.loc = v \rangle$ { $i = v \vee i \neq v$ }
2. p *unless* $v.Cur \wedge v.rStable \wedge v.rLC > k$ { Program Text }
3. $p \equiv p \wedge (v.rQ = \perp \vee v.rQ \neq \perp)$ { Excluded Middle }
4. $p \wedge (v.rQ = \perp)$ *unless* $v.Cur \wedge v.rStable \wedge v.rLC > k$ { Program Text }

If $v.rQ$ is empty, then just increment the clock ...
5. $p \wedge (v.rQ = \perp)$ *ensures* $v.Cur \wedge v.rStable \wedge v.rLC > k$ { **MA5** }

...else we have to empty the queue first.
6. $p \wedge v.rQ \neq \perp \wedge len(v.rQ) = l$ *unless* $(p \wedge v.rQ \neq \perp \wedge len(v.rQ) < l) \vee (p \wedge v.rQ = \perp)$
{ Program Text }
7. $p \wedge v.rQ \neq \perp \wedge len(v.rQ) = l$ *ensures* $(p \wedge v.rQ \neq \perp \wedge len(v.rQ) < l) \vee (p \wedge v.rQ = \perp)$
{ **MA3** }

$v.rQ$ always empties, and we are back to the first case:
8. $p \wedge v.rQ \neq \perp \mapsto (p \wedge v.rQ = \perp)$ { **7**, Induction }
9. $p \wedge v.rQ \neq \perp \mapsto v.Cur \wedge v.rStable \wedge v.rLC > k$ { **8**, **5**, Transitivity }
10. $v.Mig \wedge v.rLC = k \mapsto$
$\neg v.Mig \wedge v.rLC > k \vee \langle \exists\, i : i \neq v : i.Cur \wedge \neg i.stable \wedge res.ts = k \wedge res.loc = v \rangle$
{ $v.Cur \wedge v.stable \Rightarrow \neg v.Mig$, **1**, **9**, Cancellation }

Now consider the case when $i \neq v$, and let $r \equiv i.Cur \wedge \neg i.stable \wedge res.ts = k \wedge res.loc = v$, and let
$q \equiv \langle \exists m' : node(m'.s) = i \wedge node(m'.d) = v \wedge$
$m'.ts > k \wedge m'.loc = i \wedge m'.type = Update : m' \in i.outC[i.RouteTable(m'.d)] \rangle$:
We again have to show that $i.rQ$ empties which ensures q
: **11.** $r \equiv r \wedge (i.rQ = \perp \vee i.rQ \neq \perp)$ { Excluded Middle }

12. $r \wedge (i.rQ = \perp)$ *unless* q { Program Text }
13. $r \wedge (i.rQ = \perp)$ *ensures* q { **MA4** }
14. $r \wedge i.rQ \neq \perp \wedge len(i.rQ) = l$ *unless* $(r \wedge i.rQ \neq \perp \wedge len(i.rQ) < l) \vee (r \wedge i.rQ = \perp)$
{ Program Text }
15. $r \wedge i.rQ \neq \perp \wedge len(v.rQ) = l$ *ensures* $(r \wedge i.rQ \neq \perp \wedge len(i.rQ) < l) \vee (r \wedge i.rQ = \perp)$
{ **MA3** }
16. $r \wedge i.rQ \neq \perp \mapsto (r \wedge v.rQ = \perp)$ { **15**, Induction }
17. $r \wedge i.rQ \neq \perp \mapsto q$ { **16**, **13**, Transitivity }
18. $r \mapsto q$ { **13**, **17**, Completion }
19. $q \mapsto at(m', v) \wedge m'.ts > k$ { Property of *Node* }
Update reaches v:
20. $r \mapsto at(m', v) \wedge m'.ts > k$ { **18**, **19**, Transitivity }
21. $v.rLC = k \wedge v.Mig \mapsto (at(m', v) \wedge m'.ts > k) \vee \neg v.Mig \wedge v.rLC > k$ { **20**,**10**, Cancellation }
22. $v.rLC = k \wedge v.Mig \mapsto (at(m', v) \wedge m'.ts > k \wedge v.rLC = k \wedge v.Mig)$
$\vee (v.Fwd \wedge v.rLC > k) \vee (v.Cur \wedge \neg v.rStable \wedge v.rLC = k)$ { **21**, **MP8**, PSP }
23. $v.Fwd \wedge v.rLC > k \Rightarrow \neg v.Mig \wedge v.rLC > k$ { $v.Fwd \Rightarrow \neg v.Mig$ }
24. $v.rLC = k \wedge v.Mig \mapsto (at(m', v) \wedge m'.ts > k \wedge v.rLC = k \wedge v.Mig) \vee (\neg v.Mig \wedge v.rLC > k)$
{ **22**,**23**, **MP7**, Cancellation }
25. $at(m', v) \wedge m'.ts > k \wedge v.rLC = k \wedge v.Mig$ *unless* $\neg v.Mig \wedge v.rLC > k$ { Program Text }
26. $at(m', v) \wedge m'.ts > k \wedge v.rLC = k \wedge v.Mig$ *ensures* $\neg v.Mig \wedge v.rLC > k$ { **MA7** }
27. $v.Mig \wedge v.rLC = k \mapsto \neg v.Mig \wedge v.rLC > k$ { **24**, **26**, Cancellation }

We also establish:
$v.Mig \wedge v.rLC = k \mapsto (v.Cur \wedge \neg v.stable \wedge v.rLC = k) \vee (v.Fwd \wedge v.rLC > k)$
which is needed for the proof of **GP1**.
Proof:
28. $v.rLC = k \wedge v.Mig \mapsto (at(m', v) \wedge m'.ts > k \wedge v.rLC = k \wedge v.Mig)$
$\vee (v.Fwd \wedge v.rLC > k) \vee (v.Cur \wedge \neg v.rStable \wedge v.rLC = k)$ { **22** }
29. $at(m', v) \wedge m'.ts > k \wedge v.rLC = k \wedge v.Mig$ *unless* $v.Fwd \wedge v.rLC > k$ { Program Text }
30. $at(m', v) \wedge m'.ts > k \wedge v.rLC = k \wedge v.Mig$ *ensures* $v.Fwd \wedge v.rLC > k$ { **MA7** }
31. $v.Mig \wedge v.rLC = k \mapsto (v.Cur \wedge \neg v.stable \wedge v.rLC = k) \vee (v.Fwd \wedge v.rLC > k)$
{ **28**, **30**, Cancellation }

References

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

[2] J. Gosling and H. McGilton. The Java language environment: A White paper. Sun Microsystems, 1995.

[3] Michael Hicks, Pankaj Kakkar, Jonathan Smith, T. Moore, Carl A. Gunter, and Scott Nettles. PLAN: A Programming Language for Active Networks. Submitted to ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1998.

[4] J. R. Rao. Reasoning about probabilistic parallel programs. *ACM Transactions on Programming Languages and Systems*, 16(3):798–842, May 1994.

[5] Jonathan Smith, David Farber, Carl A. Gunter, Scott Nettles, Mark Segal, William D. Sincoskie, David Feldmeier, , and Scott Alexander. SwitchWare: Towards a 21st century network infrastructure. Whitepaper.

[6] D. Wetherall, J. Guttag, and D. L. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols. In *IEEE OPENARCH'98*, San Francisco, CA, April 1998.