# A Nitpick Analysis of Mobile IPv6


| Daniel Jackson | Yuchung Ng | Jeannette Wing |
|---|---|---|
| MIT Lab. for Comp.Sci. | Computer Science Dept. | Computer Science Dept. |
| 545 Technology Square | Cornell University | Carnegie Mellon University |
| Cambridge, MA 02139 | Ithaca, NY 14853 | Pittsburgh, PA 15213 |



22 September 1997

**Keywords:** partial specifications, model checking, Nitpick/NP, mobile internetworking protocols



**Abstract**

A lightweight formal method [JW96] enables partial specification and automatic analysis by sacrificing breadth of coverage and expressive power. By design, NP is a specification language that is a subset of Z and Nitpick is a tool that quickly and automatically checks properties of finite models of systems specified in NP. We used NP to state two critical acyclicity properties of Mobile IPv6, a new internetworking protocol that allows mobile hosts to communicate with each other. In our Nitpick analysis of Mobile IPv6 we discovered a flaw in a 1996 version of the design: one of the acyclicity properties does not hold. It takes only two hosts to exhibit this flaw. This paper gives self-contained overviews of Mobile IPv6 and of NP and Nitpick to understand the details of our specification and analysis.


## 1. Introduction

A mobile internetworking protocol (IP) is responsible for routing messages sent from one host to another where hosts may move around to different points in the network. One of the key desired properties of any mobile IP is that messages never travel indefinitely in a cycle. As a message hops from one point to the next in its route, network and site resources are consumed; if there is a cycle in the route then the message could travel forever and clog the network.

This paper discusses a formal specification of two desired acyclicity properties of the Mobile Internetworking Protocol version 6 (IPv6) as described in the June 1996 draft standard written by the Mobile IP Working Group of the Internet Engineering Task Force (IETF). In our analysis of Mobile IPv6 we discovered that it does not satisfy one of the acyclicity properties. Ironically an earlier version, IPv4 [J95], does. We brought this problem to the attention of one of the IPv6 designers (Johnson) who agreed that indeed the standard's ambiguous and imprecise language admits this design flaw; a subsequent wording change to the IPv6 documentation (1997 version) suggests a fix to the problem found.

We wrote our formal specification in NP, the input language of Nitpick, a tool for checking properties of finite binary relations. By design, NP is a subset of Z [S92]; any Z specification expressed in terms of NP is amenable to Nitpick analysis. Nitpick is one of the few semantic analyzers for Z. Like its successful model checker counterparts, which greatly inspired Nitpick's design, Nitpick generates counterexamples: when a property does not hold of a model, it informs the user of a possible state that violates this property. The counterexample produced by Nitpick uses only two hosts to exhibit the flaw in Mobile IPv6.

In what follows we first present a description of the relevant aspects of Mobile IPv6 and a tutorial overview of NP and Nitpick. We then present a line by line description of our NP specification and Nitpick analysis for the acyclicity property that does not hold; we give only a nutshell summary of the specification and analysis for the one that does, since many of the details are similar. We discuss the lessons learned about the appropriateness of using NP and Nitpick to reason about protocols and we conclude with some general remarks about using formalism in the design process.

## 2. Mobile IP

The increased use of mobile computers has grown with the demand to integrate them seamlessly into the Internet. Users want the ability to access their personal files or the Web through their laptop whether they are in the office, at home, or on the road. Internetworking protocols such as IP [P80] used in the Internet today do not currently support host movement. IPv6, the next generation of IP, addresses the need to support mobility [JP96]. This new protocol allows transparent routing of packets to mobile nodes regardless of the mobile node's current point of attachment.

### 2.1. Terminology

A *network* is made of one or more *subnets* (Figure 1). There are two types of network nodes: *routers*, which are used to connect subnets and never move, and *hosts*, which can move.

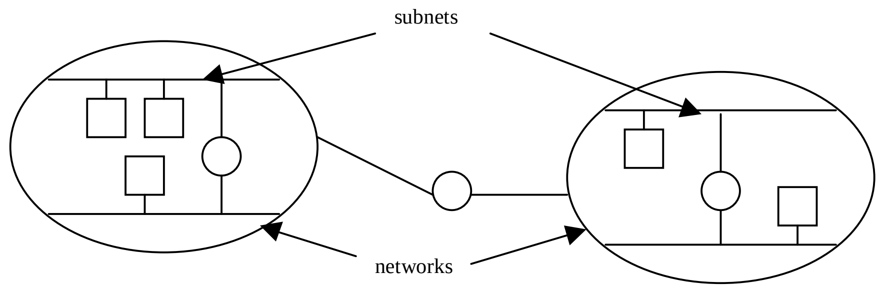


Figure 1: Networks, subnets, routers (circles) and hosts (squares)

Each host has a *home (sub)network*; when it moves it becomes attached to a *foreign network*. Associated with each host is a *home agent*, a router on the host's home network, and a permanent *home address*. A router is responsible for forwarding packets to hosts for which it is the home agent. A *care-of address* is assigned to a mobile node only when and each time it visits a foreign network. A mobile node while away from home has a primary care-of address and possibly other care-of addresses, i.e., those associated with previously visited foreign networks. A *correspondent node*, which can be mobile or stationary, is a peer node with which a mobile node is communicating (Figure 2).

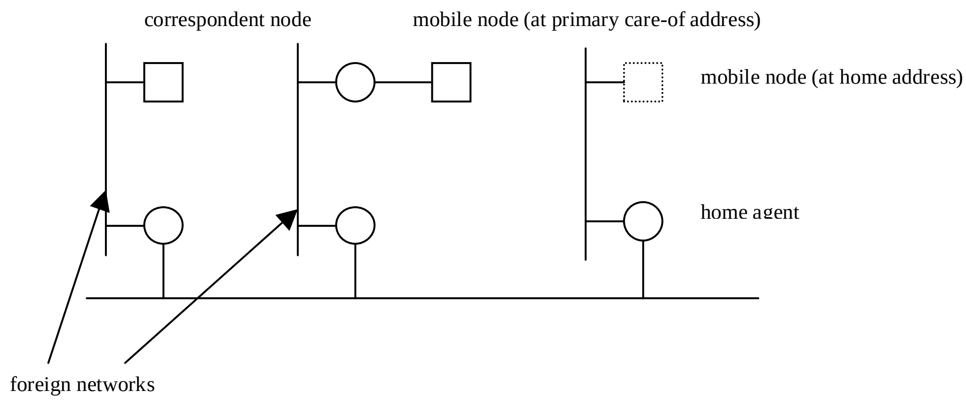


Figure 2: Mobile node at foreign network

Each time a mobile node moves from one IPv6 subnet to another, it changes its primary care-of address, and sends a Binding Update message containing the new care-of address to its home agent. Now when a packet arrives for the host that just moved, the home agent will forward it to the correct address. Home agents keep track of *bindings* between a host's home address and its care-of addresses, as well as a lifetime for each binding; these bindings are kept in a *binding cache*. When the lifetime of a binding expires it is safe to delete that binding from the cache. For efficiency, in Mobile IPv6 (unlike in IPv4), correspondent hosts can send messages directly to the mobile host when it has moved; this avoids routing the message through the home agent, only to get rerouted to the host's new location. Thus, each time a mobile node changes its primary care-of address, it also sends Binding Update messages to each of the correspondent nodes that may have an out-of-date care-of address for the mobile node in its binding cache. Thus mobile nodes as well as routers have binding caches (Figure 3).

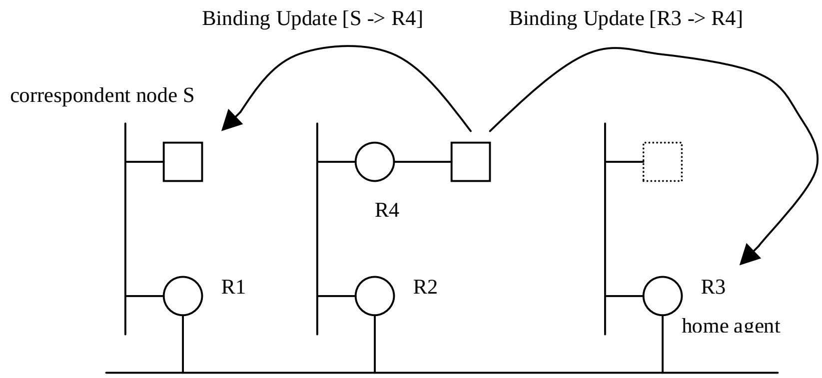


Figure 3. Binding Update messages

By analogy, most students have a permanent home address, e.g., where their parents live, say New York. When a student goes away to school, say at Carnegie Mellon in Pittsburgh, she has mail that is sent to her home address forwarded to her school address (care-of address).[1] Of course she tells her best friends her school address so mail sent by her friends need not be directed through the New York post office (home agent) but can go directly through the Pittsburgh post office. When she goes to work during the summer at yet a third location, say Palo Alto, her summer job address becomes her new primary care-of address, and again she tells her best friends where she is living for the summer. Recall that each of the bindings has a lifetime associated with it; in the case of the summer job, for example, the binding at the Palo Alto post office would be deleted when she starts school in the fall again.

### 2.2. Acyclicity Properties of Mobile IPv6

With all this moving around, especially since a mobile host can return home and visit the same foreign network multiple times, it seems that it would be easy for a message to end up travelling in a circle, forever trying to catch up to a host that is moving around. The crucial question is "Does Mobile IPv6 guarantee that messages never traverse a circular route?" There are two ways that cycles might be introduced in Mobile IPv6 since the routing information for messages is contained in the state of the network in two ways: in the binding caches and in the messages themselves.

From the bindings in the binding caches of the nodes (routers and hosts), we can derive a transitive relation (e.g., [New York -> Pittsburgh] and [Pittsburgh -> Palo Alto] implies [New York -> Palo Alto]) that indicates how messages should be routed. We need to ensure that this transitive relation includes no self-pairs (that is, routes from a host to itself). We call this property *cache acyclicity*. Figure 4 shows an example of cyclic caches.

In our example, this situation would arise if at the end of the summer in Palo Alto the student took a short vacation at home in New York before going back to school in Pittsburgh.

Nodes learn of the new location of a mobile host by information in *binding update* messages each of which essentially says "Mobile node N is at host H." Mobile IPv6 should guarantee that the location information contained in all messages does not form a cycle. We call this property *message acyclicity*. Figure 5 shows an example of cyclic messages.

[1] In the US, the yellow stickers that post offices affix to forwarded mail are the physical analog of changing packet headers to include care-of addresses.

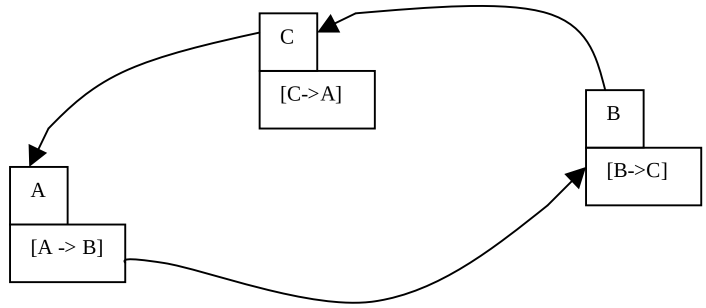


Figure 4: A cache cycle

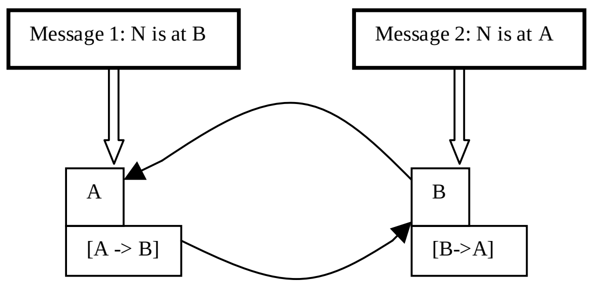


Figure 5: A message cycle

What distinguishes cyclic caches from cyclic messages is that cyclic caches can be formed without the presence of cyclic messages. Even though cyclic messages will eventually lead to the formation of cyclic caches (as in Figure 5), a cyclic cache can form over time with only one message present in the network at a given time. On the other hand, at least two messages are required to form cyclic messages. Because these properties are different, we decided to analyze acyclicity of caches and acyclicity of messages separately. We actually wrote two separate specifications and checked their properties independently. This separation of concerns allowed us to focus on a smaller specification, and to localize the problem we discovered. The smaller specification also resulted in a more tractable analysis.  Section 4 elaborates on these issues.

## 3. NP and Nitpick

### 3.1. NP Specifications

To understand our specification and its analysis, the reader must grasp three fundamental notions – global abstract states, implicit operations, and inductive invariants. These are common to the style of abstract specification represented by languages such as Larch, VDM, and Z. The reader must also overcome two incidental hurdles – the relational operators and the schema notation. These are features of Z, in a slightly adapted form. To illustrate these ideas, we shall use a simplified version of the specification; the full specification is given in Section 4. A full grammar and description of the specification language NP is available as a technical report [JD96a].

***Global Abstract States***

We view the state of the protocol globally, in terms of sets and relations. Rather than thinking about the contents of nodes and messages concretely, we think purely in terms of the abstract graph these contents imply, and its properties. Take a look at Figure 5, for example. We ignore the fact that nodes A and B have local state in which forwarding pointers are stored, and instead focus on the arcs of the graph. The entire collection of forwarding pointers stored at nodes is modeled as a single function

caches: HOST -> HOST

with the interpretation that *caches(h)* denotes the location that *h* believes the mobile host, *N*, to be at. (For simplicity, let's assume there is only one mobile host.) An immediate advantage of this global formulation is that no special value is needed to denote a missing cache entry. The function *caches* is a partial function, so that if host *h* has no forwarding pointer, it is simply omitted from the function's domain, and *caches(h)* is not defined.

More significantly, this formulation allows us to express global properties as simple formulas involving sets and relations. The function *caches* denotes paths one step long between a node and the node to which it forwards messages for the mobile host. Composing it with itself, which we write as *caches ; caches*, we get a relation that denotes paths two steps long. The transitive closure *caches+*, defined as

caches+ = caches U (caches ; caches) U (caches ; caches ; caches) U …

associates each host *h* with all the hosts through which a message originating at *h* will pass. Note that *caches+*, unlike *caches* itself, does not correspond to any data explicitly stored in the system; in modelling the protocol this distinction can be ignored.

To express the desired property that the local caches of hosts do not imply a cycle of forwarding pointers, we just assert that *caches+* has an empty intersection with the identity relation

caches+ & Id = {}

The identity relation contains a pair *(h, h)* for each host *h*. Our assertion thus says that *caches+* contains no such pair: there is no path, direct or indirect, from any host to itself.

Another easily expressed property is that the host at which the mobile host is currently docked has no cache entry for it. Using the variable

router: HOST

to represent that host, we assert

router not in dom caches

We take a similarly global view of messages. We posit an abstract set of messages *MSG* which, like *HOST*, has elements that are structureless identifiers, and three functions that define the origin, destination, and content of messages:

from, to, where: MSG -> HOST

For a message *m*, *from(m)* denotes the host that created the message; *to(m)* denotes its final destination (set by *from(m)*); *where(m)* denotes the believed location of the mobile host conveyed by the message. Note again how we sidestep the question of how and where data is stored: the name of the originating host *from(m)*, is treated no differently from the message's content, *where(m)*. Also, the location of the message in the network is not modeled. This allows us to avoid describing the mechanism by which messages are shunted around; instead we imagine an amorphous pool of messages, with hosts asynchronously inserting and extracting messages. To ensure that a host *h* only extracts a message *m* when addressed to it, we need only assert that

to(m) = h

It is convenient to introduce state variables solely to make the specification easier to read. The variable

updates: set MSG

for example, denoting the set of binding update messages in circulation, is the domain of each of the message functions:

updates = dom from
updates = dom to
updates = dom where

These assertions have two roles; they define the redundant variable *updates*, and constrain the message functions to have the same domains.

***Relational Operators***

We use a set of standard operators familiar to anyone who has studied rudimentary discrete mathematics, although we use ascii variants so that we can manipulate and communicate our specifications without special typesetting tools. So we write *s & t* for the intersection of sets *s* and *t*, rather than $s \cap t$, and *e in s* rather than $e \in s$. NP also includes the very convenient but less widely known operators of the Z specification language. For example,

s <: r

denotes the restriction of the relation *r* to the pairs whose first element belongs to the set *s*. We play fast and loose with these operators, so that the resulting specifications are terse and sometimes a little mystifying. The expression

to~ ; where

for example, denotes the relation that maps a host *A* to a host *B* if there is a message whose destination is *A* and whose content indicates that the mobile host is at *B*.

We often find it helpful to draw entity-relationship diagrams. From Figure 6, which shows the sets *MSG* and *HOST* with the two relations *to* and *where* between them, we can see that the composition of *to~*, the transpose of *to*, with *where*, maps a host to a host, by following *to* backwards for the transpose and then *where* forwards.

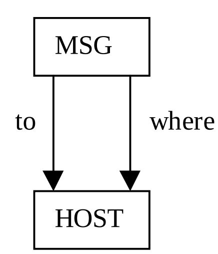


Figure 6: An entity-relationship diagram

***Implicit Operations***

In line with our global view of the state, state transitions are regarded as instances of operations that update the global state. The docking of the mobile host at a fixed host, for example, might be modeled as an operation *mh_arrive* that is parameterized by *h*, the name of the fixed host, and *m*, the binding update message generated. To specify the operation, we write a constraint that relates the global state before execution (given by some set of state variables such as *caches* and *where*) to the global state after execution (given by the same set of variables, primed to indicate that they refer to the post-state):

```
mh_arrive (h: HOST; m: MSG) = [
  caches: HOST -> HOST
  caches': HOST -> HOST
  from, to, where: MSG -> HOST
  from', to', where': MSG -> HOST
  router: HOST
  router': HOST
|
  m not in updates
  from' = from U {m -> h}
  to' = to U {m -> router}
  where' = where U {m -> h}
  caches' = caches
  router' = h
]
```

In contrast to code, this description of the operation is rather abstract. It asserts that some message, not currently in circulation, is created and addressed to the previous location of the mobile, that it originates with the host *h* at which the mobile host has just arrived, and that its contents convey the information that the mobile host is at *h*. Also, the new location of the mobile host is set to be *h*. The specification says nothing about how the message is constructed and sent, nor what identifier is chosen; it simply describes the observable effect – that some message, different from any other in circulation, is constructed and sent. In this style of specification, there are no implicit frame conditions, so we have to say explicitly that this operation has no effect on the host caches.

***Schema Notation***

Explicitly declaring the state variables is tedious and verbose; half of *mh_arrive*'s text is declarations. The Z specification language has a kind of macro feature in which declarations and assertions are bundled together into named *schemas*. Subsequent references to the schema by name cause both its declarations and assertions to be incorporated.

Our specification might then be structured as follows. First we define a schema for the state itself:

```
net = [
  router: HOST
  updates: set MSG
  from, to, where: MSG -> HOST
  caches: HOST -> HOST
|
  updates = dom from
  updates = dom to
  updates = dom where
]
```

Now our operation can be specified more tersely as:

```
mh_arrive (h: HOST; m: MSG) = [
  net
|
  m not in updates
  from' = from U {m -> h}
  to' = to U {m -> router}
  where' = where U {m -> h}
```

```
  caches' = caches
  router' = h
]
```

where mention of the schema *net* not only includes the declarations of the before and after state components (as before), but additionally includes the global state constraints on the before and after states, such as

```
dom from = updates
```

and

```
dom from' = updates'
```

It should now be clear why there are no implicit frame conditions. This latter constraint implies that the variable *updates* must change in concert with *from*; adding the constraint

```
updates' = updates
```

would actually result in an operation with no executions, since there is no assignment of values to the before and after variables for which all the constraints will be true.

Properties of the state space can likewise be written as schemas. Our cache acyclicity property, for example, becomes

```
acyclic_caches = [net | caches+ & Id = {}]
```

***Inductive Invariants***

Given a definition of the states (as some collection of variables) and some operations, we have a state machine. It is not finite, of course, since the primitive types (*HOST* and *MSG*) are unbounded, and there are an infinite number of sets and relations over these types, and thus infinitely many values for state variables such as *caches*.

Induction allows us to reason about such an infinite state machine. Although manual reasoning about even the simplest software design is generally difficult, the underlying principle is straightforward. Suppose the state machine has a set of states S and a transition relation *T: S <-> S*, and starts in a state in the set *S0*. To prove that every reachable state satisfies some property *P*, it is enough to show that *P* holds for every state in *S0*, and that for every transition *(s, s')* in *T*, if *P* holds for *s*, it also holds for *s'*. *P* is then said to be an *invariant* of the transition relation.

The transition relation, in our specification, is given symbolically by the set of operations. The property *P* will be given as a formula. Demonstrating that *P* is preserved by all transitions amounts to proving assertions of the form

```
OP and P => P'
```

where *P'* is the formula for *P* with the state variables primed, for each operation *OP*. For example, to show that the caches never form a cycle, we will have to check assertions like

```
Claim1 (h: HOST, m: MSG) :: mh_arrive (h, m) and acyclic_caches => acyclic_caches'
```

In NP, this is a special kind of schema called a *claim*, about which more will be said later. The double colon distinguishes it from regular schemas belonging to the specification proper. To do a complete analysis, it is necessary to check that every operation maintains the invariant, and that the invariant is established initially. In practice, as here, the analysis is often focused only on a few operations likely to contain errors.

*P* need not in fact be an invariant of the transition relation for it to hold in every state. *T* may include a transition from a state in which *P* holds to a state in which *P* does not hold, but so long as the pre-state is not reachable, this transition will never be executed. This means that sometimes it is necessary to strengthen the property so that such pre-states are deemed not to be acceptable; this can always be done, since we can define a predicate R that characterizes the set of reachable states, and then show that the conjunction of *R* and *P* is preserved. Alternatively, the definition of *T* itself can be altered to eliminate these bogus transitions.

It might seem that this difficulty – not shared by model checking techniques – is wholly bad, and that the user's effort in formulating an invariant is unrewarded. But strengthening the invariant is not an academic exercise, and leads to a more robust design. If a property *P* is true in all reachable states but not preserved by the transition relation, this suggests a fragility in the design. A modification of one operation that changes which states are reachable may now cause another operation to break. Requiring the strengthening of the invariant or an explicit restriction of the transition relation (by strengthening the operation's precondition) is necessary to ensure that the operation will work irrespective of modifications to other operations. Modularity in reasoning thus corresponds to modularity in design.

### 3.2. Nitpick

Nitpick is a specification checking tool. At its core lies a model finder for relational formulas. Given a formula with some variables denoting scalars, sets and relations, Nitpick searches for a model – an assignment of values to the variables for which the formula is true. For an operation schema, each model is a transition; Nitpick can therefore simulate execution of the operation. For a claim, Nitpick searches for a model not of the formula but of its negation; these models are counterexamples that refute the claim. Presented with the claim

Claim1 (h: HOST, m: MSG) :: mh_arrive (h, m) and acyclic_caches => acyclic_caches'

for example, Nitpick will search for a model of

mh_arrive (h, m) and acyclic_caches and not acyclic_caches'

Such a model is an instance of a transition of the operation *mh_arrive* from a valid state to an invalid state. If it exists, then, it demonstrates that the invariant does not hold.

The language of first-order formulas involving binary relations is not decidable, so a search for a model cannot terminate if no model exists. Nitpick therefore prompts the user to provide a *scope* that bounds the number of elements in the primitive types. We might select, for example, a scope that bounds *HOST* by 2 and *MSG* by 3; this will result in a search for models that involve at most two hosts and three messages.

The search for models rapidly becomes intractable as the scope is increased. Fortunately, however, many errors in designs can be exposed by considering only a small scope. In our study of IPv6, there were no interesting cases that required more than two hosts and two messages. Moreover, by using various reduction mechanisms, models can often be found even though the space of possible models is huge. These mechanisms, and the rationale behind Nitpick, are discussed elsewhere [JD96b, JJD97].

## 4. NP Specifications and Nitpick Analysis of Mobile IPv6

We present in detail the NP specification and Nitpick analysis of the cache acyclicity property since that is where we found a design flaw. We only briefly discuss the specification of the message acyclicity property, highlighting how we modeled certain details about messages and why the acyclicity property holds given our model. Details of both the specification and the analysis for message acyclicity can be found in [Ng97].

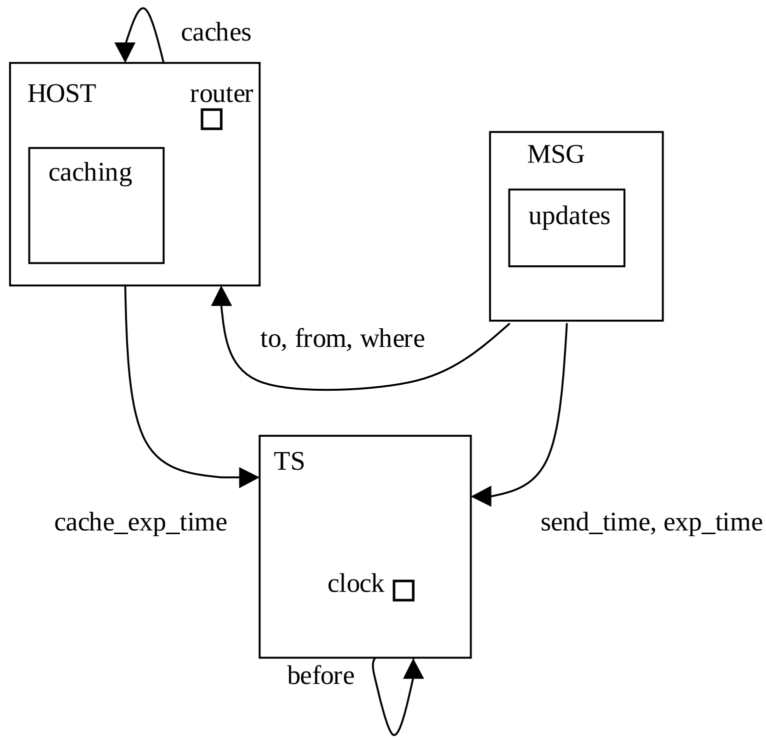


Figure 7: The entity-relationship diagram corresponding to the declarations of the schema *Net*

**4.1. Specification for Cache Acyclicity**

For cache acyclicity, we focus on how binding caches of the hosts change when the mobile host arrives at a new (sub)network and when the correspondent hosts receive a binding update message sent by the mobile host. Figure 7 gives the entity-relationship diagram for the state space and Figure 8 gives the full specification for the state space of the network and the two relevant operations, *mh_arrive* and *update_arrival*. As before, we assume for simplicity that there is only one mobile host.

**The Network**

We build upon the specification presented in Section 3 by adding the notion of time into the model. In IPv6, as with many network protocols, any message travelling in the network is associated with an expiration time after which the message is silently discarded when received by hosts. Using expiration times is necessary to prevent the indefinite circulation of messages in the network which could lead to unnecessary consumption of network resources. We define a variable *clock* to represent the current global time. The send and expiration times of packets sent from different hosts are based on the same clock, and all caches determine the expiration time of a cache entry by consulting this global clock. Lamport's synchronized clocks would be one way to achieve a close approximation of a global clock [L78].

We now walk through the specification line by line. The first line declares three basic types: HOST, MSG, and TS (for timestamps).

```
[HOST, MSG, TS]

Time = [
  clock: TS
  const timeorder: tot seq TS
  const before: TS -> TS
|
  before = next timeorder
  not before = {} ]

Hosts = [
  caches: HOST -> HOST
  caching: set HOST
  cache_exp_time: HOST -> TS
|
  dom cache_exp_time = dom caches
  caches & Id = {} ]

Messages = [
  updates: set MSG
  from,to,where: MSG -> HOST
  send_time, exp_time: MSG -> TS
|
  updates = dom to
  updates = dom from
  updates = dom where
  updates = dom send_time
  updates = dom exp_time
  (from; from~) & (to; to~) & (send_time; send_time~) <= Id
  (from~; to) & Id = {} ]

Net = [
  router: HOST
  Time
  Hosts
  Messages
|
 exp_time <= send_time; before+ ]

mh_arrive (h:HOST; m:MSG; t:TS) = [Net |
  not router = h
  router’ = h
  not m in updates
  t in before+.{clock}
  clock’ in before+.{clock}
  caching’ <= caching
  cache_exp_time’ = caching’ <: cache_exp_time :> (before+).{clock’}
  caches’ = dom(cache_exp_time’) <: caches
  updates’ = updates U {m}
  to’ = to U {m -> router}
  from’ = from U {m -> h}
  where’ = where U {m -> h}
  send_time’ = send_time U {m -> clock}
  exp_time’ = exp_time U {m -> t}
]
```

```
update_arrival (m:MSG; keeps: set HOST) = [
  Net
  const Messages
|
  clock' in before+.{clock}
  keeps <= caching
  caching' = {to.m} U keeps
  cache_exp_time' = caching' <: (cache_exp_time (+) {to.m -> exp_time.m}) :> (before+.{clock'})
  caches' = dom (cache_exp_time') <: (caches (+) {to.m -> where.m})
  router' = router
]
```

Figure 8: The specification of the network and two operations.

The state space is decomposed into several schemas: *Time* holds the clock and the orderings on time stamps; *Hosts* holds the host caches; and *Messages* holds the contents and locations of the messages. These are brought together in the schema *Net*.

Consider *Time* first. The variable *clock* models the current (global) time; *timeorder* denotes a sequence that defines a total order on the set of timestamps. (Both *tot* and *seq* are Nitpick keywords[JC96].) The function *before* denotes a binary relation on timestamps. The constraints of this schema are required, for technical reasons, to obtain an ordering of time stamps with the expected properties.

The variables of the schema *Hosts* model the cache entries. Given a host *h*, *caches(h)* denotes the location at which *h* believes the mobile host to be; and *cache_exp_time(h)*, the expiration time of *h*'s cache entry for the mobile host. The variable *caching* models the set of hosts that have a cache entry for the mobile host. The assertions of the schema say that a cache entry is always associated with an expiration time (after which the entry becomes invalid and will be deleted) and that there are no cache cycles.

In the *Messages* schema, *updates* denotes the set of binding update messages in circulation. Given a message *m*, *from(m)* denotes the host that created *m*; *to(m)*, *m*'s destination host; *where(m)*, the believed current location of the mobile host as contained in *m*; *send_time(m)*, the time at which *m* is sent; and *exp_time(m)*, the time at which *m* will expire. The constraint

```
(from; from~) & (to; to~) & (send_time; send_time~) <= Id
```

expresses the uniqueness property of a message sent from one host to another at a given time. The expression *from; from~* denotes the equivalence relation over all messages sent by the same host, and similarly, for the expressions *to; to~* and *send_time; send_time~*. The whole constraint thus says that any two messages *m1* and *m2* with the same sender (*from.m1 = from.m2*), the same receiver (to.m1 = to.m2) and the same send time (*send_time.m1 = send_time.m2*) are the same (*m1 = m2*). The last constraint in the schema says that the sender and the receiver of a message cannot be the same.

By including the schemas *Time*, *Hosts* and *Messages*, the schema *Net* incorporates not only their state components but also the relevant constraints. It adds a variable of its own – *router*, which denotes the host at which the mobile host is currently docked – and an additional constraint: that the expiration time of a message is always later than its send time. Let's examine this constraint in detail. In the subexpression

```
send_time; before+
```

we apply the forward composition operator to *send_time* and *before+* to obtain a function that maps a message *m* in the domain of *send_time* to a timestamp which is a successor of *m*'s send time. The full expression

```
exp_time <= send_time; before+
```

then imposes the constraint relating *m*'s expiration time to its send time. (Don't be confused by the ascii representation of subset: <= is not the "less than or equal" operator on timestamps!).

**The Two Operations**

The operation *mh_arrive* models the docking of the mobile host at a new host. It takes three parameters: *h*, the host at which the mobile host has just arrived; *m*, the message being sent to the previous router of the mobile host to inform it of the mobile host's current location; and *t*, the timestamp used as the expiration time of the binding update message. The first two lines

```
not router = h
router' = h
```

say that the new location of the mobile host, which is different from its previous location, is *h*. The third line

```
not m in updates
```

specifies a pre-condition that *m* is currently not in circulation. The next line

```
t in before+.{clock}
```

checks that the parameter *t* is later than the current clock time. The expression *before+.{clock}* denotes the set of timestamps consisting of successors of the clock's current time. The parameter *t* is later used as the message *m*'s expiration time. Similarly, the expression

```
clock' in before+.{clock}
```

advances the clock. The assertion

```
caching' <= caching
```

together say that the new set of caching messages is a subset of the old, thereby allowing any host to drop any cache entry according to its own cache replacement policy. This loose specification is an example of abstracting away from the details of the system: IPv6 actually uses a least recently used cache replacement policy.

The heart of the behaviour of *mh_arrive* is expressed in the next two lines

```
cache_exp_time' = caching' <: cache_exp_time :> (before+).{clock'}
caches' = dom(cache_exp_time') <: caches
```

which update *cache_exp_time* and *caches*. Updating *cache_exp_time* involves retaining only the set of cache entries that have neither been selected for dropping by the local cache replacement policy nor reached the expiration time associated with them. The two conditions are satisfied respectively by restricting the domain of the pre-state of *cache_exp_time* to the post-state of *caching* and its range to the set of timestamps that are strictly after the clock's updated time. The variable *caches* is updated to ensure that its domain and *cache_exp_time*'s domain are the same. Finally, the expressions

```
to' = to U {m -> router}
from' = from U {m -> h}
where' = where U {m -> h}
send_time' = send_time U {m -> clock}
exp_time' = exp_time U {m -> t}
```

say that *m* will be circulating in the network and record the information associated with the message *m*: *m*, which will expire at some future time, *t*, is sent by *h* at the current *clock*'s time to the previous *router* informing it that the mobile host is currently at *h*.

The second operation *update_arrival*, parameterized by a message *m* and a set of hosts *keeps*, describes what happens when a message arrives at its destination. The receiver simply modifies the cache entry for the mobile host according to the information in the binding update message. We explain only the significant differences between its set of constraints and *mh_arrive*'s.

First, note that declaring the schema *Messages* to be constant asserts implicitly that no component of *Messages* changes, and thus obviates the need for constraints such as

updates' = updates

The variable *caching* is updated by the assertions

keeps <= caching
caching' = {to.m} U keeps

As before, the subset relation abstracts from the possibly different local cache replacement policies of the different hosts. Some arbitrary set of hosts drop the entries from their caches. The union operation, together with the relation override operation ((+)) used in the following expressions, imply that the corresponding cache entry for the mobile host is modified unconditionally when the destination host receives the binding update message. The expressions

cache_exp_time' = caching' <: (cache_exp_time (+) {to.m -> exp_time.m}) :> (before+.{clock'})
caches' = dom cache_exp_time' <: (caches (+) {to.m -> where.m})

update the variables *cache_exp_time* and *caches* in the same way as we did for *mh_arrive*, except that we now insert a new entry into the cache (or modify an existing one) based on the information contained in the message. There are two implications for the unconditional modification of the cache entries. The first is that we are assuming that a cache has infinite storage capacity. The use of the union and relational override operators means that we are adding a new entry into the cache regardless of whether there is any space left in the cache. The second implication is that while it is true that the cache can accommodate the new binding update message even if it is full (in which case the host can selectively replace an existing cache entry with a new according to its own cache replacement policy), the use of the relational override operator in the specification means that the host is deprived of the freedom to choose whether or not to accept the binding update message and modify the cache entry accordingly. In a real network, of course, the host can accept the arriving packet at its discretion. As far as our cyclic cache property is concerned, however, discarding the message would not lead to any change to our model.

### 4.2. Analysis for Cache Acyclicity

The property we want to check is that the cache entries of the local caches of hosts do not form a cycle of forwarding pointers. If there were a cycle in the caches then a message could circulate in the network indefinitely, obviously an undesirable behaviour. Recall that we can state this property as an invariant in NP

acyclic_caches = [Net | caches+ & Id = {}]

The NP claims that *mh_arrive* and *update_arrival* preserve cache acyclicity are

Claim1 (h:HOST; m:MSG; t:TS) ::
mh_arrive (h, m, t) and acyclic_caches => acyclic_caches'

```
Claim2 (m:MSG; ks: set HOST) ::
        update_arrival (m, ks) and acyclic_caches => acyclic_caches’
```

We execute the claims with the three basic types bounded by the scopes |*HOST*| = 3, |*MSG*| = 1 and |*TS*| = 3. Here is the justification for our choice: For *HOST*, although it suffices to model a cyclic cache with just two hosts, a less trivial case would be if three hosts were involved. For *MSG*, one message suffices since each operation requires only one message for it to execute. For *TS*, at least two timestamps are needed since two are needed for each operation to proceed (to update the clock during the transition). However, with only two timestamps, all cache entries will reach their expiration time when the clock is being updated, which would trivially eliminate the possibility of forming a cycle of forwarding pointers; hence we need at least three timestamps. In practice, we use Nitpick iteratively by trying different combinations of scopes – both to validate our understanding of what is necessary and sufficient to establish or disprove a claim and to keep the turnaround time for Nitpick feedback within reasonable bounds.

The output from the execution of *Claim1* is shown in Appendix 1. Nitpick fails to find any counterexamples. We can in fact argue that the claim holds for any scope. In *mh_arrive* we update *cache_exp_time* by trimming out the out-of-date cache entries and by not adding any new entries. By restricting the domain of *caches* to be the same as that of *cache_exp_time*, we are in the best case not removing anything from *caches* and certainly not adding anything to it. Thus, if we know that *caches* was acyclic before executing the operation it remains acyclic after.

The output from the execution of *Claim2* is shown in Appendix 2. Nitpick produces a counterexample. Let us use Nitpick’s output to help us figure out what the problem is. The only relevant effect of executing *update_arrival* is that the receiver of the binding update message, *m*, modifies its cache entry for the mobile host according to the information in *m* about the current location of the mobile host. The relation *before* gives an ordering on the timestamps *t0*, *t1*, and *t2* such that *t0* < *t1* < *t2*, as reflected in the lines

```
before: TS <-> TS is
 { t0 -> t1,
 t1 -> t2 }
```

The lines

```
cache_exp_time: HOST -> TS is
 { h1 -> t2 }
cache_exp_time’: HOST -> TS is
 { h0 -> t2,
  h1 -> t2 }
```

confirm that the cache entries for both *h0* and *h1* have not yet expired, since the expiration time for both is *t2*, while the lines

```
clock: TS is t0
clock’: TS is t1
```

indicate that the current time is *t1*. The lines

```
caches’: HOST -> HOST is
{ h0 -> h1,
 h1 -> h0 }
```

show that the two hosts, *h0* and *h1*, are involved in the cyclic set of cache entries; each host thinks that the mobile host is docked at the other. The lines

```
router: HOST is h1
router’: HOST is h1
```

show that the mobile host is docked at router *h0* throughout the operation. Before the operation, that is, before the binding update message is received by *h0*, the cache entry in *h1* points to *h0*, as shown in the lines

caches: HOST -> HOST is
{ h1 -> h0 }

which means that *h1* thinks that the mobile host resides at *h0*. During the operation, router *h0*, as shown in the lines

to: MSG -> HOST is
{ m0 -> h0 }

receives the message *m0*, given by

m: MSG is m0

from *h1* informing *h0* that the mobile host is currently at *h1*, indicated by the lines

where: MSG -> HOST is
{ m0 -> h1 }

and hence requesting *h0* to create a new cache entry point to *h1* for the mobile host. It is this modification of the cache entry for *h0* that results in a cycle. After the operation, *h0* contains a cache entry pointing to *h1* and *h1* still contains an unexpired cache entry pointing to *h0*.

At first glance it may seem strange that *h1* thinks that the mobile host is at *h0* even though it is actually at *h0* before and after the operation. Here is the key insight to how this scenario can arise for real: *h1* is a router that the mobile has visited previously. Before the operation, the mobile host has just returned to the same location again, that is, *h1*. It is plausible for *h1* to contain a cache entry for the mobile host since when the mobile host left *h1* in an earlier visit to *h0*, it sent *h1* a binding update message saying that it has moved to *h0*. This message causes *h1* to create a cache entry pointing to *h0* for the mobile host. Some time later when the mobile host moves back from *h0* to *h1*, the mobile host similarly sends a binding update message to *h0* requesting *h0* to create a cache entry pointing to *h1*, resulting in the formation of a cycle.

The counterexample reflects a subtle flaw in the Mobile IPv6 protocol. Nowhere in the draft standard is it specified that the mobile host, *h*, needs to inform router *h1* that it is revisiting *h1* and thus to request *h1* to remove its cache entry for *h*. In fact, the draft standard provides no mechanism (e.g., no message type) that would enable the mobile host to tell the router to stop forwarding messages destined for the mobile host when it comes back to the same location. Since the mobile host is obliged to send a binding update message to *any* previous router, including one it has already visited, that previous router will make a cache entry for the mobile host. Since there is no means for the mobile host to inform the router that it is revisiting, a cyclic set of cache entries can be formed. This flaw exists in IPv6 but not in IPv4. So one way to resolve the bug is to adopt IPv4's solution, which is to require that the mobile host send a message to the router saying that it has returned so the router can stop forwarding messages for the mobile host. In talking with the IPv6 designers, it seems that they wanted to reduce the number of messages required to support mobility, given the inevitable increase in the number of messages to be exchanged. Whereas IPv4 requires this additional message needed to inform *h1* of the mobile host's return to *h1*, this message is omitted from IPv6's design.

Finally note that although the cyclic cache counterexample is formed between only two hosts, the scenario obviously generalizes to the case where any number of hosts (greater than two) are involved in a cycle.

### 4.3. Acyclicity of Messages

Message acyclicity means that location information contained within all circulating messages does not form a cycle. Determining that message acyclicity holds hinges largely on our abstract model of global time and correspondingly, our use of lifetimes in messages. At any point in time, if there are two or more binding update messages for a mobile host circulating in the network, only one is ever considered *valid* where validity depends on the message expiration time. Message acyclicity considers only valid messages; and since there is only at most one valid message per mobile host, no cycle can ever form.

Thus, we need to assume that at any point in time we can always determine which messages are valid or not. In particular, IPv6 requires that upon receipt of a binding update message a host determines whether the message is valid by checking that it has not reached its maximum lifetime and that its lifetime is nonzero; furthermore, all other binding update messages for that same mobile host are considered invalid. If an invalid binding update message is received by a host then it is silently discarded.

In our specification, since we model the net globally, it is easy to state in one fell swoop that a subset of messages in the network become invalid: We model a set of *valid* messages and simply update this variable appropriately in both the *mh_arrive* and *update_arrival* operations. Without explaining the (admittedly obscure) right hand side of the constraint, here are the relevant lines extracted from the specification for *mh_arrive*:

```
valid: set MSG

valid’ = (valid U {m}) \ ((dom (exp_time’ :> (before~)+.{clock}))
          U (dom ((send_time’; before; before) :> (before~)+.{clock’})))
```

The point is that global state variable *valid* gets updated in some way. (Implementing this effect of an atomic update to a global set of messages relies on the use of sequence numbers in IPv6 message headers so a host can determine recency and duplication of received messages.)

In our Nitpick analysis of message acyclicity for scopes of three hosts, three messages, and three timestamps, Nitpick failed to find any counterexamples.

Intuitively, here is why. When the mobile host moves from router *h0* to *h1*, it sends a binding update message *m* to router *h0* so that *h0* will be able to cache its current location. If there are any binding update messages other than *m* with destination *h0* that are currently circulating in the network, these messages will become invalid since *m* is a message sent by the mobile host and therefore contains the most updated information about the mobile host. In our model, when a mobile host sends a binding update message to a correspondent host, all other (circulating) binding updates currently sent to this correspondent host become invalid and hence will subsequently be discarded when received by the host. Thus we eliminate the possibility of the formation of a message cycle during the execution of either the *mh_arrive* or *update_arrival* operations.

## 5. Discussion

This case study was done not only to specify properties of Mobile IPv6 formally but also to test the limitations of NP and Nitpick as a formal method. In this section we also reemphasize, giving specific examples, the importance of modularity and abstraction in writing formal specifications.

### 5.1. Expressiveness of NP

NP has no quantifiers. In theory, quantifiers are never necessary: any first-order constraint can be written with relational operators alone. In practice, the omission of quantifiers can make constraints hard to read and write. Here is an example taken from our specifications. Suppose we want to say that two messages with the same source, the same destination, and the same send time must be the same. With quantifiers, we can express the uniqueness property as

$$\forall \text{ m1, m2 : MSG.}$$
$$\text{from.m1 = from.m2} \wedge \text{to.m1 = to.m2} \wedge \text{send\_time.m1 = send\_time.m2} \Rightarrow \text{m1 = m2}$$

In contrast, NP forces us to write:

(from; from~) & (to; to~) & (send_time; send_time~) <= Id

Some users find this annoying, but others take readily to this style. There are in fact several recurrent idioms which, once familiar to the specifier, make such constraints easy to handle. The expression *(f ; f~)* for example, is commonly used to define, for a function *f*, the equivalence that equates domain elements mapped to the same range element.

### 5.2. Effectiveness of Nitpick As a Specification Tool

Because NP is not decidable, automatic checking cannot be sound and complete. Nitpick cannot verify properties; when no counterexample is found, there is no guarantee that one would not be found by a search in a larger scope. This limitation only diminishes Nitpick's utility when assurance is the aim of the analysis. When the aim is to find violations of properties as rapidly and easily as possible, Nitpick works well.

Nitpick currently handles only small specifications, and can only analyze small scopes. Despite the various reduction mechanisms used by Nitpick to reduce the search space (derived variable analysis, short-circuit enumeration, and isomorph elimination), for our specifications it takes a thousand minutes to complete the search (i.e., in the absence of counterexamples) for a scope of $|HOST| = 3$, $|MSG| = 3$ and $|TS| = 3$.

A subtle change in the specification can have a major effect on Nitpick's performance. For example, when a state variable is declared as a function, Nitpick will only enumerate function values. If, however, one merely asserts in the body of the specification that the variable is a function, the variable is treated as an arbitrary relation, and non-functional values are generated and then subsequently eliminated by testing – a process that can take much longer.

Despite its limitations, Nitpick is easy to use. One of the authors was a relative novice to formal specification, and found that it was particularly helpful to have Nitpick generate instances of schemas and thus demonstrate consistency. Using Nitpick makes specifying a more compelling and enjoyable activity.

### 5.3. How Nitpick Influenced the Way the Specification is Written

Nitpick cannot generally analyze a specification that was developed without Nitpick in mind. However, by careful and tailored application of two standard principles – modularity and abstraction – it is possible to obtain a specification that is both faithful to the problem and analyzable.

*Modularity*. The modularization of a specification involves breaking the specification that is composed of *n* properties of the system, where $n >= 2$, into *n* separate specifications, each of which attempts to capture a single property. Presumably, each of the separate specifications is smaller in size when compared to the original specification and consequently, modularity makes the specification more readable. The major benefit that comes with modularity, however, is the reduction in runtime, which is especially desirable in light of the problem of Nitpick's limitation to handling only small specifications because of the amount of time required to search through a given state space. Decomposing the specification results in the possibility of eliminating variables and operations that are irrelevant to the property we are trying to prove of a decomposed specification. Here is an application of the principle of modularity in our specifications: We postponed the introduction of the definition of validity until we presented the specification for message acyclicity, since the notion of validity is irrelevant to proving the cache acyclicity property. For the same reason, the state variables *caches* and *cache_exp_time*, which are specific to cache acyclicity, are omitted from the second specification. The smaller the number of state variables is, the smaller the number of cases the checker needs to examine and hence the smaller the amount of time the checker takes to search through the entire space.

*Abstraction*. The complexity involved in a software system in the real world makes it difficult to manage. A specification gives us a nice abstraction that hides the details of the system in the model. Here are three applications of abstraction in our specifications. Although it is natural to think of incorporating these details into the specifications so as to make them more complete, they are irrelevant with respect to the two properties we are trying to prove and hence will only be a source of overhead.

- Acknowledgment messages: It is specified in the Mobile IPv6 protocol that the correspondent host should send back an acknowledgment message upon the receipt of a binding update message to its sender. The reason we can abstract from modelling these acknowledgment messages is that the sole function of an acknowledgment message is to acknowledge the receipt of a data message by a receiver without altering the cache entries of the hosts and hence these messages are irrelevant to the properties we are trying to prove.

- Encapsulation and tunneling of packets: Encapsulation is the process of wrapping a network header to the packet which contains the routing information. Tunneling is simply the path followed by a packet while it is encapsulated. Encapsulation and tunneling occur only when the correspondent node sending the packet to the mobile host cannot find its binding cache entry for the mobile node's care-of address, in which case the correspondent node sends that packet to its home agent where the packet will be intercepted, encapsulated, and then tunneled to the home agent. Our ability to abstract from modelling encapsulation and tunneling arises from the fact that we are concerned only with the possibility of the formation of a message cycle, regardless of whether the messages are encapsulated/tunneled. Essentially we take into account every valid message that is sent across the network and do not distinguish between encapsulated/tunneled packets and non-encapsulated/non-tunneled packets.

- Home agents: A home agent is a router in the home network of the mobile host that encapsulates packets destined for the mobile host and tunnels them to its current location while it is away from the home network. Modelling the home agent is likely to introduce a serious amount of complications, since the home agent performs specific functions for the mobile host and hence has to be treated differently from our routers in the network. On the other hand, the presence of a home agent does not play any role in the formation of a cyclic cache, since the correspondent hosts do not cache the location of the mobile host when it is at home.

**6. Conclusions**

Our study of IPv6 illustrates, more generally, our attitude to formal methods [JW96]. Formality is not an end in itself, and indeed we recognize that a formal specification is usually harder to write (and read) than an informal one. Not many problems can be solved entirely by formal means, or merit complete formalization. But at the same time, most practical software designs have subtle or critical aspects that can be effectively treated with formal methods. Using abstraction, it is possible to specify and analyze such aspects in isolation, expending only as much effort as is needed to understand the design's subtleties and expose its deficiencies.

Judiciously applied, then, formalism can be of great benefit. Here, it helped in two ways. First, describing the protocol in terms of abstract, global states made it easier to capture the properties motivating its design. The designer of the protocol complained that our formulation was at variance with his operational intuitions, but paradoxically, it seems that it was precisely taking a different viewpoint that exposed the protocol's flaws. After we walked him through an earlier version of an NP specification of Mobile IPv4 during which we discovered a critical design subtlety, he readily and admittedly added assertional thinking to his repertoire of "mental tools". Second, casting the description in a formal notation made it amenable to automatic analysis. Whether we would have found the flaws without Nitpick is not clear, but a manual analysis would certainly have been tedious and error prone.

**Acknowledgments**

This work was done when the first and second authors were at Carnegie Mellon University. The authors are grateful to David Johnson, the principal designer of the mobile protocol studied here, for patiently explaining the operation of the protocol and its intricacies.

This research is sponsored in part by the Defense Advanced Research Projects Agency and the Wright Laboratory, Aeronautical Systems Center, Air Force Materiel Command, USAF, F33615-93-1-1330, and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-2-0031 and in part by the National Science Foundation under Grant No. CCR-9523972. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright annotation thereon.

The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Defense Advanced Research Projects Agency Rome Laboratory or the U.S. Government.

**References**

[JD96a] Daniel Jackson and Craig A. Damon. Nitpick Reference Manual. CMU-CS-96-109. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1996.

[JD96b] Daniel Jackson and Craig A. Damon. "Elements of Style: Analyzing a Software Design Feature with a Counterexample Detector," *IEEE Transactions on Software Engineering*, Vol. 22, No. 7, July 1996, pp. 484–495.

[JJD97] Daniel Jackson, Somesh Jha and Craig A. Damon, "Isomorph-free Model Enumeration: A New Method for Checking Relational Specifications," to appear, *ACM Transactions on Programming Languages and Systems.*

[JW96] Daniel Jackson and Jeannette Wing, "Lightweight Formal Methods," *IEEE Computer*, April 1996, pp. 21-22.

[JP96] David B. Johnson and C. Perkins, "Mobility Support in IPv6," Mobile IP Working Group INTERNET-DRAFT, June 1996.

[J95] David B. Johnson, "Scalable Support for Transparent Mobile Host Internetworking," *Wireless Network*, Vol. 1, October 1995, pp. 311-321.

[L78] Leslie Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," *CACM*, Volume 21, Number 7, July 1978, pp. 558-565.

[P80] J.B. Postel, "Internetworking Protocol Approaches," IEEE Transactions on Communications, April 1980, Vol. 28, pp. 604-611.

[S92] J.M. Spivey, The Z Notation: A Reference Manual, second edition, Prentice-Hall, 1992.

**Appendix 1: Nitpick Transcript for Analysis of Claim1**

Checking claim "Claim1"
Creating Executable …
Finished creating executable
Elapsed time 0:00.21
  Isomorph elimination on
  Derived variable detection on
  Short circuiting on
  Restricting HOST to 3 elements { h0, h1, h2 }
  Restricting MSG to 1 elements { m0 }
  Restricting TS to 3 elements { t0, t1, t2 }
  …
  Completed checking entire case space

Finished checking claim "Claim1"
After checking 122299 cases of 3.91379e+11 possible
  220 unlabeled cases examined
  10756 unlabeled cases skipped due to short-circuiting
No counter examples were found
Executed 1.16908e+07 instructions checking claim
Elapsed time 0:48.88

**Appendix 2: Nitpick Transcript for Analysis of Claim2**

Checking claim "Claim2"
Creating Executable ...
Finished creating executable
Elapsed time 0:00.66
  Isomorph elimination on
  Derived variable detection on
  Short circuiting on
  Restricting HOST to 3 elements { h0, h1, h2 }
  Restricting MSG to 1 elements { m0 }
  Restricting TS to 3 elements { t0, t1, t2 }
...

Claim "Claim2" is contradicted in case:

cache_exp_time: HOST -> TS is
  { h1 -> t2 }
cache_exp_time': HOST -> TS is
  { h0 -> t2,
    h1 -> t2 }
caching: Set HOST is { h1 }
caching': Set HOST is { h0, h1 }
caches: HOST -> HOST is
  { h1 -> h0 }
caches': HOST -> HOST is
  { h0 -> h1,
    h1 -> h0 }
clock: TS is t0
clock': TS is t1
exp_time: MSG -> TS is
  { m0 -> t2 }
exp_time': MSG -> TS is

```
  {m0 -> t2}
from: MSG -> HOST is
  { m0 -> h1 }
from': MSG -> HOST is
  {m0 -> h1 }
before: TS <-> TS is
  { t0 -> t1,
    t1 -> t2 }
m: MSG is m0
router: HOST is h1
router': HOST is h1
send_time: MSG -> TS is
  { m0 -> t0 }
send_time': MSG -> TS is
  { m0 -> t0 }
ks: Set HOST is { h1 }
timeorder: tot seq TS is < t0, t1, t2 >
to: MSG -> HOST is
  { m0 -> h0 }
to': MSG -> HOST is
  { m0 -> h0 }
updates: Set MSG is { m0 }
updates': Set MSG is { m0 }
where: MSG -> HOST is
  { m0 -> h1 }
where': MSG -> HOST is
  { m0 -> h1 }

...
Completed checking entire case space

Finished checking claim "Claim2"
After checking 846411 cases of 4.34865e+10 possible
  301 unlabeled cases examined
  24787 unlabeled cases skipped due to short-circuiting
960 counter examples found
Executed 6.38056e+07 instructions checking claim
Elapsed time 5:01.35
```