Formal Aspects of Computing (1999) 11: 591–615

Formal Aspects of Computing

A Nitpick Analysis of Mobile IPv6

Daniel Jackson[1], Yu-Chung Ng[2] and Jeannette Wing[3]

[1] MIT Lab. for Comp. Sci. Cambridge, MA, USA;
[2] Computer Science Dept. Cornell University, Ithaca, NY, USA;
[3] Computer Science Dept. Carnegie Mellon University, Pittsburgh, PA, USA

Keywords: Declarative specification, Model checking, Nitpick, Z, Mobile internetworking protocols

Abstract. A lightweight formal method enables partial specification and automatic analysis by sacrificing breadth of coverage and expressive power. 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 design flaw: 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 sufficient 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

Correspondence and offprint requests to: Daniel Jackson, Email: dnj@lcs.mit.edu

version, IPv4 [Joh95], 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 state machines with complex relational state. By design, NP is roughly a subset of Z [Spi92]. Although few existing Z specifications are within NP, many can be straightforwardly translated into NP (e.g, by eliminating unavailable constructs, most notably quantifiers). Hence, Nitpick is a semantic analysis tool for a subset of Z; it complements both Z animators, such as ZAL [MSH98] and PIZA [HOS97], and Z-based theorem provers, such as Z/Eves [Saa97] and HOL-Z [HOLZ].

Like its successful model checker counterparts, which inspired Nitpick's development, 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.

Protocols of this sort have been specified in a state-based manner in languages such as Z, but as far as we know, have never been analysed automatically. Model checkers have been used to analyse protocols, but they require the specification to be written in a more operational form, in terms of local actions on local state. Our case study shows that it is possible to bring full automation to declarative, state-based specification, in which properties of the global state can be described directly and succinctly.

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 close 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 [Pos80] used in the Internet today do not currently support host movement. IPv6, the next generation of IP, addresses the need to support mobility [JoP96]. 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* (Fig. 1). There are two types of network nodes: *routers*, which are used to connect subnets and never move, and *hosts*, which can move.

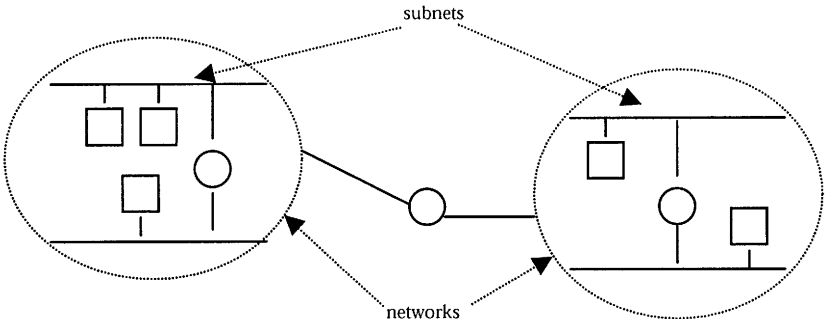

Fig. 1. Two networks connected by a router. Each network has two subnets. Routers are depicted as circles; *hosts*, as squares.

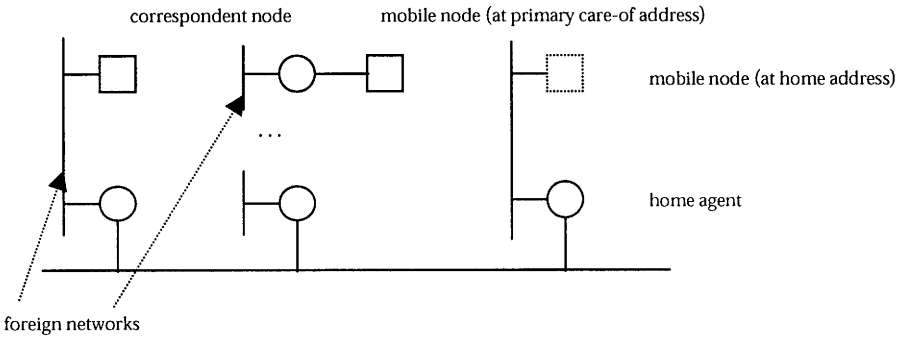

Fig. 2. Mobile node at foreign network.

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 (Fig. 2).

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

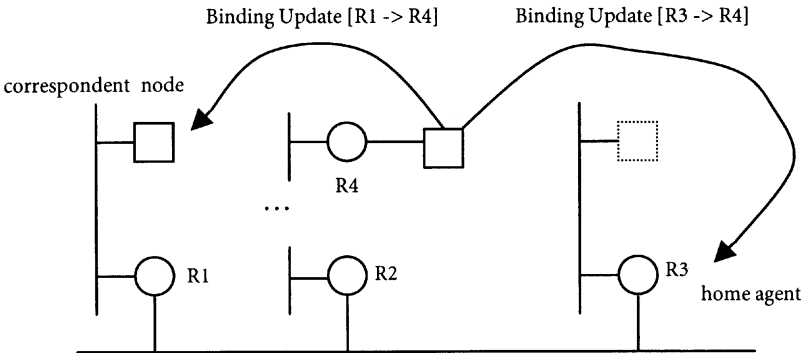

Fig. 3. Binding Update messages.

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 (Fig. 3).

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). (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.)

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. One desired property is that this transitive relation include 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

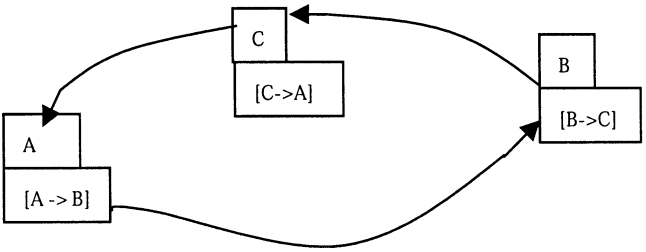

Fig. 4. A cache cycle.

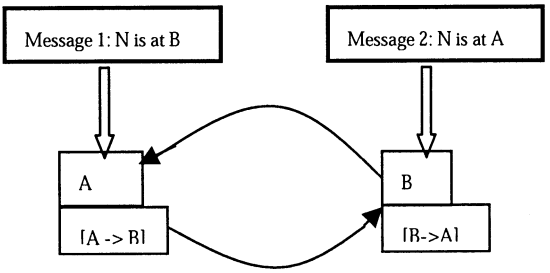

Fig. 5. A message cycle.

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." A second desired property of Mobile IPv6 is 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.

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 Fig. 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 analyse 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 localise the problem we discovered. Each smaller specification also resulted in a more tractable analysis.

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 5. A full grammar and description of the specification language NP is available as a technical report [JaD96].

3.2. 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.

We posit an abstract set of hosts, *HOST*, which has elements that are structureless, uninterpreted identifiers. In what follows, for simplicity we assume there is only one mobile host. We model the entire collection of forwarding pointers stored at nodes as a single function

```
caches: HOST -> HOST
```

with the interpretation that *caches.h* denotes the location that *h* believes the mobile host to be at. (NP uses a dot to denote function application.) 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. (Relational composition in NP is denoted by a semicolon.) The transitive closure *caches+*, defined as

$$\texttt{caches+} = \texttt{caches} \cup \texttt{(caches ; caches)} \cup \texttt{(caches ; caches ; caches)} \cup \ldots$$

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 modeling the protocol this distinction can be ignored.

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

$$\texttt{caches+ \& Id} = \{\}$$

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. Let *MSG* be an abstract set of messages and the following three functions 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; *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:

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

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

3.3. 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 Z's repertoire of relational operators. For example, we saw already that

```
p ; q
```

denotes the composition (join) of relations *p* and *q*. Also,

```
s <: r
```

denotes the restriction of the relation *r* to the pairs whose first element belongs to the set *s*, and

```
r ~
```

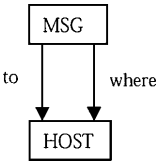

Fig. 6. An entity-relationship diagram.

denotes the transpose of the relation *r*. The expression

```
to ~ ; where
```

thus 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 Fig. 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.

3.4. 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 node, for example, might be modeled as an operation *mh_arrive* that is parameterised by *h*, the fixed node, 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) = [
    /* state variables */
    caches: HOST -> HOST
    caches': HOST -> HOST
    from, to, where: MSG -> HOST
    from', to', where': MSG -> HOST
    router: HOST
    router': HOST
    updates: set MSG
    updates': set MSG
    |
    /* constraints */
    m not in updates
    from' = from U {m -> h}
    to' = to U {m -> router}
    where' = where U {m -> h}
    router' = h
    caches' = caches
    ]
```

In contrast to code, this description of the operation is abstract. For example, parameters, such as *h* and *m*, should not be thought of as arguments to an executable operation, but rather as the free variables in the predicate denoted by the operation's set of constraints. The above specification asserts that some message, not currently in circulation, is created and addressed to the previous location of the mobile host, 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; 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.

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

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

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

```
dom from = updates
```

and

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

(Note that in NP, there is no need for Z's *delta*: the argument list identifies the schema as an operation, and causes two copies of the state components to be included, one primed and one unprimed.) 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 it is not possible to add pairs to *from*, *to*, and *where* while keeping *updates* fixed.

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

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

3.6. 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: $S \leftrightarrow S$, and starts in a state in the set S_0. To prove that every reachable state satisfies some property P, it is enough to show that P holds for every state in S_0, 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*, and

```
Init => P'
```

for the initialisation condition *Init*. For example, to show that the caches never form a cycle, we will have to check assertions like

```
host_move_OK (h: HOST, m: MSG) :: [net | 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 characterises 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 spurious transitions.

This can be a chore in practice. The specifier is forced effectively to give an argument for correctness by characterising the reachable states. Model checkers avoid this by computing the reachable state set automatically. The inductive approach has benefits, however. First, because it is modular, it allows the analysis of a partial specification: we can pick, as here, a complex operation and ignore all others. A model checker, on contrast, requires as input an abstract program that defines the system's behaviour completely.

Second, the inductive approach is more robust. 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.

4. 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

```
host_move_OK (h: HOST, m: MSG) :: [net | 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 claim 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*

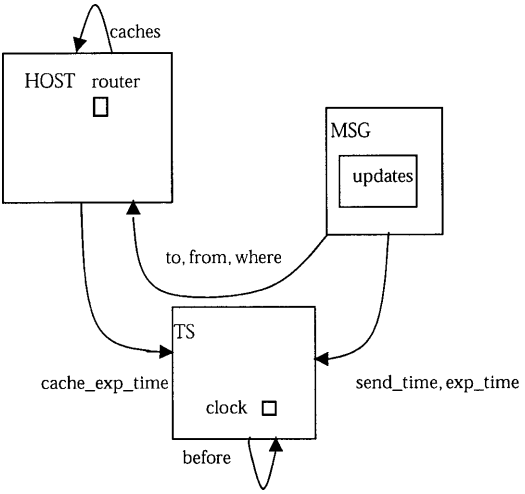

Fig. 7. A diagram corresponding to the declarations of the schema net.

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, we needed just two hosts and one message to find a flaw. 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 [JJD97, Jac98].

5. 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 modelled 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].

5.1. 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. As before, we assume for simplicity that there is only one mobile host. Figure 7 gives a kind of entity-relationship diagram in which set containment is shown in the style of Venn diagrams.

Here is a bird's eye view of the NP specification given in its entirety in Figures 8 and 9. The global abstract state of the network, *net*, contains variables (as before) for holding information about host caches and binding update messages;

```
[HOST, MSG, TS]
net = [
  router: HOST
  caches: HOST -> HOST
  cache_exp_time: HOST -> TS
  updates: set MSG
  to, from, where: MSG -> HOST
  send_time, exp_time: MSG -> TS
  clock: TS
  const before: TS <-> TS
|
  dom to = updates
  dom from = updates
  dom where = updates
  dom send_time = updates
  dom exp_time = updates
  dom cache_exp_time = dom caches
  caches & Id = {}
  (from; from~) & (to; to~) & (send_time; send_time~) <= Id
  (from~; to) & Id = {}
  exp_time <= send_time; before

  /* axiomatize the time ordering as a total order */
  /* (t1, t2) in before means t1 is before t2 */
  before = before+
  before & Id = {}
]
```

Fig. 8. The full specification (Part I): the network state space.

it also models global time. The specification has two operations, *mh_arrive* and *update_arrival*; one invariant property, *acyclic_caches*; and two claims (that each of the operations preserves the invariant), *host_move_OK* and *loc_update_OK*. The first operation, *mh_arrive*, is as before, but takes into consideration time; the second operation, *update_arrival*, describes how host caches are updated when a message arrives at its destination. We will see from our Nitpick analysis that the first claim is valid but the second is not.

5.1.1. The Network

We build upon the specification presented in Section 4 by adding the notion of time into the model. In IPv6, as with many network protocols, any message traveling in the network has an associated 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

```
Init () = [net |
 caches' = {}
 updates' = {}
 ]
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})
  cache_exp_time' <= (cache_exp_time :> before.{clock'})
  caches' <= caches
  updates' = updates U {m}
  send_time' = send_time U {m -> clock}
  exp_time' = exp_time U {m -> t}
  to' = to U {m -> router}
  from' = from U {m -> h}
  where' = where U {m -> h}
]
update_arrival (m:MSG) = [net |
  exp_time.m in before.{clock'}
  clock' in before.{clock}
  caches' <= caches (+) {to.m -> where.m}
  cache_exp_time' <= (cache_exp_time (+) {to.m -> exp_time.m}) :>before.{clock'}
  router' = router
  updates' = updates
  to' = to
  from' = from
  where' = where
  send_time' = send_time
  exp_time' = exp_time
]
acyclic_caches = [net | caches+ & Id = {}]
/* valid */
host_move_OK (h:HOST; m:MSG; t:TS) ::
      [net | acyclic_caches and mh_arrive (h, m, t) => acyclic_caches']
/* invalid */
loc_update_OK (m:MSG) ::
      [net | acyclic_caches and update_arrival (m) => acyclic_caches']
```

Fig. 9. The full specification (Part II): two operations, the cache acyclicity property, and two claims.

synchronised 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 in Fig. 8 declares three basic types: *HOST*, *MSG*, and *TS* (for timestamps).

The first state variable, *router*, denotes the host at which the mobile host is currently docked. The rest of the state space is grouped roughly according to information about host caches, messages, and time.

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 constraint *dom cache_exp_time = dom caches* says that a cache entry is always associated with an expiration time (after which the entry becomes invalid and will be deleted). The constraint *caches & Id = {}* says that there are no self-cycles in a cache.

The variable *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 constraint

```
(from~; to) & Id = {}
```

says that the sender and the receiver of a message cannot be the same. As mentioned, the variable *clock* models the current (global) time. The constant *before* denotes a binary relation on timestamps; the two axioms define it to be an irreflexive, transitively-closed relation.

Let us examine in detail how we express the constraint that the expiration time of a message is always later than its send time. In the subexpression

```
send_time; before
```

we apply the forward composition operator to *send_time* and *before* to obtain a relation 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 that says *m*'s send time is before its expiration time. (Confusingly, the symbol <= is our ASCII representation of subset, not the "less than or equal" operator on timestamps.)

5.1.2. The Operations

Figure 9 contains the specification of the initial condition, two operations, the cache acyclicity property, and two claims that each operation preserves it. We will use Nitpick to see if these claims are true. The operation *mh_arrive* models the docking of the mobile host at a new host. It has 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 expiration time *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. Similarly, the expression

```
clock' in (before.{clock})
```

advances the clock. The assertion

```
caches' <= caches
```

says that the new set of caching hosts 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 behavior of *mh_arrive* is expressed in the next line

```
cache_exp_time' <= (cache_exp_time :> before.{clock'})
```

which updates *cache_exp_time*. Updating *cache_exp_time* involves retaining only the set of cache entries that have neither reached the expiration time associated with them nor been selected for dropping by the local cache replacement policy. The two conditions are captured through the combination of restricting the range of *cache_exp_time* to the set of timestamps that are strictly after the clock's updated time and stating in the specification for *net* the constraint, *dom cache_exp_time* = *dom caches*, which requires the replacement policy to drop stale messages.

Finally, the assertions

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

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* describes what happens when a message, *m*, arrives at its destination. It is similar to *mh_arrive*, except that the receiver definitively modifies the cache entry for the mobile host according to the information in the binding update message. In using the relational override operator (+), the specification forces *caches* to contain the entry *to.m -> where.m*, thereby depriving the host the freedom to choose whether or not to accept the binding update message.

```
...
Variable values are:
before: {T0->T2,T1->T0,T1->T2}
cache_exp_time': {H0->T2,H1->T2}
cache_exp_time: {H1->T2}
caches': {H0->H1,H1->H0}
caches: {H1->H0}
clock': T0
clock: T1
exp_time': {M0->T2}
exp_time: {M0->T2}
from': {M0->H1}
from: {M0->H1}
m: M0
router': H1
router: H1
send_time': {M0->T1}
send_time: {M0->T1}
to': {M0->H0}
to: {M0->H0}
updates': {M0}
updates: {M0}
where': {M0->H1}
where: {M0->H1}
```

Fig. 10. Excerpt of Nitpick output with counterexample to second claim, loc_update_OK.

5.2. 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

```
host_move_OK (h:HOST; m:MSG; t:TS) :: [net | acyclic_caches and
                        mh_arrive (h, m, t) => acyclic_caches']

loc_update_OK (m:MSG, k: set HOST) :: [net | acyclic_caches and
                      update_arrival (m, k) => acyclic_caches']
```

In practice, to check a claim we use Nitpick iteratively by trying different combinations of scopes, usually increasing bounds of different types one or more at a time. We follow this incremental process 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 time bounds.

For all the scopes we tried, Nitpick fails to find any counterexamples in checking the first claim, *host_move_OK*. 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.

In executing Nitpick with the scope $|HOST| = 2$, $|MSG| = 1$, and $|TS| = 3$, we discover a counterexample for the second claim, *loc_update_OK*, in about three seconds. We need at least three timestamps to show the flaw: we need two for each operation to proceed (to update the clock during the transition); we need a third since, with only two, 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.

Let us use the excerpt of Nitpick's output shown in Fig. 10 to help us construct a problematic scenario. 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 $T1 < T0 < T2$, as reflected in the line

```
before: {T0->T2,T1->T0,T1->T2}
```

The lines

```
clock': T0
clock: T1
```

indicate that the time after executing the operation is *T0* and the lines

```
cache_exp_time': {H0->T2,H1->T2}
cache_exp_time: {H1->T2}
```

confirm that the cache entries for both *H0* and *H1* have not yet expired, since the expiration time for both is *T2*. The line

```
caches': {H0->H1,H1->H0}
```

shows 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': H1
router: H1
```

show that the mobile host is docked at router *H1* throughout the operation. Before the operation, that is, before the binding update message is received by *H1*, the cache entry in *H1* points to *H0*, as shown in the line

```
caches: {H1->H0}
```

which means that *H1* thinks that the mobile host resides at *H0*. During the operation, router *H0*, as shown in the line

```
to: {M0->H0}
```

receives the message *M0*, given by

```
m: M0
```

from *H1* informing *H0* that the mobile host is currently at *H1*, indicated by the line

```
where: {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 *H1* 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.

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

Our analysis was originally performed using an early version of Nitpick in 1996. Although sufficient in this case – it indeed revealed interesting properties of the protocol – the tool did not scale well. In fact, it was not able even to analyse cases involving three messages and three hosts.

Since then, we have devised new analysis techniques. A more recent version of Nitpick performs much better; results are shown in the table below. This version uses a new technique for analysis (described in [Jac98]) based on boolean satisfaction: the problem is compiled into a boolean formula and then fed to a solver. In the implementation used here, two solvers are available, and the user may switch between them. DP is a Davis-Putnam solver coded in Java, and not especially efficient; for example, it generates a fresh copy of the formula every time a boolean variable is split. WSAT is our own Java version of the Selman,

Levesque, and Mitchell WalkSAT solver [SLM92], which can scale to larger formulas, but works by local search and is thus not complete.

Scope	*Nitpick/DP (s)*	*Nitpick/WSAT (s)*
2	0	?
3	3	3
4	56	8
5	?	19

The table gives the performance of the two different backends of Nitpick for different scopes for finding a counterexample to *loc_update_OK*. A scope of k means k HOSTs, k MSGs, and k TSs. All times are in seconds. A question mark means that the check did not terminate in less than two minutes. For WalkSAT on a scope of two, this is because there is no counterexample, but since it performs a randomised hill-climbing search, it will search forever.

5.3. 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. In the extended specification (not shown here), we model a set of *valid* messages and simply update this variable appropriately in both the *mh_arrive* and *update_arrival* operations. (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 up to and including 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.

6. Discussion

We carried out this case study 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 reemphasise, giving specific examples, the importance of modularity and abstraction in writing formal specifications. The most important lesson we learned was that partial specifications play an invaluable role in understanding and debugging designs.

6.1. Expressiveness of NP

NP has no quantifiers. Tarski and Givant showed that quantifiers can be eliminated from formulas with subformulas each with at most three free variables [TaG87]. Since this class covers all formulas we have needed so far in practice, NP is in principle expressive enough as a practical specification language. The omission of quantifiers, however, 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

```
∀m1, m2 : MSG.
from.m1 = from.m2 ∧ to.m1 = to.m2 ∧
                          send_time.m1 = send_time.m2 ⇒ m1 = m2
```

In contrast, NP forces us to write:

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

(Partly in reaction to this syntactic limitation, NP's successor language, Alloy, has quantifiers [Jac00].)

There are other recurrent NP idioms which, once familiar to the specifier, make constraints easy to express succinctly. 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.

6.2. 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.

Like most model checkers, Nitpick's performance is somewhat unpredictable. A small change in the specification can have a dramatic effect on the behavior

of the solver. This problem arises whatever backend solver is used. The analysis problem is inherently intractable in the worst case, so obtaining good performance always involves some use of heuristics, such as picking a good variable ordering.

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. Nitpick's quick turnaround (for small scopes) and its point-and-click user interface make specifying a more compelling and enjoyable activity.

6.3. How Nitpick Influenced the Way the Specification is Written

Were we to specify and analyse this protocol using a model checker, we would more naturally focus on the protocol's control aspects and hence on the state transitions and properties true of sequences of states (or sequences of state transitions). Nitpick, however, is particularly good at checking state invariants and thus draws the specifier's attention to global properties of states. Capturing a global state property such as acyclicity is easy and natural to do in Nitpick/NP, and would be more awkward to describe and check using a typical model checker.

Even in the domain of network protocols, by careful and tailored application of two standard principles – modularity and abstraction – it is possible to obtain a state-based specification that is both faithful to the problem and analysable.

Modularity. The modularisation 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 wrote 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 tunnelling of packets: Encapsulation is the process of wrapping a network header to the packet which contains the routing information. Tunnelling is simply the path followed by a packet while it is encapsulated. Encapsulation and tunnelling 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 tunnelled to the home agent. Our ability to abstract from modeling encapsulation and tunnelling 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/tunnelled. Essentially we take into account every valid message that is sent across the network and do not distinguish between encapsulated/tunnelled packets and non-encapsulated/non-tunnelled 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.

7. Conclusions

A lightweight formal method [JaW96] enables partial specification and automatic analysis by sacrificing breadth of coverage and expressive power. Our study of IPv6 illustrates, more generally, our attitude toward formal methods. Formality is not an end in itself, and indeed we recognise 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 formalisation. 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 analyse such aspects in isolation, expending only as much effort as is needed to understand the design's subtleties and expose its deficiencies.

Many factors determine which counterexample will be generated if more than one exists. The new version of Nitpick will not, in general, produce the same counterexamples as the old one, although the counterexample shown is Section 5.2 is in fact generated first by both new and old versions. In fact, an element of luck was involved: there are less revealing counterexamples that do not suggest the deeper problem discussed in this paper, but rather just show incompleteness in the specification. Our approach can thus be viewed as a kind of "directed simulation". The analysis does not require a full specification, or an

invariant sufficiently strong to prove induction. Rather, given an operation and an invariant, it executes the operation "backwards" to find pre-states that would lead to a violation of the invariant. If these reveal a problem with the artifact itself, the specifier need go no further; if they merely expose an incompleteness in the description, the specifier can elaborate it and repeat the analysis. A major advantage of declarative specification is this ability to analyse partial descriptions. A traditional model checker, in contrast, would require a detailed, deterministic (and operational) specification of the transitions.

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.

In the four years since the work described in this paper was performed, we have made considerable advances. Our new language, Alloy [Jac00], is far more expressive, and allows a more natural specification (particularly because it allows quantifiers). Alloy's analyser, Alcoa, which uses technology based on the ideas in [Jac98], can find the flaw described in this paper using the new specification in 2 seconds (in its current version—February 2000—running on a modestly equipped PC). For a version of the specification with the error corrected and the invariant elaborated to characterise reachability, Alcoa can exhaust a scope of 5 (approximately 10^{50} states) in less than a minute.

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 IP 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, in part by the National Science Foundation under Grant No. CCR-9523972, and in part by the National Security Agency under Grant No. MDA904-99-C-5020. The US Government is authorised 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

[DaP60] Davis, M. and Putnam, H.: A computing procedure for quantification theory. *Journal of the ACM*, Vol. 7, pp. 202–215, 1960.

[HOLZ] HOL-Z.: *http://www.first.gmd.de/~santen/HOL-Z/*.

[HOS97] Hewitt, M.A., O'Halloran, C.M. and Sennett, C.T.: "Experiences with PIZA, an Animator for Z", *Proc. of ZUM'97*, LNCS 1212, 1997, pp, 37–51.

[Jac98] Jackson, D.: An Intermediate Design Language and its Analysis. *Proceedings of ACM SIGSOFT Foundations of Software Engineering*, Orlando, Florida, 1998.

[Jac00] Jackson, D.: *Alloy: A Lightweight Object Modelling Notation*. MIT Laboratory for Computer Science, Technical Report 797, February 2000.

[JaD96] Jackson, D. and Damon, C.A.: *Nitpick Reference Manual*. CMU-CS-96-109. School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, January 1996.

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

[JaW96] Jackson, D. and Wing, J.: "Lightweight Formal Methods", *IEEE Computer*, April 1996, pp. 21–22.

[Joh95] Johnson, D.B.: "Scalable Support for Transparent Mobile Host Internetworking", *Wireless Network*, Vol. 1, October 1995, pp. 311–321.

[JoP96] Johnson, D.B. and Perkins, C.: "Mobility Support in IPv6", Mobile IP Working Group INTERNET-DRAFT, June 1996.

[Lam78] Lamport, L.: "Time, Clocks, and the Ordering of Events in a Distributed System", *CACM*, Volume 21, Number 7, July 1978, pp. 558–565.

[MSH98] Morrey, I., Siddiqi, J., Hibberd, R. and Buckberry, G.: "A Toolset to Support the Construction and Animation of Formal Specifications", *Journal of Systems and Software*, vol 41, no. 3 1998, pp. 147–160.

[Ng97] Ng, Y.-C.: "A Nitpick Specification of Mobile Ipv6", Computer Science Department, Carnegie Mellon University, Pittsburgh, PA, Senior Honors Thesis, May 1997.

[Pos80] Postel, J.B.: "Internetworking Protocol Approaches", *IEEE Transactions on Communications*, April 1980, Vol. 28, pp. 604–611.

[Spi92] Spivey, J.M.: *The Z Notation: A Reference Manual*, Second edition, Prentice-Hall, 1992.

[Saa97] Saaltink, M.: "The Z/EVES System", *Proc. of ZUM'97*, LNCS 1212, 1997, pp. 72–85.

[SLM92] Selman, B., Levesque, H. and Mitchell, D.: A new method for solving hard satisfiability problems. *Proc. 10th National Conference on Artificial Intelligence (AAAI-92)*, San Jose, CA, July 1992.

[TaG87] Tarski, A. and Givant, S.: *A Formalization of Set Theory Without Variables*. American Mathematical Society Colloquium Publications, Volume 41, 1987.

Received September 1997
Accepted in revised form January 2000 by I J Mayes