Formal Verification of Standards for Distance Vector Routing Protocols[1]

KARTHIKEYAN BHARGAVAN, DAVOR OBRADOVIC, and CARL A. GUNTER
Department of Computer and Information Science
University of Pennsylvania

We show how to use an interactive theorem prover, HOL, together with a model checker, SPIN, to prove key properties of distance vector routing protocols. We do three case studies: correctness of the RIP standard, a sharp real-time bound on RIP stability, and preservation of loop-freedom in AODV, a distance vector protocol for wireless networks. We develop verification techniques suited to routing protocols generally. These case studies show significant benefits from automated support in reduced verification workload and assistance in finding new insights and gaps for standard specifications.

Categories and Subject Descriptors: C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Protocol Verification*; *Routing Protocols*; C.2.6 [**Computer-Communication Networks**]: Internetworking—*Routers*; *Standards*; D.2.4 [**Software Engineering**]: Software/Program Verification—*Formal Methods*; *Model Checking*; *Correctness Proofs*; F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning about Programs—*Mechanical Verification*; F.4.1 [**Mathematical Logic and Formal Languages**]: Mathematical Logic—*Mechanical Theorem Proving*

General Terms: Design,Verification,Theory,Reliability

Additional Key Words and Phrases: Formal Verification, Routing Protocols, Network Standards, Distance Vector Routing, RIP, AODV, Interactive Theorem Proving, HOL, Model Checking, SPIN

1. INTRODUCTION

The aim of this paper is to study how methods of automated reasoning can be used to prove properties of network routing protocols. We carry out three case studies based on distance vector routing. In each such study we provide a proof that is automated and formal in the sense that a computer assisted the construction and checking of the proof using formal mathematical logic. We are able to show that automated verification of key properties is feasible based on the IETF standard or draft specifications, and that efforts to achieve automated proofs can aid the discovery of useful properties and direct attention to potentially troublesome boundary cases. Automated proofs can effectively supplement other means of assurance like manual mathematical proofs and automated testing by identifying unexpected boundary cases and checking large numbers of cases without the need for human insight.

1.1 The Case Studies

The first case study proves the correctness of the asynchronous distributed Bellman-Ford protocol as specified in the IETF RIP standard [Hendrick 1988; Malkin 1994]. The classic proof of a 'pure' form of the protocol is given in [Bertsekas and Gallager 1991]. Our result covers additional features included in the standard to improve real-time response times (*e.g.* split horizons and poisoned reverse). These features

[1]To appear in the Journal of the ACM, 2002

add additional cases to be considered in the proof, but the automated support reduces the impact of this complexity. Adding these extensions makes the theory better match the standard and hence also its implementations. Our proof also uses a different technique from the one in [Bertsekas and Gallager 1991] and provides additional properties about network stability.

Our second case study provides a sharp real-time convergence bound on RIP in terms of the radius of the network around its nodes. In the worst case, the Bellman-Ford protocol has a convergence time as bad as the number of nodes in the network. However, if the maximum number of links any source needs to traverse to reach a destination is k (the radius around the destination) and there are no link changes, then RIP will converge in k timeout intervals for this destination. It is easy to see that convergence occurs within $2 \cdot (k-1)$ intervals, but the proof of the sharp bound of k is complicated by the number of cases that need to be checked: we show how to use automated support to do this verification, based on the approach developed in the previous case study. Thus, if a network has a maximum radius of 5 for each of its destinations, then it will converge in at most 5 intervals, even if the network has 100 nodes. Assuming the timing intervals in the RIP standard, such a network will converge within 15 minutes if there are no link changes. Our main goal is to show how automated support can cover real-time properties of routing protocols.

Our third case study is intended to explore how automated support can assist *new* protocol development efforts. We consider a distance vector routing protocol arising from work at MANET, the IETF work group for mobile ad hoc networks. The specific choice is the Ad-Hoc On-Demand Distance Vector (AODV) protocol of Perkins and Royer [1999], as specified in the second version of the IETF Internet Draft [Perkins and Royer 1998]. This protocol uses sequence numbers to protect against the formation of loops, a widely noted shortcoming of RIP. A sketch of a proof that loops cannot form is given in [Perkins and Royer 1999]. We show how to tighten some of the AODV conditions and derive this property from a general invariant for the paths formed by AODV. We use this invariant to analyze some conditions concerning failures that are not fully specified in [Perkins and Royer 1998] but could affect preservation of the key invariant if not treated properly. Issues from our analysis and that of others enabled these problems to be addressed in subsequent AODV drafts. Our primary conclusion is that the automated verification tools can aid analysis of emerging protocol specifications on acceptable scales of effort and 'time-to-market'.

1.2 Verification of Networking Standards

Automated logical reasoning about computer systems, widely known as *formal methods*, has been successful in a number of domains. Proving properties of computer instruction sets is perhaps the most established application and several major hardware vendors have programs to do modeling and verification of their systems using formal methods. Another area of success concerns safety critical devices. For instance, [Heitmeyer et al. 1998] studies invariants of a weapons control panel for submarines modeled from the contractor design documents. The study led to a good simulator for the panel and located some serious safety violations. The application of formal methods to software has been a slower process, but there has been noteworthy success with avionic systems, air traffic control systems, and others. One key impediment in applying formal methods to non-safety-critical systems concerns the existence of a specification of the software system: it is necessary to know what the software is intended to *satisfy* before a verification is possible. For many software systems, no technical specification exists, so the verification of documented properties means checking invariants from inline code comments or examples from user manuals.

An exception to this lack of documentation is software in the telecommunications area, where researchers have a penchant for detailed technical specifications. RIP offers a case study in motivation. Early implementations of distance vector routing were incompatible, so all of the routers running RIP in a domain needed to use the same implementation. Users and implementers were led to correct this problem by providing a specification that would define precise protocols and packet formats. We find below that the resulting standard [Hendrick 1988; Malkin 1994] is precise enough to support, without significant supplementation, a detailed proof of correctness in terms of invariants referenced in the specification. The proved properties are guaranteed to hold of any conforming implementation and of any network of conforming routers. RIP is perhaps better than the average in this respect, since (1) the standard seeks to bind itself closely to its underlying theory, (2) distance vector routing is simpler than some alternative routing approaches, and (3) at this stage, RIP is a highly seasoned standard whose shortcomings have been identified through substantial experience. This is not to say that RIP was already verified by its referenced theory. There are substantial gaps between [Hendrick 1988; Malkin 1994] and the asynchronous distributed protocol proved correct in [Bertsekas and Gallager 1991]: the algorithm is different in several non-trivial ways, the model is different, and the state maintained is different. Our analysis narrows this gap and extends the results of the theory as applied to the standard version of the protocol.

It is natural to expect that newer protocols, possibly specified in a sequence of draft standards, will have more gaps and will be more likely to evolve. Useful application of formal methods to such projects must 'track' this instability, locating errors or gaps quickly and leveraging other activities like revision of the draft standard and the development of simulations and implementations. To test this agility for our tools and methods we extended our analysis of RIP to newer applications of distance vector routing in the emerging area of mobile ad hoc networks. Ad hoc networks are networks formed from mobile computers without the use of a centralized authority. A variety of protocols are under development for such networks [Royer and Toh 1999], including many based on distance vector routing [Perkins and Bhagwat 1994; Chiang 1997; Murthy and Garcia-Luna-Aceves 1996; Perkins and Royer 1999]. Requirements for a routing protocol for ad hoc networks are quite different from those of other kinds of networks because of considerations like highly variable connectivity and low bandwidth links. Given the rapid rate of evolution in this area and the sheer number of new ideas, it seems like an appropriate area as a test case for formal methods as part of a protocol design effort.

1.3 Verification Attributes of Routing Protocols

There have been a variety of successful studies of communication protocols. For instance, [Mitchell et al. 1998] provides a proof of some key properties of SSL 3.0 handshake protocol [Freier et al. 1996]. However, most of the studies to date have focused on *endpoint* protocols like SSL using models that involve two or three processes (representing the endpoints and an adversary, for instance). Studies of routing protocols must have a different flavor since a proof that works for two or three routers is not interesting unless it can be generalized. Routing protocols generally have the following attributes, which influence the way formal verification techniques can be applied:

(1) An (essentially) unbounded number of replicated, simple processes execute concurrently.
(2) Dynamic connectivity is assumed and fault tolerance is required.
(3) Processes are reactive systems with a discrete interface of modest complexity.
(4) Real time is important and many actions are carried out with some timeout limit or in response to a timeout.

Most routing protocols have other attributes such as latencies of information flow (limiting, for example, the feasibility of a global concept of time) and the need to protect network resources. These attributes sometimes make the protocols more complex. For instance, the asynchronous version of the Bellman-Ford protocol is much harder to prove correct than the synchronous version [Bertsekas and Gallager 1991], and the RIP standard is still harder to prove correct because of the addition of complicating optimizations intended to reduce latencies.

In this paper we verify protocols using tools that are very general (HOL) or tuned for the verification of communication protocols (SPIN). The tools will be described in Section 2, and an overview of routing protocols including RIP and AODV is provided in Section 3. The rest of the paper consists of the three case studies. We describe a proof of the correctness of RIP in Section 4, proof of a sharp real-time bound on convergence of RIP in Section 5, and proof of path invariants for AODV in Section 6. We offer some conclusions and statistics in the final section.

2. APPROACHES TO FORMAL VERIFICATION

For centuries mathematicians have worked on techniques for verifying that algorithms have the properties they are expected to have. Instantiations of algorithms in standardized protocols and their implementation on computers is a more recent phenomenon. While traditional mathematical techniques are extremely valuable in this new context, there are some noteworthy changes. First, the implementations of protocols can be automatically tested using computers. This improves the likelihood that the protocol and its implementation achieve the desired results, even when a full mathematical proof is missing. Second, the complexity of the protocol, and especially its implementation, make mathematical proof of correctness difficult. Given the second change, it is tempting to skip mathematical proofs and rely on the advantages brought by the first change. Since testing does not cover all cases in the way a mathematical proof does, this reduces the level of assurance that the protocol and implementation have the desired properties. A happy alternative is one in which automated analysis techniques enable mathematical verification.

Automated proofs of protocols fall into three general categories. The first can be seen as an extension of testing wherein automated support is used to create tests that, in essence, include all possible cases, thus providing a proof of correctness.

Automated tools supporting this approach are often called *model checkers.* The second can be seen as a formalization of mathematics wherein logic is used to characterize mathematical reasoning, and automated formal support is used to aid the creation and checking of proofs. Automated tools supporting this approach are often called *theorem provers.* A third category is one in which exhaustive testing can be achieved in limited cases. This can improve testing by enabling better coverage, or, if supplemented by suitable mathematical arguments, it can even provide a complete proof of correctness. This third category can be supported by a combination of 'manual' mathematical reasoning, automated model checking, and automated theorem proving.

Computer protocols have long been the targets of verification efforts. Protocol design often introduces subtle bugs that remain hidden in all but a few *runs* of the protocol, but might lead to serious operational failures. In this section, we discuss the complexities involved in verifying network protocols and propose automated tool support for this task. As an example, we consider a simple protocol for leader-election in a network. A variant of this protocol is used for discovering spanning trees in an extended LAN [Perlman 1985; 1992].

The network consists of n connected nodes. Each node has a unique integer *id.* The node with the least id is called the *leader.* The aim of the protocol is for every node to discover the id of the leader. To accomplish this, each node maintains a *leader-id*: its own estimate of who the leader is, based on the information it has so far. Initially, the node believes itself to be the leader. Every p seconds, each node sends an advertisement containing its leader-id to all its neighbors. On receiving such an advertisement, a node updates its leader-id if it has received a lower id in the message.

The above protocol involves n processes that react to incoming messages. The state of the system consists of the (integer) leader-ids at each process; the only events that can occur are message transmissions initiated by the processes themselves. However, due to the asynchronous nature of the processes, the message transmissions could occur in any order. This means that in any period of p seconds, there could be more than $n!$ possible sequences of events to which the system must react. It is easy to see that manual enumeration of the potential event or state sequences becomes impossible as n is increased. For more complex protocols, manually tracing the path of the protocol for even a single sample trace becomes tedious and error-prone. Automated support for this kind of analysis is clearly required.

A well-known design tool for protocol analysis is simulation. However, to simulate the election protocol, we would first have to fix the network size and topology, and then specify the length of the simulation. Finally, we can run the protocol and look at its trace for a given initial state and a single sequence of events. This simulation process, although informative, does not provide a complete verification. A verification should provide guarantees about the behavior of the protocol on all networks, over all lengths of time, under all possible initial states and for every sequence of events that can occur.

We discuss two automated tools that can help provide these guarantees. First, we describe the model checker SPIN, which can be used to simulate and possibly

Table I. Leader Election in Promela

```
#define NODES 3
#define BUF_SIZE 1
chan input[NODES] = [BUF_SIZE] of {int};
chan broadcast = [0] of {int,int};
int leader_id[NODES];

proctype Node (int me; int myid){
    int advert;
    leader_id[me] = myid;
    do
    :: input[me]?advert ->
              if
              :: advert < leader_id[me] ->
                     leader_id[me] = advert
              :: else -> skip
              fi
    :: true -> broadcast!me,leader_id[me]
    od
}
```

verify the protocol for a given network (and initial state). We then describe the interactive theorem prover HOL, which, with more manual effort, can be used to verify general mathematical properties of the protocol in an arbitrary network.

2.1 Model Checking Using SPIN

The SPIN system (`netlib.bell-labs.com/netlib/spin/whatispin.html`) has been widely used to verify communication protocols. The SPIN system has three main components: (1) the Promela protocol specification language, (2) a protocol simulator that can perform random and guided simulations, and (3) a model checker that performs an exhaustive state-space search to verify that a property holds under all possible simulations of the system [Holzmann 1991; 1997].

To verify the leader-election protocol using SPIN, we first model the protocol in Promela. A Promela model consists of processes that communicate by message-passing along buffered channels. Processes can modify local and global state as a result of an event. The Promela process modeling the leader-election protocol at a single node is as given in Table 2.1. We then hard-code a network into the broadcast mechanism and simulate the protocol using SPIN. SPIN simulates the behavior of the protocol over a random sequence of events. Viewing the values of the leader-ids over the period of the simulation provides valuable debugging information as well as intuitions about possible invariants of the system.

Finally, we use the SPIN verifier to prove that the election protocol succeeds in a 3-node network. This involves specifying the correctness property in Linear Temporal Logic (LTL) [Manna and Pnueli 1991]. In our case, the specification simply insists that the leader-id at each node eventually stabilizes at the correct id. The verifier then carries out an exhaustive search to ensure that the property is true for every possible simulation of the system. If it fails for any allowed event

Table II. State Update Function

```
function Update (state, sender, receiver, mesg, node):int =
        if node = receiver then
           if mesg < state(receiver)
           then mesg else state(receiver)
        else state(node)
```

sequence, the verifier indicates the failure along with the counter-example, which can be subsequently re-simulated to discover a possible bug.

2.2 Interactive Theorem Proving Using HOL

The HOL Theorem Proving System (`www.cl.cam.ac.uk/Research/HVG/HOL`) is a widely used general-purpose verification environment. The main components of the HOL system are (1) a functional programming language used for specifying functions, (2) Higher-Order Logic used to specify properties about functions, and (3) a proof assistant that allows the user to construct proofs of such properties by using inbuilt and user-defined proof techniques [Gordon and Melham 1993]. Both the programming model and the proof environment are very general, capable of proving any mathematical theorem. Designing the proof strategy is the user's responsibility.

In order to model the leader-election protocol in HOL, we need to model processes and message-passing in a functional framework. We take our cue from the reactive nature of the protocol. The input to the protocol is a potentially infinite sequence of messages. Each process can then be represented by an *update* function that takes a message as input and describes how the process state is modified. In our case, the process state consists of an integer representing the current leader computed by the node. The state of the entire system is then updated in accordance with the function at each `node`, as shown in Table 2.2. Note that the generality of the programming platform allows us to define the protocol for an arbitrary network in a uniform way.

We then specify the property that we desire from the protocol as a theorem that we wish to prove in HOL.

THEOREM 2.1. *Eventually, every node's leader-id is the minimum of all the node ids in the network.*

In order to prove this property, we prove three lemmas, all of which can be easily encoded in Higher-Order Logic.

LEMMA 2.2. *At each node, the leader-id can only decrease over time.*

LEMMA 2.3. *If the state of the network is unchanged by a message from node n_1 to node n_2 as well as a message from n_2 to n_1, the leader-ids at n_1 and n_2 must be the same.*

LEMMA 2.4. *Once a node's leader-id becomes correct, it stays correct.*

Finally, we construct a proof of the desired theorem. The proof assistant organizes the proof and ensures that the proofs are complete and bug-free. We first prove the lemmas by case analysis on the states and the possible messages at each point in time. Then, Lemmas 2.2 and 2.3 are used to prove that the state of the network must 'progress' until all the nodes have the same leader-id. Moreover, since the leader node's leader-id never changes (Lemma 2.4), all nodes must end up with the correct leader-id. These proofs are carried out in a simple deductive style managed by the proof assistant.

The above proof is just one of many different proofs that could be developed in the HOL system. For example, if instead of correctness, we were interested in proving how long the protocol takes to elect a leader, we could prove the following lemma. Recall that p is the interval for advertisements.

Lemma 2.5. *If all nodes within a distance k of the leader have the correct leader-id after t seconds, then all nodes within a distance $(k + 1)$ will have the correct leader-id within $t + p$ seconds.*

In conjunction with Lemma 2.4 this enables an inductive proof of Theorem 2.1.

2.3 Model Checking Vs Interactive Theorem Proving

We have described how two systems can address a common protocol verification problem. The two systems clearly have different pay-offs. SPIN offers comprehensive infrastructure for easily modeling and simulating communication protocols and has fixed verification strategies for that domain. On the other hand, HOL offers a more powerful mathematical infrastructure, allowing the user to develop more general proofs. SPIN verifications are generally bound by *memory* and *expressiveness.* HOL verifications are bound by *programmer-months.*

Our technique is to code the protocol first in SPIN and use HOL to address limits in the expressiveness of SPIN. This is achieved by using HOL to prove *abstractions*, showing properties like: if property P holds for two routers, then it will hold for arbitrarily many routers. Or: advertisements of distances can be assumed to be equal to k or $k + 1$. In addition, we use the abstraction proofs in HOL to reduce the memory demands of SPIN proofs while ensuring that the SPIN implementation properly reflects the standard. We give examples of these trade-offs in the case studies and summarize with some statistical data in the conclusions.

3. DISTANCE VECTOR ROUTING

An internetwork can be viewed as a bipartite graph consisting of nodes representing *routers* and *networks*, and edges representing *interfaces.* A host attached to a network sends a packet with a destination network address to a router on its network. This router cooperates with other routers to determine a path for moving the packet toward its destination. A *routing protocol* is an algorithm used by routers to determine such a path. There are many types of internetworks. The connected Internet operates globally, mainly over wired links. Other kinds of internetworks, like ad hoc networks of mobile routers on radio links are a topic of current investigation. In this section, we provide some general background on routing in these two contexts, then provide background on the two protocols on which this paper is focused.

3.1 Routing in the Internet

The Internet is broadly organized into collections of networks called *Autonomous Systems (AS's)*; an AS may, for instance, be the internetwork of a company, a university, or an Internet Service Provider (ISP). Routing protocols that are used between AS's are called *Exterior Gateway Protocols (EGP's)*, while those that run within the AS's are called *Interior Gateway Protocols (IGP's).* IGPs fall into two categories: distance vector routing and link state routing. The principal EGP is the Border Gateway Protocol (BGP), which is similar to a distance vector routing protocol.

Distance-vector protocols were among the first to be used in the Internet. In such protocols, each router maintains, for each destination, the name of an adjacent router that is (thought to be) one 'hop' closer to the destination and (what is thought to be) the number of hops to reach the destination. This information is periodically *advertised* to adjacent routers, and *updated* to take account of information from the advertisements of adjacent routers. The best-known protocol of this kind is RIP, which is still widely used because of its early inclusion in Unix operating systems. RIP is described in a series of IETF RFC's [Hendrick 1988; Malkin 1993; 1994]. The *Enhanced Interior Gateway Routing Protocol (EIGRP)* (`www.cisco.com/warp/public/103/1.html`) is another distance-vector protocol; it is proprietary to Cisco, a major router vendor. The advantage of distance-vector routing protocols is their simplicity. RIP is easy to implement correctly, and the protocol works acceptably well on smaller networks. However, since the network nodes do not maintain a complete view of the network topology, there are limits to how much they can know, and hence take advantage of, about the available paths to a destination. In particular, the information available in RIP is so minimal that the protocol is unable to avoid slow convergence to correct routes when the internetwork is partitioned by failures.

Link state protocols are based on the idea that each router advertises the state of its links to other routers. As this information flows into a given router, it is used to create a map of the complete topology of the internetwork (that is, the collection of networks covered by the protocol, such as those of a given AS). This information is used to calculate complete routes and determine the correct next hop for moving a packet toward its destination. The most widely used link state protocol is the IETF Open Shortest Path First (OSPF) [Moy 1994]. Another important link state protocol is ISO's IS-IS [ISO 8473 1990]. While providing routers with global link information is useful in determining good routes, there is significant complexity involved in the sub-protocol that propagates the link states. OSPF, for instance, is one of the most complex of all RFCs.

BGP [Rekhter and Li 1995] is the dominant routing protocol between AS's in the Internet. It is a kind of distance-vector protocol in which advertisements describe complete routes (rather than just hop count) and the selection of a best route by a router for an AS is a function of both the policies of the AS and the best route as determined by its neighbors. That is, BGP allows distance metrics based on hop count to be overridden by policy-based metrics. This flexibility leads to potential short-comings. In particular, Varadhan et al. [1996] demonstrated circumstances in which routes to a given destination *oscillate*. Such behavior is undesirable in routing protocols. The extent to which it is a potential problem for BGP on the Internet is not well understood. Griffin and Wilfong [1999] demonstrated that even if the BGP topology of the Internet were known, it would not be feasible in principle to decide whether it might display oscillations. Instead, the focus has been on devising extensions and restrictions of BGP to guarantee convergence. Griffin and Wilfong [2000] devised a sufficient condition that guarantees the absence of permanent oscillations in their BGP model and used it to design a convergent extension of the protocol. Obradovic [2002] refined that result by extending the model with real-time attributes and establishing bounds on convergence time. Gao and Rexford [2000] proposed a way to use the provider-customer hierarchy of the Internet to configure BGP routers in a way that guarantees convergence. Deeper understanding of the convergence properties of BGP is likely to be a significant area of investigation over the next few years.

3.2 Routing in Ad Hoc Networks

Routing protocols like RIP make many assumptions about what is reasonable for the network on which they provide routing. For instance, it is assumed to be acceptable to exchange routing information periodically and maintain a route for each destination. These assumptions are justified by the nature of the elements of the internetwork, which consist principally of high-bandwidth, reliable links between capable routing elements serving a stable family of hosts. Consider, by contrast, a collection of mobile computers being used in an application like disaster relief where a wired infrastructure may be unavailable. Links between devices will be low-bandwidth and unreliable. Connectivity will be determined by signal strengths and the link technology (which will be sensitive to noise and obstructions), so the mobility of the nodes may cause connectivity to be extremely variable. Indeed, links to neighboring nodes may change every few minutes or seconds. On the other hand, such a network may not need complete connectivity between each pair of nodes at all times. In a disaster relief situation, it may be the case that only a few mobiles need to communicate, rather than every pair. Low bandwidth, unreliability, and rapidly changing connectivity can therefore be balanced against a potentially modest demand for end-to-end communication links by the use of *on-demand* routing. That is, routes can be determined when they are needed, thus potentially reducing the overhead of routing control messages.

Because of its simplicity, distance vector routing is a natural choice for routing in ad hoc networks. AODV [Perkins and Royer 1999; 1998] provides an instantiation of an on-demand form of distance vector routing that aims to keep control messages to a minimum. As mentioned earlier, there are a variety of other approaches to routing in ad hoc networks based on other strategies. These schemes are all grossly similar in complexity at the current time. AODV is more complex than RIP, not only because of the on-demand requirement, but also because of state added to the protocol to protect against loop-formation. These features will be our primary focus in analyzing the AODV protocol.

3.3 Routing Information Protocol (RIP)

The RIP protocol specification is given in [Hendrick 1988; Malkin 1994] and a good exposition can be found in [Huitema 1995]. This subsection gives a brief description of the protocol. Pseudo-code is given in Appendix A. Our analysis is for version 2 of the RIP Internet Standard, but also applies to version 1.

Each router running RIP maintains a routing table. The table contains one entry per destination, representing the current best route to the destination. Routers periodically *advertise* their routing tables to their neighbors. Upon receiving an advertisement, the router checks whether any of the advertised routes can be used to improve current routes. Whenever this is the case, the router updates its current route to go through the advertising neighbor.

Routes are compared exclusively by their length, measured in the number of *hops* (i.e. the number of routers on the route). A routing table entry corresponding to a destination d contains the following attributes:

hops:	number of hops to d.
nextRouter:	the first router along the route to d (the one that advertised the best route so far).
nextIface:	the interface through which the advertisement from nextRouter was received. This interface uniquely identifies the next network along the route and will be used to forward packets addressed to d.

The value of hops must be an integer between 1 and 16, where 16 has the meaning of *infinity*—a destination with hops attribute set to 16 is considered to be unreachable. RIP is not appropriate for AS's that contain a router and a destination network that are more than 15 hops apart from each other. The objective behind a relatively low upper bound on the route length is faster convergence. RIP exhibits a phenomenon called *counting to infinity*, discussed in [Hendrick 1988], which permits a worst-case loop persistence time, and therefore convergence time, proportional to the maximum allowed route length.

A router advertises its routes by broadcasting RIP packets to all of its neighbors. A RIP packet contains a list of (destination, hops)-pairs. A receiving router compares its current metric for destination to hops + 1, which is the metric of the alternative route, and updates its routing entry for the destination if the alternative route is shorter. There is one exception to this rule: if the advertising router is the nextRouter in the table of the receiving router, then the receiver adopts the alternative route regardless of its metric.

Normally, a RIP packet contains information that matches the advertising router's own routing table. This rule has an exception too, which is designed to prevent creation of loops between pairs of routers. The exception essentially prohibits advertising routes on the interfaces through which they were learned. Simply failing to advertise routes to the given destination over this interface is called a *split horizon*. A more proactive approach is to advertise what is called a *poisoned reverse* over this interface. Assume that a router r learns a route through an interface i. Whenever r advertises that route back through the interface i, the poisoned reverse advertisement sets hops to 16 (infinity). A detailed discussion of these two optimizations can be found in [Hendrick 1988].

Each routing table entry has a timer expire associated with it. Every time an entry is updated (or created), expire is re-set to 180 seconds. Routers try to advertise every 30 seconds, but due to network failures and congestion some advertisements may not get through. If a route has not been refreshed for 180 seconds, the destination is marked as unreachable and a special garbageCollect timer is set to 120 seconds. If this timer expires before the entry is updated, the route is expunged from the table.

3.4 Ad-Hoc On-Demand Distance Vector Protocol (AODV)

The AODV routing protocol is specified in a series of Internet Drafts submitted to the MANET working group (`www.ietf.org/html.charters/manet-charter.html`) at the IETF. An introduction to the protocol is given in [Perkins and Royer 1999]. This subsection describes the AODV routing protocol as specified in the version 2 Internet draft [Perkins and Royer 1998]. Pseudo-code for the protocol is given in Appendices B, C.

In AODV, a route to a destination d contains the following fields:

next_d:	Next node on a path to d.
hops_d:	Distance from d, measured in the number of nodes (hops) that need to be traversed to reach d.
seqno_d:	Last recorded *sequence number* for d.
$\mathsf{lifetime}_d$:	Remaining time before route expiration.

The purpose of sequence numbers is to track changes in topology. Each node maintains its own sequence number. It is incremented whenever the set of neighbors of the node changes. When a route is established, it is stamped with the current sequence number of its destination. As the topology changes, more recent routes will have larger sequence numbers. That way, nodes can distinguish between recent and obsolete routes.

When a node s wants to communicate with a destination d, it broadcasts a route request (RREQ) message to all of its neighbors. The message has the following format:

$$\mathsf{RREQ}(\mathsf{hops\_to\_src}, \mathsf{broadcast\_id}, d, \mathsf{seqno}, s, \mathsf{src\_seq\_no}).$$

Argument hops_to_src determines the current distance from the node that initiated the route request. The initial RREQ has this field set to 0, and every subsequent node increments it by 1. The broadcast_id field is a unique integer assigned to each RREQ originated by s — it is incremented after every RREQ. Argument seqno specifies the least sequence number for a route to d that s is willing to accept (node s uses here the last sequence number it recorded for the destination d, namely seqno_d). Argument src_seq_no is the sequence number of the initiating node s.

When a node t receives a RREQ, it first checks whether it has a route to d marked with a sequence number at least as big as seqno. If it does not, it rebroadcasts the RREQ with an incremented hops_to_src field. At the same time, t can use the received RREQ to set up a reverse route to s. This route would eventually be used to forward replies back to s. If t has a fresh enough route to d, it replies to s (forwarded via the reverse route) with a route reply (RREP) message which has the following format:

$$\mathsf{RREP}(\mathsf{hops}_d, d, \mathsf{seqno}_d, \mathsf{lifetime}_d).$$

Arguments $\mathsf{hops}_d, \mathsf{seqno}_d$, and $\mathsf{lifetime}_d$ are the corresponding attributes of t's route to d. Similarly, if t is the destination itself ($t = d$), it replies with

$$\mathsf{RREP}(0, d, \mathsf{big\_seq\_no}, \mathsf{MY\_ROUTE\_TIMEOUT}).$$

The value of big_seq_no needs to be at least as big as d's own sequence number and at least as big as seqno from the request. Parameter MY_ROUTE_TIMEOUT is the default lifetime, locally configured at d. Every node that receives a RREP increments the value of the hops packet field and forwards the packet along the reverse route to s. When a node receives a RREP for some destination d, it uses information from the packet to update its own route for d. If it already has a route to d, preference is given to the route with a higher sequence number. If sequence numbers are the same, the shorter route is chosen. This rule is used both by s and by all of the intermediate forwarding nodes. The preference rule is important for propagating error messages.

In addition to the routing table, each node s keeps track of the *active neighbors* for each destination d. This is the set of neighboring nodes that use s as their next_d on the way to d. If s detects that its route to d is broken, it sends an unsolicited RREP (error) message to all of its active neighbors for d. This message contains hops equal to 255 (infinity), and seqno equal to one more than the previous sequence number for that route. Because of the previously mentioned preference rule for route selection, such an artificially incremented sequence number forces the recipients to accept this 'route' and propagate it further upstream, all the way to the origin of the route.

4. STABILITY OF RIP

4.1 Formalization

We model the universe $\mathcal{U}$ as a bipartite connected graph whose nodes are partitioned into *networks* and *routers*, such that each router is connected to at least two networks. In other words, *routers* and *networks* are nodes, while *interfaces* are edges. The goal of the protocol is to compute a table at each router providing, for each network n, the length of the shortest path to n and the next hop along one such path. The hop count is limited to a maximum of 16, where 16 means *unreachable*.

Our proof shows that, for each destination d that is less than 16 hops away from every router, the routers will all eventually obtain a correct shortest path to d. An entry for d at a router r consists of three parameters:

$\mathsf{hops}(r)$:	current estimate of the distance metric to d (an integer between 1 and 16 inclusively).
$\mathsf{nextN}(r)$:	the next network on the route to d.
$\mathsf{nextR}(r)$:	the next router on the route to d.

Both r and $\mathsf{nextR}(r)$ must be connected to $\mathsf{nextN}(r)$. We say that r *points to* $\mathsf{nextR}(r)$. Initially, routers connected to d must have their metric set to 1, while others must have it set to values strictly greater than 1. Two routers are *neighbors* if they are connected to the same network. The universe changes its state (*i.e.* routing tables) as a reaction to *update messages* being sent between neighboring routers.

Each update message can be represented as a triple $(\mathsf{snd}, \mathsf{net}, \mathsf{rcv})$, meaning that the router snd sends its current distance estimate through the network net to the router rcv. In some cases this will cause the receiving router to update its own routing entry. An infinite sequence of such messages $(\mathsf{snd}_i, \mathsf{net}_i, \mathsf{rcv}_i)_{i \geq 0}$ is said to be *fair* if every pair of neighboring routers s and r exchanges messages infinitely often:

$$\forall i.\ \exists j > i.\ (\mathsf{snd}_i = s) \text{ and } (\mathsf{rcv}_i = r).$$

This property simply assures that each router will communicate its routing information to all of its neighbors. Distance to d is defined as

$$D(r) = \begin{cases} 1, & \text{if } r \text{ is connected to } d \\ 1 + \min\{D(s) \mid s \text{ neighbor of r}\}, & \text{otherwise.} \end{cases}$$

For $k \geq 1$, the *k-circle* around d is the set of routers

$$C_k = \{r \mid D(r) \leq k\}.$$

For $1 \leq k \leq 15$, we say that the universe is *k-stable* if the following properties S1 and S2 both hold:

(S1): Every router $r \in C_k$ has its metric set to the actual distance: that is, $\mathsf{hops}(r) = D(r)$. Moreover, if r is not connected to d, it has its next router set to the first router on some shortest path to d: that is, $D(\mathsf{nextR}(r)) = D(r) - 1$.

(S2): For every router $r \notin C_k$, $\mathsf{hops}(r) > k$.

Intuitively, in a k-stable universe, all routers inside C_k have converged to the correct routes, while those outside may not have received the advertisements that would allow them to calculate the correct routes. The aim of the routing protocol is to expand the k-stable circle until all routers are contained in it ($k = 15$).

Given a k-stable universe, we say that a router r at distance $k + 1$ from d is *$(k + 1)$-stable* if it has an optimal route: that is, $\mathsf{hops}(r) = k + 1$ and $\mathsf{nextR}(r) \in C_k$.

4.2 Proof Results

Our main goal is to show that a universe running RIP does eventually discover all the shortest paths of length less than 16:

THEOREM 4.1 CORRECTNESS OF RIP. *For any $k < 16$, starting from an arbitrary state of the universe $\mathcal{U}$, for any fair sequence of update messages, there is a time t_k such that $\mathcal{U}$ is k-stable at all times $t \geq t_k$.*

In particular, we want to show that 15-stability will be achieved. Note that the theorem applies to an *arbitrary* initial state. This is important because topology changes could occur during a run of the protocol and leave it in an arbitrary non-stable state. After each topology change, RIP effectively has to start from this arbitrary inital state and re-compute all the routing tables. But as long as these topology changes are not too frequent, Theorem 4.1 applies to the periods in between, guaranteeing eventual convergence if a period is long enough.

Our proof, which we call *the radius proof,* differs from the one described in [Bertsekas and Gallager 1991] for the asynchronous Bellman-Ford algorithm. Rather than induction on estimates for upper and lower bounds for distances, we carry out induction on the radius of the k-stable region around d. The proof has two attributes of interest:

(1) *It states a property about the RIP protocol, rather than the asynchronous distributed Bellman-Ford algorithm.* Closer analysis reveals subtle, but substantial differences between the two. In the case of Bellman-Ford, routers keep all of their neighbors' most recently advertised metric estimates, whereas RIP keeps only the best value. Furthermore, the Bellman-Ford metric ranges over the set of all positive integers, while the RIP metric saturates at 16, which is regarded as infinity. Finally, RIP includes certain engineering optimizations, such as split horizon with poisoned reverse, that do not exist in the Bellman-Ford algorithm.
(2) *The radius proof is more informative.* It shows that correctness is achieved quickly close to the destination, and more slowly further away. We exploit this in the next section to show a real-time bound on convergence.

Theorem 4.1 is proved by induction on k. There are four parts to it:

LEMMA 4.2. *The universe $\mathcal{U}$ is initially 1-stable.*

LEMMA 4.3 PRESERVATION OF STABILITY. *For any $k < 16$, if the universe is k-stable at some time t, then it is k-stable at any time $t' \geq t$.*

LEMMA 4.4. *For any $k < 15$ and router r such that $D(r) = k+1$, if the universe is k-stable at some time t_k, then there is a time $t_{r,k} \geq t_k$ such that r is $(k+1)$-stable at all times $t \geq t_{r,k}$.*

LEMMA 4.5 PROGRESS. *For any $k < 15$, if the universe $\mathcal{U}$ is k-stable at some time t_k, then there is a time $t_{k+1} \geq t_k$ such that $\mathcal{U}$ is $(k+1)$-stable at all times $t \geq t_{k+1}$.*

Lemma 4.2 serves as the basis of the overall induction. Lemma 4.3 is the fundamental safety property, ensuring that once the universe converges to the correct routes, they stay correct. Lemma 4.4 is the main progress property in the proof and gets generalized to Lemma 4.5 which is the inductive step.

4.3 Proof Details and Tool Support

First we write RIP models that can be analyzed by SPIN and HOL. The Promela model of RIP follows directly from the pseudo-code in Appendix A. The process declaration translates to a `proctype`, constants become C-style macro constants, and state is expressed using C-style structs and arrays. All events are expressed as message events on channels; advertisements are asynchronous messages. The process body consists mainly of event handlers for the different events guarded by a case statement and enclosed in an infinite do-loop. The individual event handling routines are translated into the C-style syntax that Promela uses. We simplify the model to deal with only one destination and a fixed number of interfaces.

The HOL theory consists of definitions of the routing table and the *update* function that modifies the routing table based on received advertisements. These definitions extend to a natural definition of the state sequence, which represents the successive states of all the routers in the universe $\mathcal{U}$ as the protocol is executed. Then the network model is defined as a relationship between routers and networks in $\mathcal{U}$. Finally k-stability is defined and the correctness theorem and lemmas formalized.

Lemma 4.2 is easily proved by HOL: it follows from the definition of k-stability, and the state sequence induced by RIP. The safety property, Lemma 4.3, is proved twice: once completely in HOL, and the second time using both HOL and SPIN. We compare the two proofs statistically in Section 7. To prove this lemma, one needs to show that a k-stable universe remains k-stable after an arbitrary update message. Our first HOL proof proceeds by separately verifying that each of the conditions S1 and S2 remain true after an update. This cannot be directly modeled in SPIN, since, for instance, the number of routers inside the k-circle is unknown.

However, it turns out that k-stability gives rise to a nice *abstraction* of the system, which can be used to encode the system in SPIN. We know that in a k-stable universe, the k-circle always advertises the distance k to the outside world. On the other side, all the distances that are advertised to the k-circle from the outside world are strictly greater than k. Therefore, the k-circle can now be modeled as a single router that always advertises the distance of k hops. The outside world can be modeled by a process that always advertises arbitrary distances greater than k. So for the router r such that $D(r) = k + 1$, we can abstract its environment and replace it with one node representing the k-circle and one process representing the rest of the outside world. Using this abstraction, the $\mathcal{U}$ effectively reduces to three nodes, for all properties that need to be proved about r. In addition, the hop counts are abstracted to the conditions $\mathsf{hops} < k + 1, \mathsf{hops} = k + 1$, or $\mathsf{hops} > k + 1$. So our abstract hop count, $\mathsf{abs\_hop\_cnt} \in \{LT, EQ, GR\}$, corresponding to which of the conditions is true of hops.

It is crucial that our abstractions are *finitary* and *property-preserving*. An abstraction is finitary if it reduces the system to a fixed, finite number of states. It is property-preserving (with respect to a specific property) if whenever the abstract system satisfies the property it is also the case that the concrete system satisfies the property. Finitary abstractions, like the one we have described for RIP, are useful because they enable proofs using state-space exploration in a model checker. Our first proof in HOL does not make use of the abstraction and is fairly long, needing 9 intermediate lemmas and 903 steps of deduction. For the second proof, we first prove in HOL that in a k-stable universe, our abstraction is property-preserving for the router r at distance $k+1$. The relationship between the abstract system and the real system is represented as an invariant of the state at r, and is proved inductively using key properties of k-stability. This is a fairly large proof as well and re-uses large chunks of the first HOL proof of Lemma 4.3. However, once the abstraction is proved correct, we can use it to reduce the universe to a finite 3-process system that can be model checked in SPIN. Proofs that can be carried out in either tool are typically done in SPIN, since it provides more automation.

Lemma 4.4, the main progress property in the proof, is proved with SPIN, using the Lemma 4.3 abstraction again. The proof as a whole illustrates well how verification can be split between the two systems: we justify the abstractions using a theorem prover and then we prove the property of the abstract system using a model checker. These two parts are independent and therefore can be done in parallel. Moreover, once a suitable abstraction has been proved, it can often be reused to prove many properties.

Lemma 4.5 is the inductive step, which is derived in HOL as an easy generalization of Lemma 4.4, using the fact that the number of routers is finite. Statistics on the lengths of the proofs and models are presented in Section 7.

5. SHARP TIMING BOUNDS FOR RIP STABILITY

In the previous section, we proved convergence for RIP under the assumption that the topology stays unchanged for some period of time. We now calculate how big that period of time must be. To do this, we need to have some knowledge about the times at which protocol events must occur. In the case of RIP, we use the following:

Fundamental Timing Assumption. There is a value Δ, such that during every topology-stable time interval of the length Δ, each router gets at least one update message from each of its neighbors.

This is the only assumption we make about timing of update messages. RIP routers normally try to exchange messages every 30 seconds; a failure to receive an update within 180 seconds is treated as a link failure. Thus $\Delta = 3$ minutes satisfies the Fundamental Timing Assumption for RIP.

As in the previous section, we will concentrate on a particular destination network d. Our timing analysis is based on the notion of weak k-stability. For $2 \leq k \leq 15$, we say that the universe $\mathcal{U}$ is *weakly k-stable* if the following conditions hold:

(WS1): The universe $\mathcal{U}$ is $(k-1)$-stable.

(WS2): For all routers r on the k-circle: that is $D(r) = k$, either r is k-stable ($\mathsf{hops}(r) = k$ and $\mathsf{nextR}(r) \in C_{k-1}$), or $\mathsf{hops}(r) > k$.

(WS3): For all routers r outside C_k ($D(r) > k$), $\mathsf{hops}(r) > k$.

Weak k-stability is stronger than $(k-1)$-stability, but weaker than k-stability. Conditions WS1 and WS3 are similar to the conditions S1 and S2 for k-stability. However, we make a distinction for the routers r on the k-circle. The only restriction on r is that it cannot have the correct hop count ($\mathsf{hops}(r) = k$) and an incorrect next pointer ($\mathsf{nextR}(r) \notin C_{k-1}$). This ensures that r will get the correct route when it gets the next advertisement from inside C_k. The disjunction in WS2 (which distinguishes weak stability from the ordinary stability) will typically introduce additional complexity in case analyses arising from reasoning about weak stability.

As with k-stability, we have the following:

Lemma 5.1 Preservation of weak stability. *For any $2 \leq k \leq 15$, if the universe is weakly k-stable at some time t, then it is weakly k-stable at any time $t' \geq t$.*

We must also show that the initial state inevitably becomes weakly 2-stable after messages have been exchanged between every pair of neighbors:

Lemma 5.2 Initial progress. *If the topology does not change, the universe becomes weakly 2-stable after Δ time.*

The main progress property says that it takes 1 update interval to get from a weakly k-stable state to a weakly $(k+1)$-stable state. This property is shown in two steps: first we show that condition WS1 for weak $(k+1)$-stability holds after Δ:

LEMMA 5.3. *For any $2 \leq k \leq 15$, if the universe is weakly k-stable at some time t, then it is k-stable at time $t + \Delta$.*

and then we show the same for conditions WS2 and WS3. The following puts both steps together:

LEMMA 5.4 PROGRESS. *For any $2 \leq k < 15$, if the universe is weakly k-stable at some time t, then it is weakly $k+1$-stable at time $t + \Delta$.*

The *radius* of the universe (with respect to d) is the maximum distance from d:

$$R = \max\{D(r) \mid r \text{ is a router}\}.$$

The main theorem describes convergence time for a destination in terms of its radius:

THEOREM 5.5 RIP CONVERGENCE TIME. *A universe of radius R becomes 15-stable within $\min\{15, R\} \cdot \Delta$ time, assuming that there were no topology changes during that time interval.*

The theorem is an easy corollary of the preceding lemmas. Consider a universe of radius $R \leq 15$. To show that it converges in $R \cdot \Delta$ time, observe what happens during each Δ-interval of time:

after Δ	weakly 2-stable	(by Lemma 5.2)
after $2 \cdot \Delta$	weakly 3-stable	(by Lemma 5.4)
after $3 \cdot \Delta$	weakly 4-stable	(by Lemma 5.4)
...	...	...
after $(R-1) \cdot \Delta$	weakly R-stable	(by Lemma 5.4)
after $R \cdot \Delta$	R-stable	(by Lemma 5.3)

R-stability means that all the routers that are not more than R hops away from d will have shortest routes to d. Since the radius of the universe is R, this includes *all* routers.

An interesting observation is that progress from (ordinary) k-stability to (ordinary) $(k+1)$-stability is not guaranteed to happen in less than $2 \cdot \Delta$ time (we leave this to the reader). Consequently, had we chosen to calculate convergence time using stability, rather than weak stability, we would get a worse upper bound of $2 \cdot (R-1) \cdot \Delta$. In fact, our upper bound is sharp: in a linear topology, update messages can be interleaved in such a way that convergence time becomes as bad as $R \cdot \Delta$. Figure 1 shows an example that consists of k routers and has the radius k with respect to d. Router r_1 is connected to d and has the correct metric. Router r_2 also has the correct metric, but points in the wrong direction. Other routers have no route to d. In this state, r_2 will ignore a message from r_1, because that route is no better than what r_2 (thinks it) already has. However, after receiving a message from r_3, to which it points, r_2 will update its metric to 16 and lose the route. Suppose that, from this point on, messages are interleaved in such a way that during every update interval, all routers first send their update messages and then receive update messages from their neighbors. This will cause exactly one new

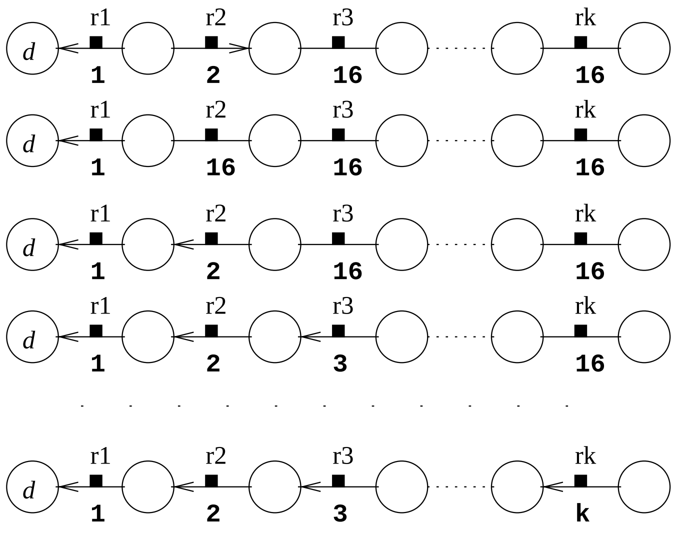

Fig. 1. Maximum Convergence Time

router to discover the shortest route during every update interval. Router r_2 will have the route after the second interval, r_3 after the third, …, and r_k after the k-th. This shows that our upper bound of $k \cdot \Delta$ is reachable.

5.1 Proof Details and Tool Support

The proof of RIP convergence time is similar in structure to that of RIP correctness described in Section 4. The main result, Theorem 5.5 is broken down into Lemmas 5.2, 5.4, and 5.1. An abstraction is used to create a finite abstract model of the universe, for which the Lemmas are proved by model checking in SPIN.

Recall that the abstraction used to prove results about stability, used k-stability to reduce the universe to a 3-process system: representing those inside the k-circle, those outside it, and a router on the edge of the circle. Since weak $(k+1)$-stability implies k-stability, the same abstraction is applicable for weak-stability as well. In addition, we abstract hop counts to $\mathsf{abs\_hop\_cnt} \in \{LT, EQ, GR\}$ as before, representing $\mathsf{hops} < k+1, \mathsf{hops} = k+1$, and $\mathsf{hops} > k+1$. This abstraction yields a finite property-preserving model, for which Lemma 5.1 is proved automatically in SPIN. Lemma 5.2 is similarly proved, using the abstraction instantiated for $k = 1$. For Lemma 5.4 we extend the abstract hop counts to $\mathsf{abs\_hop\_cnt} \in \{LT, EQ, EQ', GR\}$ corresponding to $\mathsf{hops} < k+1, \mathsf{hops} = k+1, \mathsf{hops} = k+2$, and $\mathsf{hops} > k+2$. Lemma 5.4 is then proved automatically in SPIN as well, completing the proof.

SPIN turned out to be extremely helpful for proving properties such as Lemma 5.4, which involve tedious case analysis. To illustrate this, assuming weak k-stability at time t, let us look at what it takes to show that condition WS2 for weak $(k+1)$-stability holds after Δ time. (WS1 will hold because of Lemma 5.3, but further effort is required for WS3.)

To prove WS2, let r be a router with $D(r) = k+1$. Because of weak k-stability at the time t, there are two possibilities for r: (1) r has a k-stable neighbor, or (2) all of the neighbors of r have $\mathsf{hops} > k$. To show that r will eventually progress into either a $(k+1)$-stable state or a state with $\mathsf{hops} > k+1$, we need to further break the case (2) into three sub-cases with respect to the properties of the router that r points to: (2a) r points to $s \in C_k$ (the k-circle), which is the only neighbor of r from C_k, or (2b) r points to $s \in C_k$, but r has another neighbor $t \in C_k$ such that $t \neq s$, or (2c) r points to $s \notin C_k$. Each of these cases, branches into several further sub-cases based on the relative ordering in which r, s and possibly t send and receive update messages.

Doing such proofs by hand is difficult and prone to errors. Essentially, the proof is a deeply nested case analysis in which *final* cases are straight-forward to prove—an ideal task for a fully automated model checker. Our SPIN verification is divided into four parts accounting for differences in possible topologies. These differences arise from the case analyses similar to the one sketched above. Each part has a distinguished process representing r and another processes modeling the environment for r. An environment is an abstraction of the 'rest of the universe'. It generates all message sequences that could possibly be observed by r. In order to simplify the model, our abstraction allows the environment to also generate some message sequences that are not possible in reality. Such abstractions will still be property-preserving for 'all path' properties, stating that something holds in *every* possible run of the system. SPIN considered more cases than a manual proof would have required, 21,487 of them altogether for Lemma 5.4, but it checked these in only 1.7 seconds of CPU time. Even counting set-up time for this verification, this was a significant time-saver. The resulting proof is probably also more reliable than a manual one. We summarize similar analyses for our other results in the conclusions (Section 7).

6. AODV LOOP FREEDOM

As mentioned before, loop-freedom is an important property for distance vector routing protocols. In the context of mobile, ad hoc networking, the topologies are much more dynamic. As a result, the routing protocol is always in a transient state, and loop-freedom becomes even more important. Perkins and Royer [1999] sketch a hand-proof that AODV is loop-free by appealing to the rules by which AODV routes can be formed. However, it is not clear that the proof applies to the AODV standard in all its complexity, especially since significant parts of the standard were still unspecified at the time of that work. We aim to analyze the AODV standard, version 2, to verify that the routes formed by AODV indeed have no loops.

We first attempt to prove loop-freedom for the simple network shown in Figure 2. The tool we use for this finite-instance verification is the model checker SPIN. We write a Promela model of AODV, along the lines of the standard pseudo-code shown in Appendix B, that SPIN can analyze.

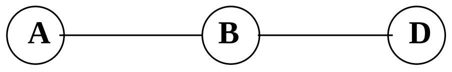

Fig. 2. Sample 3-node Network

We run AODV processes at all 3 nodes—A, B and D. D is the only destination and both A and B attempt to send data to D. The link B—D is fragile and may be broken at any time. The challenge to AODV is to gracefully discover that the B—D link has broken and there is no longer any route from A or B to D. Note that, if A and B form a routing loop, they will never discover that D is unreachable. We model the network and the processes in SPIN and attempt to verify that there is no sequence of events that can result in a routing loop between A and B.

6.1 Loop Conditions

Let A and B have active routes to D to begin with (Figure 3). When we try to verify using SPIN that this configuration will never result in a loop between A and B, SPIN finds a number of counter-examples. On analyzing these counter-examples, we discover 3 scenarios in which a routing loop will indeed be formed. We describe the scenarios below as sequences of events that lead to routing loops.

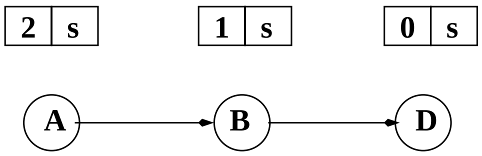

Fig. 3. Initial Routes

LS1. When the link B—D goes down, B generates a RREP with hop count infinity and increments its sequence number for D. If the RREP gets dropped, and B deletes its route before A's route expires, there will be a loop. This scenario is depicted in Figure 4, and is due to Joshua Broch and Dave Maltz who found it by manual inspection. It is also found by SPIN automatically.

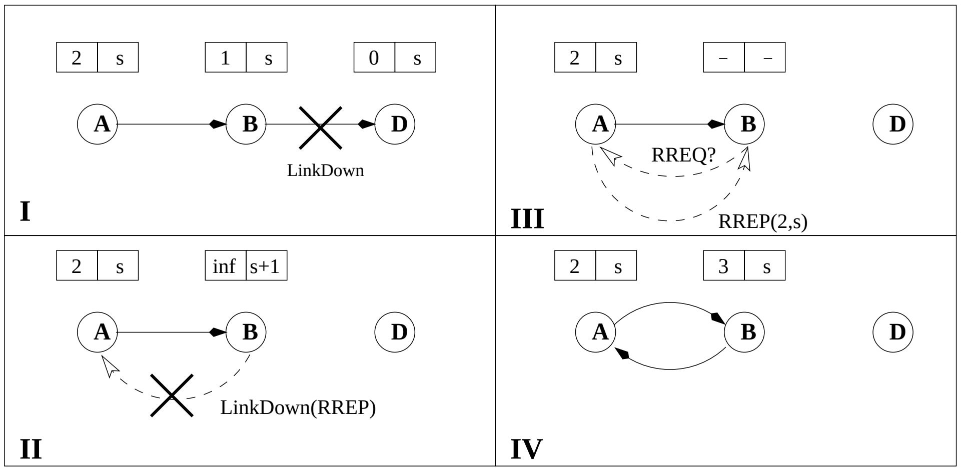

Fig. 4. Loop Condition LS1

LS2. Suppose B's route expires while A is still pointing to it. The standard does not explicitly say what happens when a route expires. Consider the following alternatives for an implementation:

a. Suppose B deletes the route on expiry. Then, there is a sequence of events that lead to a loop as shown in Figure 5.

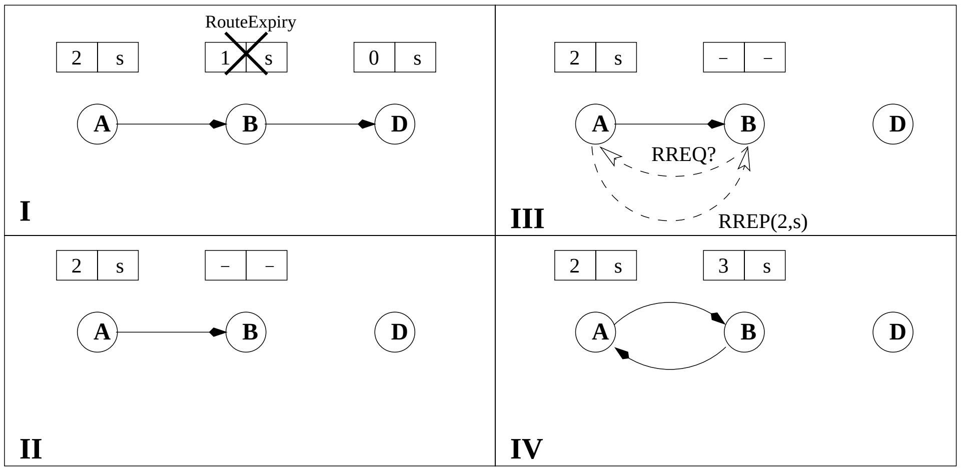

Fig. 5. Loop Condition LS2(a)

b. Suppose B keeps the route, unchanged, as an expired route. Then again, there will be a loop (Figure 6).

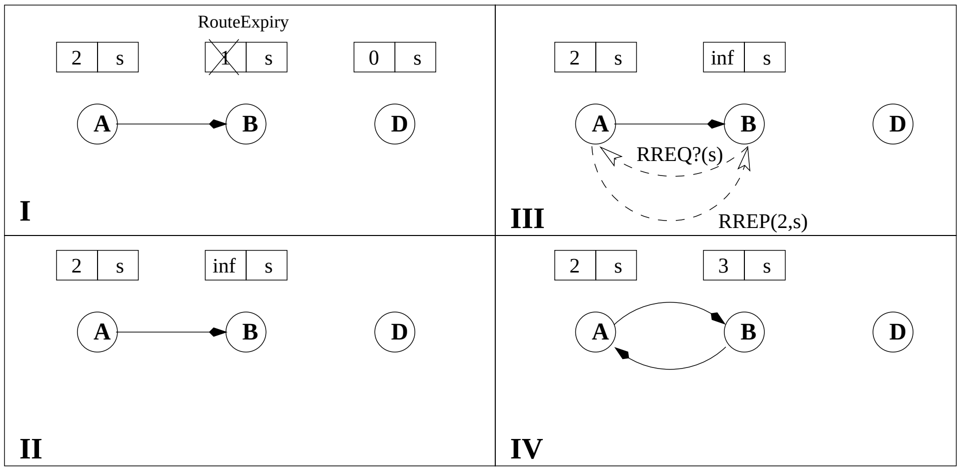

Fig. 6. Loop Condition LS2(b)

c. Suppose B keeps the route as an expired route, increments the route's sequence number for D, and deletes it after some time. B may even decide to send an error message to A. Even in this case, there is a sequence of events (Figure 7) that lead to a loop.

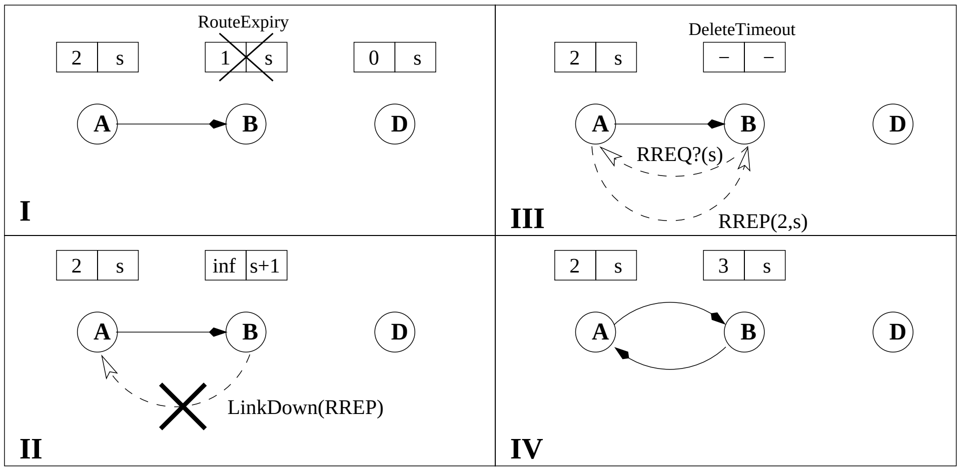

Fig. 7. Loop Condition LS2(c)

d. Finally, suppose B keeps the route as an expired route, increments the route's sequence number, and *never* deletes it. In this case SPIN cannot find a loop. Since an AODV process has unbounded state, SPIN cannot authoritatively say that this alternative will produce no loops. However, it is a good indicator that we have found a loop-free solution.

LS3. Suppose the AODV process at B is restarted suddenly, because of a reboot following (say) a crash. If A does not detect the restart as a link-breakage, and continues to point to B, then there will be a loop when B comes back up and looks for a route to D. This scenario is depicted in Figure 8.
Here, we assume that B restarts in a vanilla state, so this case is essentially equivalent to one in which all the routes at B suddenly expire and are deleted. This leads to the problems in case **LS2**a above.

Each of the scenarios described in this section illustrates gaps in the AODV standard that allow routing loops to be formed despite the loop-prevention mechanisms built into the protocol. However, these counter-examples also indicate the conditions that must hold for loop-freedom to be guaranteed for AODV.

6.2 Tool Support

In this section, we explain what we mean when we say that SPIN found a counter-example that demonstrates a loop. We have described earlier in Section 3.4 how AODV can be specified in a pseudo-code notation. The pseudo-code for AODV is shown in Appendix B.

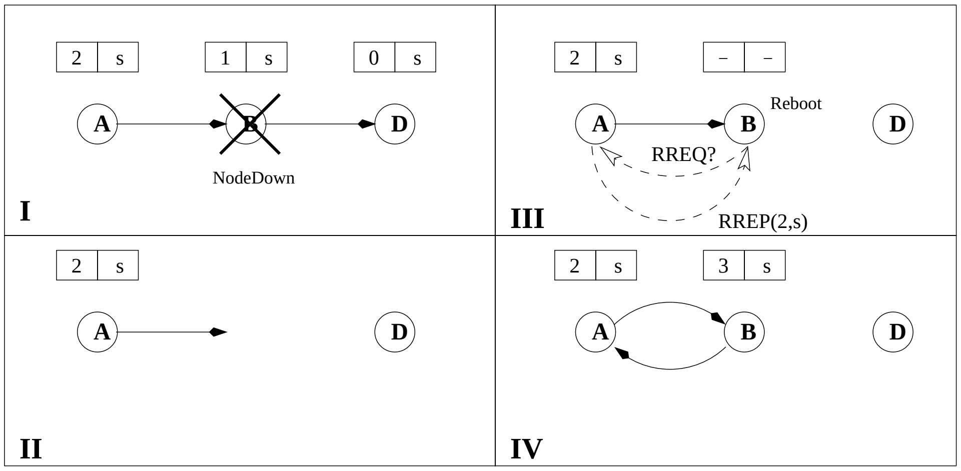

Fig. 8. Loop Condition LS3

The Promela model for AODV follows directly from the pseudo-code in Appendix B. The process definition in the pseudo-code is translated to a Promela `proctype`, as in the case of RIP. The main issue that we have to deal with in the modeling is timing. Promela has no notion of real time, whereas AODV depends crucially on timeouts that trigger various events. We omit the precise timer values and make all timeouts non-deterministic events that can take place at any time. This clearly allows a much larger set of event sequences to occur. As in RIP, we limit the AODV Promela process to one destination and a fixed number of neighbors.

After generating a satisfactory Promela model for AODV, we describe the environment: the 3-node AODV network. There are 3 processes, named A, B and D. The destination node is set to D. The neighbor relation is encoded as follows: B is the only neighbor known to A; A and D are known to B; B is known to D. Each node is only capable of sending messages to the in-queues of its known neighbors. This topology can be hard-coded into the processes themselves or can be implemented by a 'connections' process running in parallel that captures all sent messages and delivers them when appropriate. For a small topology like this one, hard-coding has significant performance advantages in SPIN.

To denote the fragility of the B–D link, we add a non-deterministic clause to the environment process, which at any one point in the execution can send link-broken events to both B and D.

Finally, we define the loop-free property in LTL as a state invariant:

$$\Box(!((\mathsf{next}_D(A) == B) \wedge (\mathsf{next}_D(B) == A)))$$

That is, never during the execution of the 3-node AODV network, does there occur a state where the next pointers of A and B (for the destination D) point to each other.

The Promela processes and LTL definitions are then handed over to SPIN. SPIN is a push-button tool that compiles the Promela process definitions and LTL property definitions into an executable called a protocol analyzer. When executed, the protocol analyzer compares the 3-node AODV network model against the desired loop-free property and generates the counter-examples we have shown. The counter-examples are presented in the form of execution *trails*, which can be visually simulated using the XSPIN graphical environment. We view the simulation and read the trail, in conjunction with reading the Promela code to uncover the sequence of events and possible bugs that led to the counter-example. The scenarios presented in this section are our analyses of the SPIN counter-examples, highlighting the main events that led to the formation of loops in each case.

6.3 Ambiguities in the Standard

Virtually all standard specifications contain ambiguities and omissions. Version 2 of the AODV standard is no exception. A programmer implementing the standard will naturally attempt to resolve the ambiguities reasonably, taking account of special knowledge of the aims of the standard. We have outlined some scenarios in which the AODV standard, version 2, allows loops to be formed; in these scenarios, the standard fails to anticipate some sequence of events that consequently leads to the loop. Each of the scenarios points to an instance in which an implementation could conform to the standard but fail to satisfy a desired property. In the next section, we propose some fixes to the standard, and an invariant-based proof of loop-freedom for the fixed standard. In this section, we outline some other ambiguities in the standard and how we resolved them.

Here are the primary areas we felt needed some further specification.

—The standard does not describe the initial state of the AODV process. Intuitively, it seems clear that the AODV process should start up with empty routing tables; this choice is indeed safe with respect to loops. However, if we choose to start with some *default* routes, then SPIN can demonstrate cases with loops. For our analysis, we assumed that an AODV process begins with an empty routing table.

—The event handler for the reception of RREP packets is not fully described in the standard. An incorrectly written RREP handler could easily cause loops. We resolved this based on the description in [Perkins and Royer 1999]. This matched the approach subsequently described in section 9.5 of version 4 of the standard.

—When an AODV node discovers that the next node on the way to the destination is no longer reachable, the standard says that it must send a route error message in an RREP packet to its neighbors. This RREP packet has a hop count of infinity and a sequence number one more than the sequence number stored at the node. However, the standard does not explicitly say that the sequence number stored at the node must also be increased by one. Indeed, if the stored sequence number is not incremented, SPIN finds a scenario in which there will be a loop. This omission was fixed in later versions, and we believe that it was always intended that the AODV node would increment its stored sequence number as well.

  In a related omission, when AODV was revised to version 5, a new kind of packet (RERR) was introduced to denote route errors, thus simplifying the role of the RREP packets. However, the standard failed to require the incrementing of sequence numbers in the outgoing RERR packets. This error leads to a looping scenario that was discovered by Madanlal Musuvathi using Murϕ (`http://sprout.cs.stanford.edu/dill/murphi`).

Our model of the AODV protocol needs to take these ambiguities into account. To remove these ambiguities, we change the standard pseudo-code described in Appendix B. We make the first three modifications as described in Appendix C, and it is this modified pseudo-code that we model in SPIN and use for the analysis described in Section 6.1.

6.4 Guaranteeing AODV Loop Freedom

Guided by the looping scenarios demonstrated in the previous sections, we describe 3 assumptions under which we claim that AODV will produce and maintain loop-free routes. These assumptions are to be treated as recommendations for changes to the AODV protocol.

A1. When a node discovers that its route to a destination has expired or broken, it increments the sequence number for the route.

A2. Nodes *never* delete routes.

A3. Nodes *always* immediately detect when a neighbor restarts its AODV process. The restart is treated as if all links to the neighbor have broken.

We need to modify the AODVv2 pseudo-code in accordance with these assumptions, and the result is shown as the fourth modification to the pseudo-code described in Appendix C. This modification guarantees assumptions A1 and A2. A3 is an environmental assumption and is not reflected in the pseudo-code. To ensure A3, the environment must send *NChange* events to all neighbors of the restarted node before the node comes up again (before the restart is completed).

Hereafter, we shall analyze the modified version of the AODV pseudo-code.

THEOREM 6.1. *Consider an arbitrary network of nodes running AODVv2. If all nodes conform to the assumptions A1-A3, there will be no routing loops formed.*

To understand why this theorem is true, note that A1 avoids looping scenario LS2(b). Assumption A2 avoids the scenarios LS1 and LS2(a,c). Finally, A3 avoids the scenario LS3.

As mentioned before, a hand proof of AODV loop-freedom is sketched in [Perkins and Royer 1999]. That proof does not take into account many details of AODV like route expiry. We provide a complete automated proof of Theorem 6.1 using the SPIN model checker and HOL theorem prover. Moreover, the proof in [Perkins and Royer 1999] was by contradiction, while our proof is a corollary of the preservation of a key path invariant of the protocol. This invariant is also used to prove route validity.

For arbitrary nodes n and d, we write $\mathsf{seqno}_d(n)(t)$ to denote n's sequence number for the destination d at the time t. We use a similar notation for hops and next. In non-temporal properties, we shall omit the time argument, when it is clear that we are talking about the current values at some given time. Finally, $\mathsf{restart}(n)(t)$ is true if and only if the node n was restarted at time t. Note that the time t we use in this notation is not real-time. It simply indicates points in the execution trace of the model. In particular, if an event e occurs at time t, the event-handler(s) for e are executed and finish execution at t itself. The state at the node n at time t is the state after all event-handlers for events at t have finished executing.

The following is an invariant (over time) of the AODV process at a node n, for every destination d:

THEOREM 6.2. *If* $\mathsf{next}_d(n) = n'$, *then*

$$\begin{array}{ll}(1) & \mathsf{seqno}_d(n) \leq \mathsf{seqno}_d(n'), \text{ and} \\ (2) & \mathsf{seqno}_d(n) = \mathsf{seqno}_d(n') \Rightarrow \mathsf{hops}_d(n) > \mathsf{hops}_d(n').\end{array}$$

The theorem says that the pair $(-\mathsf{seqno}_d, \mathsf{hops}_d)$ strictly decreases in the lexicographic ordering when a next_d pointer is followed. This invariant has two important consequences:

Loop Freedom. Consider the network at any instant and look at all the routing-table entries for a destination d. Any data packet traveling toward d would have to move along the path defined by the next_d pointers. However, we know from Theorem 6.2 that at each hop along this path, either the sequence number must increase or the hop count must decrease. In particular, a node cannot occur at two points on the path. This guarantees loop-freedom for AODV. In other words, Theorem 6.2 implies Theorem 6.1. We prove this in HOL.

Route Validity. Loop-freedom in a finite network guarantees that data paths to a destination are finite. This does not guarantee that the path ends at d, a property we call *route validity*. However, if all the sequence numbers along a path are the same, hop counts must strictly decrease (by Theorem 6.2). In particular, the last node n_l on the path cannot have hop count INFINITY. In AODV, a node with non-infinite hop count must have a route to d. Since n_l does not have a next pointer for d, it must be equal to d.

To prove Theorem 6.2, we first prove the following properties about the routing table at each node n, now considered as a function of time.

LEMMA 6.3. *If* $t_1 \leq t_2$, *and* $\forall t : t_1 < t \leq t_2 . \neg\mathsf{restart}(n)(t)$, *then* $\mathsf{seqno}_d(n)(t_1) \leq \mathsf{seqno}_d(n)(t_2)$.

LEMMA 6.4. *If* $t_1 \leq t_2$, *and* $\mathsf{seqno}_d(n)(t_1) = \mathsf{seqno}_d(n)(t_2)$, *and* $\forall t : t_1 < t \leq t_2 . \neg\mathsf{restart}(n)(t)$, *then* $\mathsf{hops}_d(n)(t_1) \geq \mathsf{hops}_d(n)(t_2)$.

Intuitively, Lemma 6.3 states that the sequence number for a single destination never decreases over time, as long as the node is up and running. Lemma 6.4 says that if the sequence number stays unchanged over some period of time, then the hop count does not increase during that time.

Suppose $\mathsf{next}_d(n)(t) = n'$. Intuitively, this route must be the result of a route update message sent to n by n' at some earlier time. The following lemma captures this intuition, in terms of this last update time (lut).

LEMMA 6.5. *If* $\mathsf{next}_d(n)(t) = n'$, *then there exists a time* $\mathsf{lut} \leq t$, *such that:*

$$\begin{array}{ll}(1) & \mathsf{seqno}_d(n)(t) = \mathsf{seqno}_d(n')(\mathsf{lut}), \text{ and} \\ (2) & \mathsf{hops}_d(n)(t) = 1 + \mathsf{hops}_d(n')(\mathsf{lut}), \text{ and} \\ (3) & \forall t' : \mathsf{lut} < t' \leq t . \neg\mathsf{restart}(n')(t').\end{array}$$

This lemma says that if n points to n', this must be a result of the last update sent from n' to n (at time lut). Moreover, n' cannot have restarted in the meantime, because A3 assures us that if it had then n would no longer be pointing at n'.

It is not hard to see that the three lemmas together imply Theorem 6.2. First, assume that at time t, $\mathsf{next}_d(n)(t) = n'$. Then we have from Lemma 6.5 that $\forall t' : \mathsf{lut} < t' \leq t. \neg\mathsf{restart}(n')(t')$.

Now, we use Lemmas 6.3 and 6.5, applied to lut and t, yielding

$$\mathsf{seqno}_d(n)(t) = \mathsf{seqno}_d(n')(\mathsf{lut}) \leq \mathsf{seqno}_d(n')(t),$$

which is the first part of Theorem 6.2. Furthermore, if $\mathsf{seqno}_d(n')(\mathsf{lut}) = \mathsf{seqno}_d(n')(t)$, then we have

$$\mathsf{hops}_d(n)(t) - 1 = \mathsf{hops}_d(n')(\mathsf{lut}) \geq \mathsf{hops}_d(n')(t)$$

because of Lemmas 6.4 and 6.5. This shows that $\mathsf{hops}_d(n)(t) > \mathsf{hops}_d(n')(t)$, which is the second part of Theorem 6.2. As indicated earlier, Theorem 6.2 suffices to guarantee loop freedom (Theorem 6.1).

6.5 Proof Details and Tool Support

In the previous section, we showed how loop-freedom for AODV networks (Theorem 6.1) reduces to three local properties: Lemmas 6.3, 6.4, and 6.5. The proof that these lemmas together imply Theorem 6.2 is carried out in HOL and involves a few steps of simple deductive reasoning, along the lines of the informal argument in the previous section. Using Theorem 6.2, we also prove Theorem 6.1 in HOL. This proof is slightly longer and involves the definition of a loop and deductive reasoning on how the loop-free invariant between neighbors (Theorem 6.2) extends to loop-freedom over a path (Theorem 6.1). This again is just a formalization of the argument presented in the previous section.

Each of the Lemmas 6.3, 6.4, and 6.5 is individually proved in SPIN. This is possible because these lemmas express properties of the state at one or at most two AODV processes.

We have earlier described how AODV processes are modeled in SPIN. For Lemma 6.3, we take one AODV process and try to prove that the sequence numbers are monotonically non-decreasing. This can be done by composing the AODV process A with an environment process E that generates all possible messages as input to A. Then, we wish to prove that, in this model, the sequence number of A never decreases. As long as both the number of states of the AODV process and the number of possible messages have low enough bounds, we can carry out the automatic verification in SPIN. However, sequence numbers in AODV are 32-bit integers, and exploring the entire sequence number space for this property is not feasible.

We solve this problem by introducing a property-based abstraction for sequence numbers and hop counts at every node. Note that all we need to show is that for any state s, the sequence number at s is greater than or equal to the sequence number at the next state s'. So we manually *slice* the Promela code with respect to this property and discover that all the boolean conditions it depends on involve comparisons between the current sequence number and hop count (at s) with the sequence number and hop count in the message. Therefore, we abstract the state and the messages as follows:

Let $s = (\mathsf{seqno}, \mathsf{hops}, \mathsf{next})$. For every state $s' = (\mathsf{seqno}', \mathsf{hops}', \mathsf{next}')$ or message $m' = (\mathsf{seqno}', \mathsf{hops}')$:

—Instead of the sequence number seqno', we record only whether $\mathsf{seqno}' > \mathsf{seqno}$, $\mathsf{seqno}' = \mathsf{seqno}$, or $\mathsf{seqno}' < \mathsf{seqno}$. The abstract sequence number, $\mathsf{abs\_seqno} \in \{GR, EQ, LT\}$ accordingly.

—Instead of the hop count hops', we record only whether $\mathsf{hops}' > \mathsf{hops}$, $\mathsf{hops}' = \mathsf{hops}$, or $\mathsf{hops}' < \mathsf{hops}$. The abstract hop count, $\mathsf{abs\_hop\_cnt} \in \{GR, EQ, LT\}$ accordingly.

—Instead of the next pointer next', we record only whether $\mathsf{next}' = \mathsf{next}$ or $\mathsf{next}' \neq \mathsf{next}$. The abstract next, $\mathsf{abs\_next} \in \{EQ, NE\}$ accordingly.

We then modify (abstract) the Promela code of the AODV process, so that it reacts to abstract messages, and maintains the abstract state of the process. The required modifications are well-known for such abstractions [Clarke et al. 1994]. They simply involve modifying all boolean conditions involving the abstracted variables, and assignments to the abstracted variables. Observe that now both the state space and the message space are very small. We can now ask SPIN to verify that the new state after one transition must have sequence number greater than or equal to the sequence number at s. SPIN compares the abstract AODV model with this property, and verifies that it is true, by generating all possible (abstract) messages and events and executing all the event-handlers in the AODV model. The events include link-breakage and node-restart. Lemma 6.3 is thus proved. The same abstraction is then used to prove Lemma 6.4 automatically in SPIN.

Finally, to prove Lemma 6.5, we need to add some information to the abstract state. We also need the following information for every state $s' = (\mathsf{seqno}', \mathsf{hops}', \mathsf{next}')$ or message $m' = (\mathsf{seqno}', \mathsf{hops}')$:

—For the sequence number seqno', whether $\mathsf{seqno}' = \mathsf{seqno}(n')(\mathsf{lut})$, or $\mathsf{seqno}' \neq \mathsf{seqno}(n')(\mathsf{lut})$. Now, the abstract sequence number:
$\mathsf{abs\_seqno} \in \{GR, EQ, LT\} \times \{EQ, NE\}$, where the first component is as before and the second component represents the new information.

—For the hop count hops', whether $\mathsf{hops}' = \mathsf{hops}(n')(\mathsf{lut})+1$, or $\mathsf{hops}' \neq \mathsf{hops}(n')(\mathsf{lut})+1$. The new abstract hop count:
$\mathsf{abs\_hop\_cnt} \in \{GR, EQ, LT\} \times \{EQ, NE\}$ accordingly.

—For the next pointer next', whether $\mathsf{next}' = n'$ or $\mathsf{next}' \neq n'$. The new abstract next: $\mathsf{abs\_next} \in \{EQ, NE\} \times \{EQ, NE\}$ accordingly.

Subsequently, Lemma 6.5 is also automatically verified by SPIN.

One aspect of the proof remains incomplete. How do we know that the abstractions described above are correct? We have argued informally that the abstraction works for the particular case of the property described in Lemma 6.3. However, we have not formally proved that the modifications are property-preserving. In fact, it is not clear whether it is possible for an abstract AODV process to compute its next abstract state based on the reduced information available about the messages and previous state. In this case, we were able to manually modify the code in accordance with the abstract state and messages. Instead, it would be desirable to specify the modifications and prove them correct in HOL, and find an automated strategy for carrying out the abstraction. However, in this case, we choose not to carry out the proof, as it would require too much effort, not commensurate with the expected gain. We acknowledge that the methodology we outlined for RIP has limitations in applicability as the protocols grow more complex. For larger protocols like AODV, formally specifying all abstractions and carrying out complete end-to-end proofs may not be feasible with current tools.

6.6 Alternative Strategies

In Section 6.4, we proposed a way to address loop-freedom problems in the AODVv2 standard based on the assumptions A1-A3. We then proved that the modified specification, as shown in Appendix C, is loop-free. However, there are certainly other ways to ensure loop-freedom without making a strong assumption like A2 (*Nodes never delete routes*). In fact, at the time of this writing, the AODV standard has been revised to version 10, and from version 5, it contains alternative strategies, proposed by us, for addressing the issues that we have found.

In particular, A2 is replaced by a weaker assumption:

A2' A node n does not delete its route as long as some other node is using the route ($\exists n'.\mathsf{next}_d(n') = n$).

However, it is not obvious how assumptions like A2' and A3 are to be guaranteed. One possibility is to add some reliability to the routing protocol: ensure that error messages always reach the intended recipient. However, this would involve substantial changes to the protocol, such as adding new kinds of packets. Our proposal to the AODV standard team involved using existing timers in the protocol to guarantee the assumptions. Informally, we ensure A2' by making sure that whenever $\mathsf{next}_d(n') = n$, the lifetime of the route of n' is less than the lifetime of the route of n. In this way, n will never delete its route before n' does. To ensure A3, we stipulate that when a node n restarts, it must idly wait for a long enough period so that all routes using n expire. Essentially, n must wait for a time interval of $\max\{\mathsf{lifetime}_d(n') \mid \mathsf{next}_d(n') = n\}$.

These changes are different from those described in this paper but try to achieve the same logical behavior by using subtle relationships between timers. A full formal analysis of AODVv5 or later versions would require tools that can analyze real-time behavior beyond what SPIN, as it currently stands, is able to achieve. Arguably, the standard should have been modified to aid simpler reasoning strategies based on assumptions like A1-3 rather than more minimal but subtler conditions on timers. Progress in addressing this kind of balance should be an area of research for achieving higher assurance in networking standards.

7. CONCLUSION

This paper demonstrates the feasibility and value of automated verification of routing protocols. Our results show that it is possible to provide formal analysis of correctness for routing protocols from IETF standards and drafts with reasonable effort and speed, thus demonstrating that these techniques can effectively supplement other means of improving assurance such as manual proof, simulation, and testing. Specific technical contributions include: the first proof of the correctness of the RIP standard, statement and automated proof of a sharp real-time bound on the convergence of RIP, and an automated proof of loop-freedom for AODV.

Table 7 summarizes some of our experience with the complexity of the proofs in terms of our automated support tools. The complexity of an HOL verification

Table III. Protocol Verification Effort

Task	HOL	SPIN
Modeling RIP	495 lines, 19 defs, 20 lemmas	141 lines
Proving Lemma 4.3 Once	9 lemmas, 119 cases, 903 steps	
Proving Lemma 4.3 Again	29 lemmas, 102 cases, 565 steps	207 lines, 439 states
Proving Lemma 4.4	Reuse Lemma 4.3 Abstractions	285 lines, 7116 states
Proving Lemma 5.1	Reuse Lemma 4.3 Abstractions	216 lines, 1019 states
Proving Lemma 5.2	Reuse Lemma 4.3 Abstractions	221 lines, 1139 states
Proving Lemma 5.4	Reuse Lemma 4.3 Abstractions	342 lines, 21804 states
Modeling AODV	95 lines, 6 defs	302 lines
Proving Lemma 6.3		173 lines, 5106 states
Proving Lemma 6.4		173 lines, 5106 states
Proving Lemma 6.5		157 lines, 721668 states
Proving Theorem 6.2	4 lemmas, 2 cases, 5 steps	
Proving Theorem 6.1	4 lemmas, 5 cases, 49 steps	

for the human verifier is described with the following statistics measuring things written by a human: the number of *lines* of HOL code, the number of *lemmas* and *definitions*, and the number of proof *steps*. Proof steps were measured as the number of instances of the HOL construct `THEN`. The HOL automated contribution is measured by the number of *cases* discovered and managed by HOL. This is measured by the number of `THENL`'s, weighted by the number of elements in their argument lists. The complexity of SPIN verification for the human verifier is measured by the number of *lines* of Promela code written. The SPIN automated contribution is measured by the number of *states* examined and the amount of *memory* used in the verification. As we mentioned before, SPIN is memory bound; each of the verifications took less than a minute and the time is generally proportional to the memory used. Most of the lemmas consumed the SPIN-minimum of 2.54MB of memory; Lemma 6.5 required 22.8MB. The figures were collected for runs on a lightly-loaded Sun Ultra Enterprise with 1016MB of memory and 4 CPU's running SunOS 5.5.1. The tool versions used were HOL90.10 and SPIN-3.24. We carried out parallel proofs of Lemma 4.3, the Stability Preservation Lemma, using HOL only and HOL together with SPIN. The HOL proof scripts and SPIN models used in this paper are available for reference on the World Wide Web (`www.cis.upenn.edu/verinet/RoutingVerification`).

Perhaps because of the difficulties in adapting unbounded or infinite state verification to finite state verification tools, there have been relatively few attempts at verifying routing protocols. However, there are successful efforts that verify specific configurations or search for defects. One study [Jackson et al. 1999] used a tool called Nitpick (`www.cs.cmu.edu/~nitpick`) to discover the possibility of caching loops in the internetwork protocol Mobile IPv6. Another study [Cypher et al. 1998] analyzed the ATM network routing protocol PNNI using SPIN as the verification tool and Promela as the specification language. A verification [Wang et al. 2000] of an active network routing protocol for a specific network configuration was given using the Maude system (`maude.csl.sri.com`). Our own work on bug searching has focused on the analysis of network simulation traces. A toolset called Verisim for logical testing of network simulations is described in [Bhargavan et al. 2002] and applied to AODV. If an error is found in an implementation, it is important to know whether it comes from an incorrectly implemented standard or from a flaw in the standard itself. We provided an automated approach to making this determination using a technique we call Fault Origin Adjudication [Bhargavan et al. 2000c]. A broader survey of tool-specific issues for specification, verification and testing of routing protocols can be found in [Bhargavan et al. 2000b]. A classification of logical testing techniques is presented in [Bhargavan et al. 2000a].

APPENDIX

We provide pseudo-code for the RIP and AODV protocols in this appendix. The pseudo-code for a protocol process is broken down into six sections. *Constants* lists some fixed or locally configured constants that the routing process uses. *State* describes information that the router keeps in variables and tables as well as timers that generate timeout events after a certain amount of time has passed. *Initially* describes the initial state of the variables. *Events* lists the events that the routing process recognizes. *Utility functions* describes functions that the routing process can invoke; these may cause events recognized by other routing processes. *Event handlers* describes how the events recognized by the process are dealt with. Events and their handlers generally fall into two categories: receipt of a packet and expiration of a timer. The former is represented abstractly here as an event with some associated data, typically the contents of the received packet.

Timers can be thought of as 'stopwatches'. A timer is a special kind of variable that continuously decreases its value as long as it is greater than zero. When a timer reaches zero, it generates a *timeout* event. Just like a stopwatch, one can **set** a timer to a specific value, or **deactivate** it. The current value of a timer (the remaining time before timeout) can be read at any moment.

Our syntax for any kind of 'packet send' operation requires that contents of the packet be enclosed in rectangular brackets. Our packet format generally reflects logical, rather than physical structure. In some cases, AODV needs to use the IP destination field of an IP packet. We include that field at the end, after the logical contents. A typical packet is hence denoted as [*logical contents*; *DestIP*].

A. RIP PSEUDO-CODE

```
process RIPRouter

state:
  $me$                                  // ID of the router
  $interfaces$                          // Set of router's interfaces
  $known$                               // Set of destinations with known routes
  $hops_{dest}$                         // Estimated distance to $dest$
  $nextRouter_{dest}$                   // Next router on the way to $dest$
  $nextIface_{dest}$                    // Interface over which the route advertisement was received
  timer $expire_{dest}$                 // Expiration timer for the route
  timer $garbageCollect_{dest}$         // Garbage collection timer for the route
  timer $advertise$                     // Timer for periodic advertisements

initially:
```

```
{
  $known \leftarrow$ the set of all networks to which the router is connected.
  for $dest \in known$
  {
    $hops_{dest} = 1$
    $nextRouter_{dest} = me$
    $nextIface_{dest} =$ the interface that connects the router to $dest$.
  }
  set $advertise$ to $30$ $seconds$
}

events:
  $receive\ RIP\ (router, dest, hopCnt)\ over\ iface$
  $timeout\ (expire_{dest})$
  $timeout\ (garbageCollect_{dest})$
  $timeout\ (advertise)$

utility functions:
  $broadcast(msg, iface)$
  {
    Broadcast message $msg$ to all the routers attached to the network on the other side
    of interface $iface$.
  }

event handlers:
  $receive\ RIP\ (router, dest, hopCnt)\ over\ iface$
  {
    $newMetric \leftarrow min\ (1 + hopCnt, 16)$
    if $(dest \notin known)$ and $(newMetric < 16)$ then
    {
      $known \leftarrow known \cup \{dest\}$
      $hops_{dest} \leftarrow newMetric$
      $nextRouter_{dest} \leftarrow router$
      $nextIface_{dest} \leftarrow iface$
      set $expire_{dest}$ to $180$ $seconds$
    } else
    {
      if $(hops_{dest} < 16$ and $router = nextRouter_{dest})$ or $(newMetric < hops_{dest})$
      {
        $hops_{dest} \leftarrow newMetric$
        $nextRouter_{dest} \leftarrow router$
        $nextIface_{dest} \leftarrow iface$
        if $(newMetric = 16)$ then
        {
          deactivate $expire_{dest}$
          set $garbageCollect_{dest}$ to $120$ $seconds$
        } else
        {
          deactivate $garbageCollect_{dest}$
          set $expire_{dest}$ to $180$ $seconds$
        }
      }
    }
  }

  $timeout$ ($expire_{dest}$)
  {
    $hops_{dest} \leftarrow 16$
    set $garbageCollect_{dest}$ to $120$ $seconds$
  }

  $timeout$ ($garbageCollect_{dest}$)
  {
    $known \leftarrow known - \{dest\}$
  }

  $timeout$ ($advertise$)
  {
    for each $dest \in known$ do
      for each $i \in interfaces$ do
      {
        if ($i \neq nextIface_{dest}$) then
        {
          $broadcast$ ([$RIP(me, dest, hops_{dest})$], $i$)
        } else
        {
          $broadcast$ ([$RIP(me, dest, 16)$], $i$)        // Split horizon with poisoned reverse
        }
      }
      set $advertise$ to $30$ $seconds$
  }
```

B. AODVV2 PSEUDO-CODE

```
process AODVRouter

constants:
  ∞ (INFINITY)                    = 255                // Maximum expected network diameter
  NET_DIAMETER                    = 35                 // Set according to network size
  NODE_TRAVERSAL_TIME             = 40 milliseconds    // Set according to link characteristics
  RREP_WAIT_TIME                  = 3* NODE_TRAVERSAL_TIME * NET_DIAMETER
  ACTIVE_ROUTE_TIMEOUT            = 3000 milliseconds
  MY_ROUTE_TIMEOUT                = 6000 milliseconds
  BAD_LINK_LIFETIME               = 2* RREP_WAIT_TIME
  REV_ROUTE_LIFE                  = RREP_WAIT_TIME
  BCAST_ID_SAVE                   = 30000 milliseconds

state:
  me                   // ID of the router
  mySeqno              // Router's own sequence number
  myBcastID            // Router's current broadcast ID
  known                    // Set of destinations with known routes
  neighbors                // Set of known neighbors
  $seqno_{dest}$           // Last destination sequence number known for dest
  $hops_{dest}$            // Distance in hops to dest
  $next_{dest}$            // Next hop toward dest
  $active_{dest}$          // Set of active neighbors using route to dest
  timer $lifetime_{dest}$           // Route expiration timer
  timer $activeTimer_{dest,n}$      // Active neighbor timer

events:
  receive RREQ(hopCnt, bcastID, dest, destSeqno, source, sourceSeqno) from sender
                                        // Received a Route Request broadcast by some neighbor
  receive RREP(hopCnt, dest, destSeqno, lifetime); DestIP from sender
                                        // Received a Route Reply, to be forwarded to DestIP
  receive NChange                       // Triggered when the set of neighbors change
  receive Packet; DestIP from sender    // Received a IP packet that is not an RREP,
                                        // to be forwarded to DestIP
  timeout ($lifetime_{dest}$)           // Triggered when $lifetime_{dest}$ times out
  timeout ($activeTimer_{dest,n}$)      // Triggered when $activeTimer_{dest,n}$ times out

utility functions:
  seen (source, bcastID)
  {
    Determines whether a RREQ from source with the same or more recent broadcast ID as
    bcastID has already been received by the router within the last BCAST_ID_SAVE milliseconds.
  }

  updateRoute (dest, destSeqno, hopCnt, nextHop, ltime)
  {
    Update the routing table with a new route to dest, which is hopCnt hops long,
    continues via nextHop and has the attached destination sequence number destSeqno.
    If no previous route to dest exists or if the new route is better than a previously existing one,
    install the new route with $lifetime_{dest}$ timer set to ltime and include dest in known.
  }

  updateTable ()
  {
    Invalidate all entries in the routing table that use a non-neighbor as their nextHop
    by setting their hops to infinity and their lifetime to BAD_LINK_LIFETIME.
  }

  broadcast (msg)
  { Broadcast the message msg to all neighboring nodes. }

  neighborcast (msg, n)
  { Send the message msg to the neighbor n. }

  computeNeighbors ()
  { Return the current (most recent) set of neighbors. }
```

```
event handlers:
  receive $RREQ(hopCnt, bcastID, dest, destSeqno, source, sourceSeqno)$ from sender
  {
    if not $seen(source, bcastID)$
    {
      $hopCnt \leftarrow max(hopCnt + 1, \infty)$
      if $(dest = me)$ then
      {
        $updateRoute\ (source, sourceSeqno, hopCnt, sender,$ ACTIVE_ROUTE_TIMEOUT$)$
        $mySeqno \leftarrow max(mySeqno, destSeqno)$
        $neighborcast\ ([RREP(0, me, mySeqno,$MY_ROUTE_TIMEOUT$); source], next_{source})$
      } else
      {
        $updateRoute\ (source, sourceSeqno, hopCnt, sender, max($REV_ROUTE_LIFE$, lifetime_{source}))$
        if $(dest \in known)$ and $(hops_{dest} < \infty)$ and $(seqno_{dest} \geq destSeqno)$ then
        {
          $neighborcast\ ([RREP(hops_{dest}, dest, seqno_{dest}, lifetime_{dest}); source], next_{source})$
          $n \leftarrow next_{dest}$
          $active_{source} \leftarrow active_{source} \cup \{n\}$
          set $activeTimer_{source,n}$ to ACTIVE_ROUTE_TIMEOUT
        } else
        {
          $broadcast\ ([RREQ(hopCnt, bcastID, dest, destSeqno, source, sourceSeqno)])$
        }
      }
    }
  }

  receive $RREP\ (hopCnt, dest, destSeqno, lifetime); DestIP$ from sender
  {
    // The standard does not specify exactly how to handle incoming RREPs.
    // They are supposed to be forwarded towards $DestIP$ with incremented $hopCnt$.
  }

  receive $NChange$
  {
    $newNeighbors \leftarrow computeNeighbors\ ()$
    $disconnected \leftarrow neighbors - newNeighbors$
    $neighbors \leftarrow newNeighbors$
    $mySeqno \leftarrow mySeqno + 1$
    for $dest \in known$
    {
      if $(next_{dest} \in disconnected)$
      {
        for $n \in active_{dest}$
        {
          $neighborcast\ ([RREP(\infty, dest, 1 + seqno_{dest}, 0); n], n)$
        }
      }
    }
    $updateTable\ ()$
}

$receive\ Packet; DestIP\ from\ sender$
{
  if ($DestIP \neq me$)
  {
    if ($DestIP \in known$) then
    {
      $active_{DestIP} \leftarrow active_{DestIP} \cup \{sender\}$
      set $lifetime_{DestIP}$ to ACTIVE_ROUTE_TIMEOUT
      set $activeTimer_{DestIP,sender}$ to ACTIVE_ROUTE_TIMEOUT
      $neighborcast\ ([Packet; DestIP], next_{DestIP})$    // Forward the packet towards $DestIP$
    } else
    {
      $myBcastID \leftarrow myBcastID + 1$
      $broadcast\ ([RREQ\ (0, myBcastID, DestIP, seqno_{DestIP}, me, mySeqno)])$
      Queue the packet and forward it upon establishing a route to $DestIP$.
    }
  }
}

$timeout\ (lifetime_{dest})$
{
  if ($hops_{dest} = \infty$) then
  {
    Mark entry for $dest$ as 'erasable'. Erasable entries can be garbage collected.
    Garbage collecting sets $seqno_{dest}$ to 0, and $next_{dest}$ to some undefined value.
  } else
  {
    $hops_{dest} \leftarrow \infty$
    $known \leftarrow known - \{dest\}$
    set $lifetime_{dest}$ to BAD_LINK_LIFETIME
  }
}

$timeout\ (activeTimer_{dest,n})$
{
  $active_{dest} \leftarrow active_{dest} - \{n\}$
}
```

C. MODIFIED AODV PSEUDO-CODE

Below we list four modifications to the original AODVv2 pseudo-code. The first three modifications account for the ambiguities in the standard that needed to be filled in before the verification. These ambiguities are discussed in Section 6.3. The last modification is a real addition to the standard that is needed to prevent loops. It is based on the recommendations A1 and A2 from Section 6.4.

(1) We include the initialization section.

initially:

```
{
  $mySeqno \leftarrow 0$
  $myBcastID \leftarrow 0$
  $known \leftarrow \emptyset$
}
```

(2) We include the handler for RREP events, which was missing in the standard. Given the rest of the specification, we believe that the following code accurately describes the desired functionality.

```
$receive\ RREP\ (hopCnt, dest, destSeqno, lifetime); DestIP\ from\ sender$
{
  **if** $(DestIP = me)$ **then**
  {
    **if** $(hopCnt = \infty)$ **and** $(dest \in known)$ **and** $(seqno_{dest} < destSeqno)$ **then**
    {
      $updateRoute\ (dest, destSeqno, \infty, sender,$ BAD_LINK_LIFETIME$)$
      **for** $n \in active_{dest}$
      {
        $neighborcast\ ([RREP\ (\infty, dest, destSeqno, 0); n], n)$
      }
    }
    **else**
    {
      $hopCnt \leftarrow max(hopCnt + 1, \infty)$
      $updateRoute\ (dest, destSeqno, hopCnt, sender, lifetime)$
    }
  }
  **else**
  {
    $hopCnt \leftarrow max(hopCnt + 1, \infty)$
    $updateRoute\ (dest, destSeqno, hopCnt, sender, lifetime)$
    $neighborcast\ ([RREP(hopCnt, dest, destSeqno, lifetime); DestIP], next_{DestIP})$
  }
}
```

(3) If a local topology change breaks the node's route to some destination, the node should increase the sequence number for that destination. Notice that this is consistent with the sequence number that the node advertises in an unsolicited RREP in that case. Below is the modified pseudo-code for the *NChange* handler. The shaded part is the addition.

```
  $receive\ NChange$
  {
    $newNeighbors \leftarrow computeNeighbors\ ()$
    $disconnected \leftarrow neighbors - newNeighbors$
    $neighbors \leftarrow newNeighbors$
    $mySeqno \leftarrow mySeqno + 1$
    **for** $dest \in known$
    {
      **if** $(next_{dest} \in disconnected)$
      {
        for $n \in active_{dest}$
        {
          $neighborcast\ ([RREP(\infty, dest, 1 + seqno_{dest}, \mathsf{BAD\_LINK\_LIFETIME}); n], n)$
          $seqno_{dest} \leftarrow seqno_{dest} + 1$
        }
      }
    }
    $updateTable\ ()$
  }
```

(4) Nodes should never 'forget' sequence numbers unless they restart the AODV process. This simplifies the handler for route expiry, which only disables the route and increases the sequence number.

```
timeout ($lifetime_{dest}$)
{
  $hops_{dest} \leftarrow \infty$
  $seqno_{dest} \leftarrow seqno_{dest} + 1$
  $known \leftarrow known - \{dest\}$
}
```

ACKNOWLEDGMENTS

We would like to thank the following people for their assistance and encouragement: Roch Guerin, Elsa L. Gunter, Luke Hornof, Sampath Kannan, Insup Lee, and Charles Perkins. We also thank anonymous referees for their input. This research was supported by NSF Contract CCR-9505469, and DARPA Contract F30602-98-2-0198.

REFERENCES

Bertsekas, D. P. and Gallager, R. 1991. *Data Networks.* Prentice Hall.

Bhargavan, K., Gunter, C., Kim, M., Lee, I., Obradovic, D., Sokolsky, O., and Viswanathan, M. 2002. Verisim: Formal Analysis of Network Simulations. *IEEE Transactions on Software Engineering 28,* 2 (February), 129–145. Originally appeared in Proc. International Symposium on Software Testing and Analysis (ISSTA), 2000.

Bhargavan, K., Gunter, C. A., and Obradovic, D. 2000a. A Taxonomy of Logical Network Analysis Techniques. Tech. Rep. MS-CIS-00-14, University of Pennsylvania.

Bhargavan, K., Gunter, C. A., and Obradovic, D. 2000b. An Assessment of Tools used in the Verinet Project. Tech. Rep. MS-CIS-00-15, University of Pennsylvania.

Bhargavan, K., Gunter, C. A., and Obradovic, D. 2000c. Fault Origin Adjudication. In *Formal Methods in Software Practice (FMSP).* Portland, OR.

Chiang, C. 1997. Routing in Clustered Multihop, Mobile Wireless Networks with Fading Channel. In *Proceedings of IEEE SICON '97.* 197–211.

Clarke, E. M., Grumberg, O., and Long, D. E. 1994. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems 16,* 5, 1512–1542.

Cypher, D., Lee, D., Martin-Villalba, M., Prins, C., and Su, D. 1998. Formal Specification, Verification, and Automatic Test Generation of ATM Routing Protocol: PNNI. In *Formal Description Techniques & Protocol Specification, Testing, and Verification (FORTE/PSTV) IFIP.*

Freier, A. O., Karlton, P., and Kocher, P. C. 1996. *Secure Socket Layer.* IETF Draft. home.netscape.com/eng/ssl3.

Gao, L. and Rexford, J. 2000. Stable Internet Routing Without Global Coordination. In *ACM SIGMETRICS*.

Gordon, M. J. C. and Melham, T. F., Eds. 1993. *Introduction to HOL: A theorem proving environment for higher order logic.* Cambridge University Press.

Griffin, T. G. and Wilfong, G. 1999. An Analysis of BGP Convergence Properties. In *Proceedings of ACM SIGCOMM '99 Conference*, G. Parulkar and J. S. Turner, Eds. Boston, 277–288.

Griffin, T. G. and Wilfong, G. 2000. A Safe Path Vector Protocol. In *Proceedings of INFOCOM 2000 Conference.* Tel Aviv, Israel.

Heitmeyer, C., Kirby, J., and Labaw, B. 1998. Applying the SCR Requirements Method to a Weapons Control Panel: An Experience Report. In *Formal Methods in Software Practice.* ACM SIGSOFT.

Hendrick, C. 1988. Routing Information Protocol. RFC 1058, IETF. June.

Holzmann, G. J. 1991. *Design and Validation of Computer Protocols.* Prentice Hall.

Holzmann, G. J. 1997. The SPIN model checker. *IEEE Transactions on Software Engineering 23,* 5 (May), 279–295.

Huitema, C. 1995. *Routing in the Internet.* Prentice Hall.

ISO 8473 1990. *Intermediate System to Intermediate System Intra-Domain Routeing Exchange Protocol for Use in Conjunction with the Protocol for Providing the Connectionless-mode Network Service.* ISO 8473.

Jackson, D., Ng, Y., and Wing, J. 1999. A Nitpick Analysis of Mobile IPv6. *Formal Aspects of Computing 11,* 6 (November), 591–615.

Malkin, G. 1993. RIP Version 2 Carrying Additional Information. RFC 1388. January.

Malkin, G. 1994. RIP Version 2 Carrying Additional Information. RFC 1723, IETF. November.

Manna, Z. and Pnueli, A. 1991. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag.

Mitchell, J. C., Shmatikov, V., and Stern, U. 1998. Finite-State Analysis of SSL 3.0. In *Seventh USENIX Security Symposium.* USENIX, San Antonio, 201–216.

Moy, J. 1994. OSPF Version 2. RFC 1583, IETF. March.

Murthy, S. and Garcia-Luna-Aceves, J. 1996. An Efficient Routing Protocol for Wireless Networks. *ACM Mobile Netowrks and Applications Journal*. Special Issue on Routing in Mobile Communication Networks.

Obradovic, D. 2002. Real-time Model and Convergence Time of BGP. In *Proceedings of IEEE INFOCOM 2002.* New York.

Perkins, C. E. and Bhagwat, P. 1994. Highly Dynamic Destination-Sequenced Distance-Vector Routing (DSDV) for Mobile Computers. *Computer Communications Review*, 234–244.

Perkins, C. E. and Royer, E. M. 1998. Ad Hoc On Demand Distance Vector (AODV) Routing. Internet-Draft Version 2, IETF. March.

Perkins, C. E. and Royer, E. M. 1999. Ad-Hoc On-Demand Distance Vector Routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computer Systems and Applications.* 90–100.

Perlman, R. 1985. An algorithm for distributed computation of spanning trees in an extended LAN. In *Proceedings of the Ninth Data Communications Symposium.* 44–53.

Perlman, R. 1992. *Interconnections: Bridges and Routers.* Addison-Wesley.

Rekhter, Y. and Li, T. 1995. A Border Gateway Protocol 4 (BGP-4). RFC 1771, IETF. March.

Royer, E. M. and Toh, C.-K. 1999. A Review of Current Routing Protocols for Ad Hoc Mobile Wireless Networks. *IEEE Personal Communications*, 46–55.

Varadhan, K., Govindan, R., and Estrin, D. 1996. Persistent Route Oscillations in Inter-Domain Routing. ISI Technical Report 96-631, USC/Information Sciences Institute.

Wang, B.-Y., Meseguer, J., and Gunter, C. A. 2000. Specification and Formal Verification of a PLAN Algorithm in Maude. In *Proceedings of the 2000 ICDCS Workshop on Distributed System Validation and Verification*, T. Lai, Ed. IEEE Computer Society, E:49–E:56.

Received ; revised ; accepted