Formal Verification of Standards for Distance Vector Routing Protocols

Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter
Department of Computer and Information Science
University of Pennsylvania

November 22, 2000

Abstract

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 realtime 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.

Keywords. 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 aided 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 [15, 20]. The classic proof of a 'pure' form of the protocol is given in [2]. Our result covers additional features included in the standard to improve realtime response times (*e.g.* split horizons and poison reverse). These features add additional cases to be considered in the proof, but the automated support reduces the impact of this complexity. Adding these extensions make the theory better match the standard and hence also its implementations. Our proof also uses a different technique from the one in [2] and provides additional properties about network stability.

Our second case study provides a sharp realtime 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 hops 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. We did not find a statement of this result in the literature, but it may be folklore. Our main point is to show how automated support can cover realtime 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 [31], as specified in the second version of the IETF Internet Draft [30]. 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 [31]. 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 [30] 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 is safety critical devices. For instance, [14] 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 implementors 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 [15, 20] 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 conformant implementation and of any network of conformant 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 [15, 20] and the asynchronous distributed protocol proved correct in [2]: 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 [35], including many based on distance vector routing [29, 8, 28, 31]. 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, [26] provides a proof of some key properties of SSL 3.0 handshake protocol [10]. 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 [2], 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 realtime 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

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 [32, 33].

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, 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 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 model-checking system ([17, 18, 36]) 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.

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 1. We then hard-code a

Table 1: 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
}
```

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) [25]. 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 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 [11, 16] 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. 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. The processes can then be considered as functions that take a message as input and describe how the system state is modified. The resulting model is essentially the function in Table 2. Note that the generality

Table 2: State Update Function

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

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 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 specify some lemmas that must be true of the protocol as well, all of which can be easily encoded in Higher-Order Logic.

Lemma 2 *At each node, the leader-id can only decrease over time.* □

Lemma 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 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 and 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 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 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 4 this enables an inductive proof of Theorem 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 *man-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 $k+1$. Also, abstraction proofs in HOL were used to reduce the memory demands of SPIN proofs and assure that the SPIN implementation properly reflected the standard. We give examples of these tradeoffs 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 (with a capital 'I') 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 the paper focuses.

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 [15, 24, 21, 22, 23]. The *Enhanced Interior Gateway Routing Protocol (EIGRP)* (`www.cisco.com/warp/public/103/1.html`) is a distance-vector protocol which is propriety 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 Open Shortest Path First (OSPF) [27]. The fact that routers are provided with global information is useful in determining good routes, but there is significant complexity involved in the protocol that propogates the link states. OSPF is one of the most complex of all RFC's.

BGP [34] 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, Govindan and Estrin [37] demonstrated circumstances in which routes to a given destination *oscillate*. Such behavior is often quite 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 [12] demonstrated that even if the BGP topology of the Internet were known, it would be infeasible in principle to decide whether it might display oscillations. 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 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 [15, 20] and a good exposition can be found in [19]. This subsection gives a brief description of the protocol. Pseudo-code is given in the appendix. 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 advertisment, 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 advertisment 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 quicker loop elimination. RIP exhibits a phenomenon called *counting to infinity*, discussed in [15], which permits the worst case loop persistence 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 routers, 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 rule 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 *poison 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

poison reverse advertisment sets hops to 16 (infinity). A detailed discussion of these two optimizations can be found in [15].

Each route 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 advertisments 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)

This subsection describes the AODV routing protocol whose pseudo-code is given in the appendix. Our analysis is for version 2 of the AODV Internet Draft.

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:

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

Argument hops_to_src determines the current distance from the node which initiated the route request. The initial RREQ has this field set to 0, and every subsequent node increments it by 1. Argument seqno specifies the least sequence number for a route to d that s is willing to accept (node s usually 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.

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 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 (unicast 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 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 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 which 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 src 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 network and next router set the first network and router on some shortest path to d: that is, $D(\mathsf{nextR}(r)) = D(r) - 1$.

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

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 first goal is to show that RIP indeed eventually discovers all the shortest paths of length less than 16:

Theorem 6 (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, 15-stability will be achieved, which is our original goal. Notice that the result applies to an *arbitrary* initial state. This is critical for the fault tolerance aspect of RIP, since it assures convergence even in the presence of topology changes. As long as the changes are not too frequent, we can apply the theorem to the periods in between them.

Our proof, which we call *the radius proof,* differs from the one described in [2] for the asynchronous Bellman-Ford algorithm. Rather than inducting on estimates for upper and lower bounds for distances, we induct on the 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 the asynchrous 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 poison 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 realtime bound on convergence.

Theorem 6 is proved by induction on k. There are four parts to it:

Lemma 7 *The universe $\mathcal{U}$ is initially 1-stable.* □

Lemma 8 (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 9 *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 10 (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 $t \geq t_{k+1}$.* □

4.3 Proof Methodology

Lemma 7 is easily proved by HOL and serves as the basis of the overall induction. Lemma 8 is the fundamental safety property, which we proved both in HOL and in SPIN. To prove the lemma, one needs to show that a k-stable universe remains k-stable after an arbitrary update message. Our HOL proof proceeds by separately verifying that each of the conditions S1 and S2 remain true after an update. This can not 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. In this case, 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.

It is crucial that our abstractions are *finitary* and *property-preserving*. An abstraction is finitary if it reduces to a system with 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. We proved in HOL the fact that our abstraction is property-preserving for the Lemmas 8 and 9. Proofs that can be carried out in either tool are typically done in SPIN, since it provides more automation. However, SPIN can be used only in cases where there is a constant upper bound on the number of states.

Lemma 9, the main progress property in the proof, is proved with SPIN, using the described abstraction. The proof as a whole illustrates well how can verfication 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. [**What do we do with the 'replica generalization' story (commented out)?**]

Lemma 10 is the inductive step, which is derived in HOL as an easy generalization of Lemma 9, considering the fact that the number of routers is finite.

5 Sharp Timing Bounds for RIP Stability

In the previous section we proved convergence for RIP conditioned on the fact 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) $\mathcal{U}$ is $(k-1)$-stable.

(WS2) $\forall r.\ D(r) = k \ \Rightarrow\ (r$ is $k-$stable or $\mathsf{hops}(r) > k)$.

(WS3) $\forall r.\ D(r) > k \ \Rightarrow\ \mathsf{hops}(r) > k$.

Weak k-stability is stronger than $(k-1)$-stability, but weaker than k-stability. 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 11 (Preservation of weak stability) *For any $2 \le k \le 15$, if the universe is weakly k-stable at some time t, then it is weakly k-stable at any time $t' \ge t$.* □

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

Lemma 12 (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 13 *For any $2 \le k \le 15$, 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 14 (Progress) *For any $2 \le k < 15$, 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 15 (RIP convergence time) *A universe of radius R becomes 15-stable within $max\{15, R\} \cdot \Delta$ 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 $R \le 15$. 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 12)
after $2 \cdot \Delta$	weakly 3-stable	(by Lemma 14)
after $3 \cdot \Delta$	weakly 4-stable	(by Lemma 14)
...	...	...
after $(R-1) \cdot \Delta$	weakly R-stable	(by Lemma 14)
after $R \cdot \Delta$	R-stable	(by Lemma 13)

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

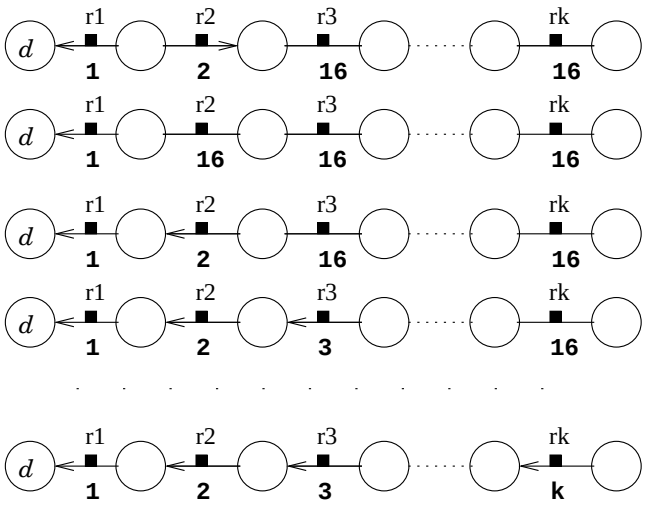

Figure 1: Maximum Convergence Time

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 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 Methodology

Lemmas 11, 12, and 14 are proved in SPIN (Lemma 13 is a consequence of Lemma 14). The proofs are carried out using an abstraction similar to the one mentioned in Section 4, appropriately tuned for the weak stability instead of the ordinary stability. Theorem 15 is then derived as a corollary in HOL. SPIN turned out to be extremely helpful for proving properties such as Lemma 14, 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 13, 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 subcases 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 subcases 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 generate even some sequences that are not possible in reality. Such abstractions will still be property-preserving for *universal* 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 14, 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.

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. In [31], Perkins and

Royer give 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 7, that SPIN can analyze.

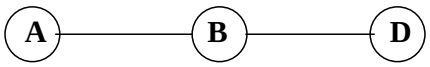

Figure 2: Sample 3-node Network

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 and verify using SPIN that the 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.

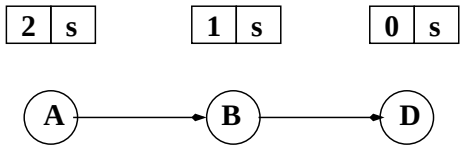

Figure 3: Initial Routes

S1. When the link B—D goes down, B generates a RREP with hop count infinity and increments its seqno 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.

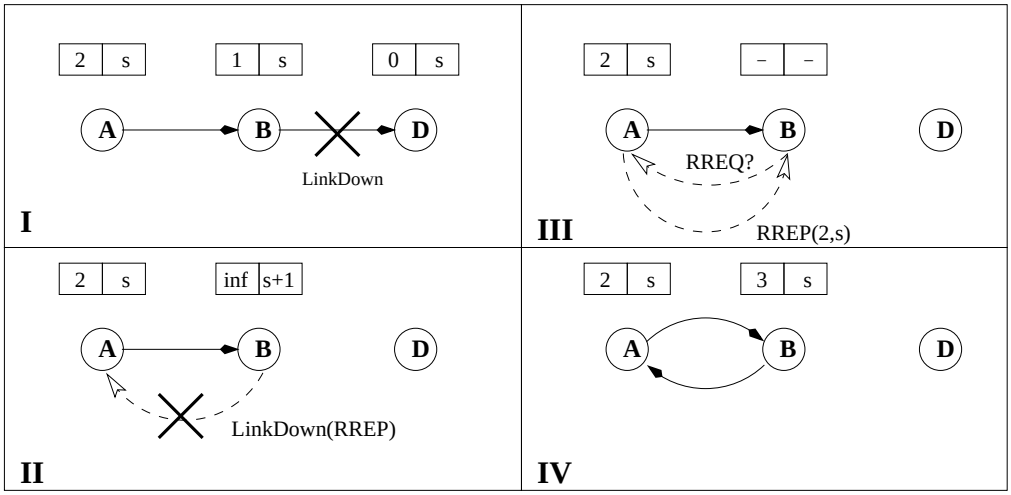

Figure 4: Loop Condition S1

S2. Suppose B's route expires while A is still pointing at 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.

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

c. Supposed, B keeps the route around as an expired route, it increments the route's seqno 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.

d. Finally, suppose B keeps the route around as an expired route, it increments the route's seqno, 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.

S3. 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.

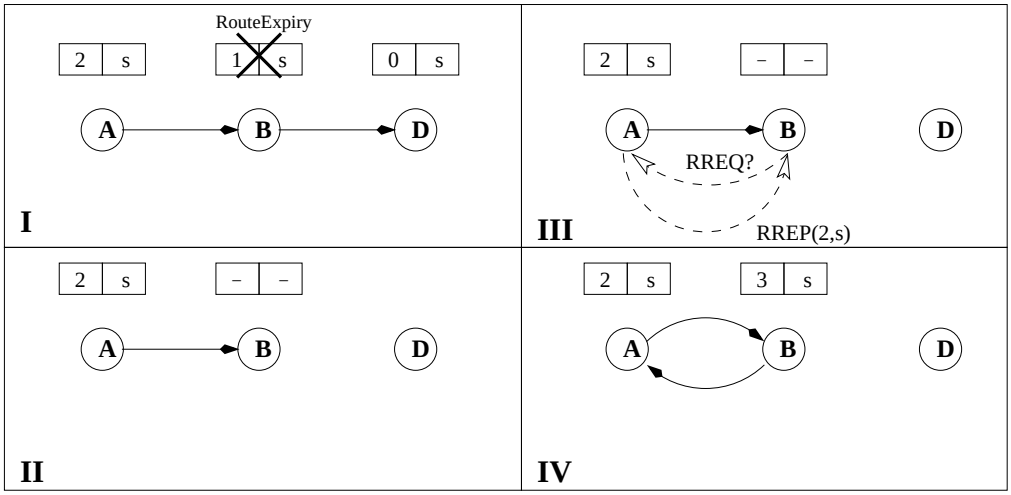

Figure 5: Loop Condition S2(a)

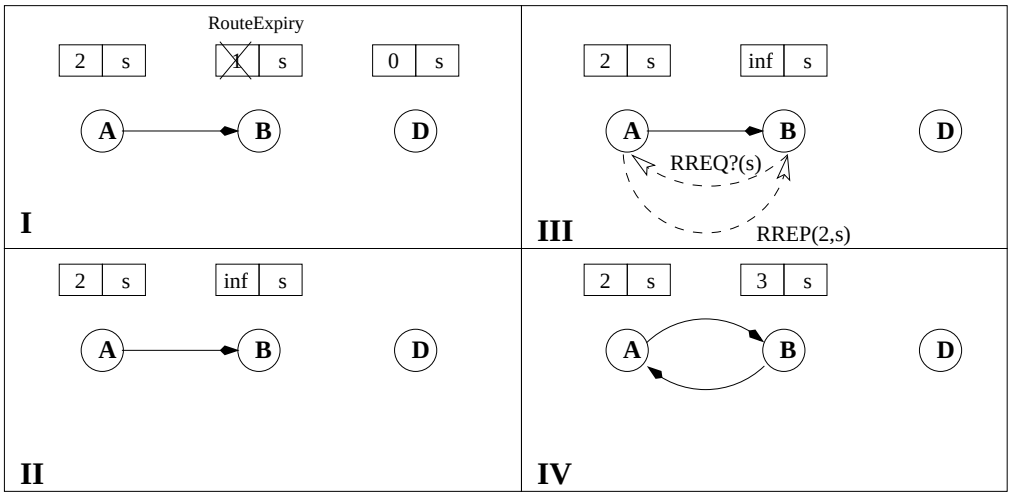

Figure 6: Loop Condition S2(b)

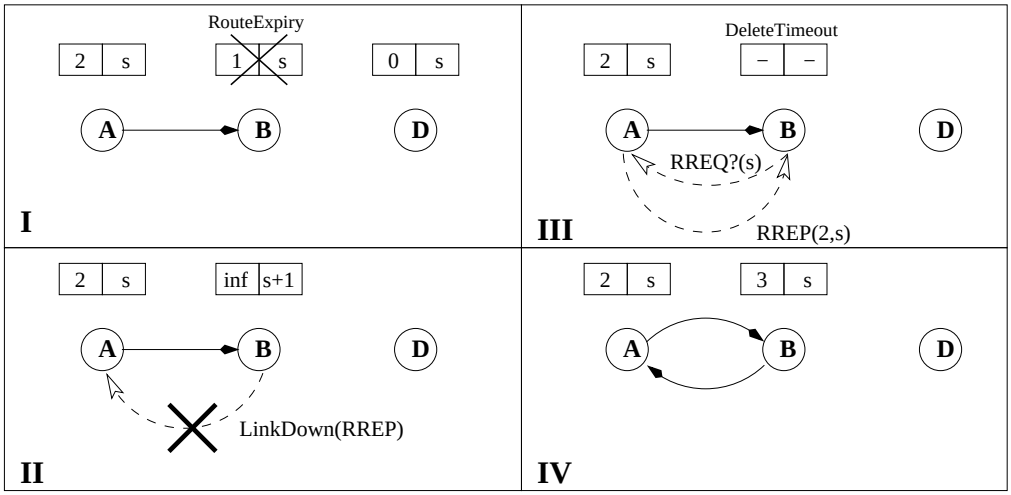

Figure 7: Loop Condition S2(c)

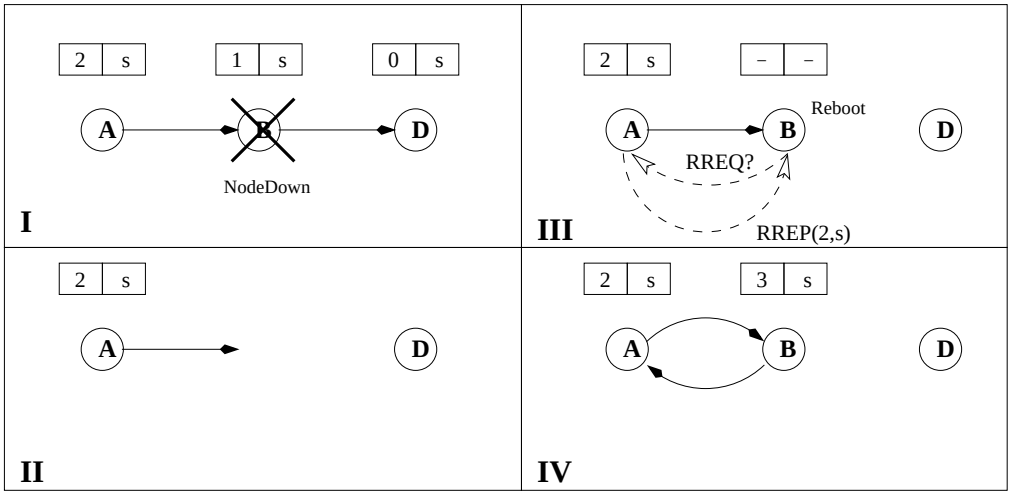

Figure 8: Loop Condition S3

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.

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

6.2 Ambiguities in 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. We attribute this failing to an incomplete understanding of the invariants that ensure loop-freedom for AODV. Each of the scenarios points to a significant flaw in the standard. In the next section, we propose some fixes to the standard, and an invariant-based proof of loop-freedom for the fixed standard.

However, there are other looping scenarios in which the fault is in the under-specification or ambiguity in the language of the standard. We have not included these scenarios in the preceding section, because it can be argued that in these cases the intent of the standard is intuitively clear. We describe three such ambiguities below.

- 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 have some *default* routes, that the process will start with, then SPIN can demonstrate that there will be loops.

- The event handler for the reception of RREP packets is not described in the standard. Again, we can fill this in based on our intuitive knowledge of how the protocol works, but an incorrectly written RREP handler could easily cause loops.

- 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 has since been fixed in later versions, and we believe that it was always intended that the AODV node would increment its stored sequence number as well.

However, this omission has had some interesting consequences. When AODV was revised to version 5, the authors chose to use a new kind of packet (RERR) to denote route errors, to remove some confusion in the handling of the RREPs. But in addition, they felt they no longer had to increment sequence numbers in the outgoing RERR packets. This error lead to a looping scenario that was discovered by Madanlal Musuvathi using Murϕ. We believe that the incompleteness of the standard, version 2, in this section, may have led to these incorrect revisions in version 5.

To remove these ambiguities, we change the standard pseudo-code described in Appendix 7. We make the first three modifications as described in Apendix 7, and it is this modified pseudo-code that we model in SPIN and use for the analysis described in Section 6.1.

6.3 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 seqno for the route.

A2. Nodes *never* delete routes.

A3. Nodes *always* 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 acordance with these assumptions, and the result is shown as the fourth modification to the pseudo-code described in Apendix 7. This modification guarantees assumptions A1 and A2. A3 is an environmental assumption and is not reflected in the code. Hereafter, we shall analyze this modified version of the AODV pseudo-code.

Theorem 16 *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 S2(b). Assumption A2 avoids the scenarios S1 and S2(a,c). Finally, A3 avoids the scenario S3.

As mentioned before, a hand proof of AODV loop-freedom is given in [31]. That proof does not take into account many details of AODV like route expiry. We provide a complete automated proof of Theorem 16 using the SPIN model-checker and HOL theorem-prover. Moreover, the proof in [31] 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. Similarly for hops and next. In non-temporal properties we omit the time argument, understanding that we are talking about current values at some given time.

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

Theorem 17 *If* $\mathsf{next}_d(n) = n'$, *then*

1. $\mathsf{seqno}_d(n) \leq \mathsf{seqno}_d(n')$, *and*
2. $\mathsf{seqno}_d(n) = \mathsf{seqno}_d(n') \Rightarrow \mathsf{hops}_d(n) > \mathsf{hops}_d(n')$. □

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:

1. (Loop-Freedom) Consider the network at any instant and look at all the routing-table entries for a destination d. Any data packet traveling towards d would have to move along the path defined by the next_d pointers. However, we know from Theorem 17 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.

2. (Route Validity) Loop-freedom in a finite network guarantees that data paths are finite. This does not guarantee that the path ends at d. However, if all the sequence numbers along a path are the same, hop-counts must strictly decrease (by Theorem 17). In particular, the last node n_l on the path cannot have hop-count INFINITY (no route). But since n_l does not have a route to d, it must be equal to d.

To prove Theorem 17, we first prove the following properties about the routing table at each node n, now considered as a function of time.

Lemma 18 *If* $t_1 \leq t_2$, *then*
$\mathsf{seqno}_d(n)(t_1) \leq \mathsf{seqno}_d(n)(t_2)$. □

Lemma 19 *If* $t_1 \leq t_2$ *and*
$\mathsf{seqno}_d(n)(t_1) = \mathsf{seqno}_d(n)(t_2)$, *then*
$\mathsf{hops}_d(n)(t_1) \geq \mathsf{hops}_d(n)(t_2)$. □

Intuitively, Lemma 18 states that the sequence number for a single destination never decreases over time. Lemma 19 says that if the sequence number stays unchanged over some period of time, then the hop-count can only improve or remain the same during that time.

Suppose $\mathsf{next}_d(n)(t) = n'$. Then we define lut (last update time), to be the last time before t, when $\mathsf{next}_d(n)$ *changed* to n'. We claim that following lemma holds for times t and lut:

Lemma 20 *If* $\mathsf{next}_d(n)(t) = n'$, *then*

1. $\mathsf{seqno}_d(n)(t) = \mathsf{seqno}_d(n')(\mathsf{lut})$, *and*
2. $\mathsf{hops}_d(n)(t) = 1 + \mathsf{hops}_d(n')(\mathsf{lut})$. □

The lemma essentially describes the way node n updated its routing table at time lut to point to n'.

It is not hard to see that the three lemmas together imply Theorem 17. First, lemmas 18 and 20 applied to lut and t yield

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

which is the first part of Theorem 17. Furthermore, if the equality holds above, then we have

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

because of lemmas 19 and 20. This shows that $\mathsf{hops}_d(n)(t) > \mathsf{hops}_d(n')(t)$, which is the second part of Theorem 17. This suffices to guarantee the loop freedom (Theorem 16).

6.4 Proof Methodology

We model AODV in SPIN by a Promela process for each node. As described earlier, each process needs to maintain state in the form of a broadcast-id, a sequence number and a routing table. The process needs to react to several events, possibly updating this state. The main events are neighbor discovery, data or control (RREP/RREQ) packet arrival and timeout events like link failure detection and route expiration. It is relatively straight-forward to generate the Promela processes from [30]. Pseudo-code for the AODV process is given in the Appendix.

In order to use SPIN to prove loop freedom under the conditions A1-A3, we first needed to reduce the problem to a finite-state verification. In the case of RIP, we did it by constructing a finitary property-preserving abstraction of the system and then proving the *original* property on the abstracted system. In the case of AODV, we take a different approach. We reduce the *property* (and not the model) to an invariant that holds on certain pairs of nodes. By doing so, we reduced an unbounded n-node verification to a 2-node verification. However, this verification still contains an infinite state space, because of the unbounded range of sequence numbers.[1] We solved this problem by introducing an abstraction for sequence numbers at every node. Instead of working with actual constants, we only consider whether an advertised sequence number is smaller, equal or larger than the current one.

[1] An actual implementation of the protocol is likely to have some bound on on the size of sequence numbers. However, even "reasonable" bounds would still introduce prohibitive complexity in the verification.

Now we proved all three lemmas in SPIN. Each verification involves at most two AODV processes reacting to events produced by an environment of AODV routers. Lemmas 18 and 19 can be proved without restrictions on the events produced by the environment. Lemma 20 is trickier and requires the model to record the incoming $\mathsf{seqno}_d(n')$ and $\mathsf{hops}_d(n')$ whenever the protocol decides to change $\mathsf{next}_d(n)$ to n'. This is easily done by the addition of two variables. Subsequently, Lemma 20 is also verified by SPIN.

Finally, the proof that the three lemmas together imply Theorem 17 involves standard deductive reasoning, outlined in the previous subsection. That part is done in the HOL theorem prover.

6.5 Alternative Approaches

We have proposed to fix the AODV standard in this section, along the lines of the assumptions A1-A3. We then prove that the fixed standard, as shown in Appendix 7 is loop-free.However, there are other ways to fix the standard without making strong assumptions like A2 (*Nodes never delete routes*), which may be impractical for real protocols.

The AODV standard has since been revised to version 6, and from version 5, it contains alternative fixes, proposed by us, for the errors that we have found. These fixes consist of corrections to the text to guarantee A1 and recommendations for timer values that guarantee weaker properties than A2 and A3, which are still adequate for loop-freedom. These fixes are implementationally different from those described in this paper, but try to achieve the same logical behavior, using subtle relationships between timers. A formal analysis of AODV, version 5, would require powerful tools that can analyze real-time behavior. We believe that SPIN, as it currently stands, is not adequate for this task.

7 Conclusion

This paper provides the most extensive automated mathematical analysis of a class of routing protocols to date. 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 realtime bound on the convergence of RIP, and an automated proof of loop-freedom for AODV.

Table 3 summarizes some of our experience with the complexity of the proofs in terms of our automated support tools. The complexity of an HOL verification 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

Table 3: Protocol Verification Effort

Task	HOL	SPIN
Modeling RIP	495 lines, 19 defs, 20 lemmas	141 lines
Proving Lemma 8 Once	9 lemmas, 119 cases, 903 steps	
Proving Lemma 8 Again	29 lemmas, 102 cases, 565 steps	207 lines, 439 states
Proving Lemma 9	Reuse Lemma 8 Abstractions	285 lines, 7116 states
Proving Lemma 11	Reuse Lemma 8 Abstractions	216 lines, 1019 states
Proving Lemma 12	Reuse Lemma 8 Abstractions	221 lines, 1139 states
Proving Lemma 14	Reuse Lemma 8 Abstractions	342 lines, 21804 states
Modeling AODV	95 lines, 6 defs	302 lines
Proving Lemma 18		173 lines, 5106 states
Proving Lemma 19		173 lines, 5106 states
Proving Lemma 20		157 lines, 721668 states
Proving Theorem 17	4 lemmas, 2 cases, 5 steps	

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. Most of the lemmas consumed the SPIN-minimum of 2.54MB of memory; Lemma 20 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 8, the Stability Preservation Lemma, using HOL only and HOL together with SPIN.

Extensions of the results in this paper are possible in several areas: additional protocols, better tool support, and techniques for proving other kinds of properties. We could prove correctness properties similar to the ones in this paper for other routing protocols, including other approaches like link state routing [1]. It would be challenging to prove OSPF [27] because of the complexity of the protocol specification, but the techniques used here would apply for much of what needs to be done. We do intend to pursue better tool support. In particular, we are interested in integration with simulation and implementation code ([3]). This may allow us to leverage several related activities and also improve the conformance between the key artifacts: the standard, the SPIN program, the HOL invariants and environment model, the simulation code, and, of course, the implementation itself. For instance, one can analyze the conformance between an implementation and a specification by simultaneously running them

on the same input sequence and observing the output. A tool like SCR* [13] can be used for that purpose, since it has a built-in support for simulating the specification. One program analysis technology of particular interest is *slicing*, since it is important to know which parts of a program might affect the values in messages. We are also interested in how to prove additional kinds of properties such as security and quality of service (including reservation assurances). Security is particularly challenging because of the difficulty in modeling secrecy precisely.

Due to the difficulties in adapting unbounded or infinite state verification to finite state verification tools, there has been relatively few attempts at verifying routing protocols. Cypher et al. in [9] describe their verification of an ATM network routing protocol PNNI. Having SPIN as the verification tool and Promela as the specification language, they were essentially forced to perform instance verification, since the protocol could not be specified in Promela in full generality. We have addressed this problem by using a combination of model checking and theorem proving [6]. In situations where full verification seems infeasible, one can instead concentrate on finding errors through different kinds of testing. A toolset for logical testing of network simulations is described in [3]. A classification of logical testing techniques is presented in [7]. 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. This question is studied in [5]. A broader survey of tool-specific issues for specification, verification and testing of routing protocols can be found in [4].

CITE Jeanette Wing's nitpick detection of loops in mobile ip!!!

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, Charles Perkins, and Jonathan Smith. This research was supported by NSF Contract CCR-9505469, and DARPA Contract F30602-98-2-0198.

Appendix

We provide pseudo-code for the RIP and AODV protocols in this appendix. Pseudo-code is classified into six groups. *Constants* section lists some fixed or locally configured constants that the routing process uses. *State* provides 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* list the events that the routing process recognizes. *Utility functions* describe functions that the routing process can invoke; these may cause events recognized by other routing processes. *Event handlers* describe how the events recognized by the process are dealt with. Events and their handlers generally fall into two categories: reciept 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". They are a special kind of variables that coutinuously decrease their 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 $[logical\ contents; IP\_DEST]$.

RIP Pseudo-code

process RIPRouter

state:

me	// ID of the router
interfaces	// Set of router's interfaces
known	// Set of destinations with known routes
$hops_{dest}$	// Distance estimate
$nextRouter_{dest}$	// Next router on the way to *dest*
$nextIface_{dest}$	// Interface over which the route advertisment was received
timer $expire_{dest}$	// Expiration timer for the route
timer $garbageCollect_{dest}$	// Garbage collection timer for the route
timer *advertise*	// Timer for periodic advertisments

initially:

{
  known ← the set of all networks to which the router is connected.
  for *dest* ∈ *known*
  {
    $hops_{dest} = 1$
    $nextRouter_{dest} = me$
    $nextIface_{dest}$ = the interface which 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 ← min (1 + hopCnt, 16)
    if (dest ∉ known) then
    {
      if (newMetric < 16)
      {
        hops_dest ← newMetric
        nextRouter_dest ← router
        nextIface_dest ← iface
        set expire_dest to 180 seconds
        known ← known ∪ {dest}
      }
    } else
    {
      if (router = nextRouter_dest) or (newMetric < hops_dest)
      {
        hops_dest ← newMetric
        nextRouter_dest ← router
        nextIface_dest ← iface
        set expire_dest to 180 seconds
        if (newMetric = 16) then
        {
          set garbageCollect_dest to 120 seconds
        } else
        {
          deactivate garbageCollect_dest
        }
      }
    }
  }

  timeout (expire_dest)
  {
    hops_dest ← 16
    set garbageCollect_dest to 120 seconds
  }

  timeout (garbageCollect_dest)
  {
    known ← known − {dest}
  }
```

```
timeout (advertise)
{
  for each dest ∈ known do
    for each i ∈ interfaces do
    {
      if (i = 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
}
```

AODVv2 Pseudo-code

process AODVRouter

constants:

RREP_WAIT_TIME	// Set as described in the standard.
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
sn_d	// Destination sequence number
$hops_d$	// Distance in hops
$next_d$	// Next hop
neighbors	// Set of all neighbors
$active_d$	// Set of active neighbors
timer $lifetime_d$	// Route expiration timer
timer $activeTimer_{dest,n}$	// Active neighbor timer

events:

receive RREQ(hopCnt, bcastID, dest, destSeqno, source, sourceSeqno) from sender	
receive RREP(hopCnt, dest, destSeqno, lifetime); IP_DEST from sender	
receive NChange	// Triggered when the set of neighbors change
receive Packet; IP_DEST from sender	// Triggered when a the router receives a packet to forward to IP_DEST
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 expiration timers 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 $seen(source, bcastID)$
{
$hopCnt \leftarrow hopCnt + 1$
if $(dest = me)$ **then**
{
$updateRoute\ (source, sourceSeqno, hopCnt, sender, ACTIVE\_ROUTE\_TIMEOUT)$
$neighborcast\ ([RREP(0, me, max(mySeqno, destSeqno), MY\_ROUTE\_TIMEOUT); source], next_{source})$
} **else**
{
$updateRoute\ (source, sourceSeqno, hopCnt, sender, max(REV\_ROUTE\_LIFE, lifetime_{dest}))$
if $(dest \in known)$ **and** $(hops_{dest} < \infty)$ **and** $(sn_{dest} \geq destSeqno)$ **then**
{
$neighborcast\ ([RREP(hops_{dest}, dest, sn_{dest}, lifetime_{dest}); source], next_{source})$
$n \leftarrow next_{dest}$
$active_{source} \leftarrow active_{source} \cup \{n\}$

```
        set activeTimer_{source,n} to ACTIVE_ROUTE_TIMEOUT
      } else
      {
        broadcast ([RREQ(hopCnt, bcastID, dest, destSeqno, source, sourceSeqno)])
      }
    }
  }
}

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

receive NChange
{
  newNeighbors ← computeNeighbors ()
  disconnected ← neighbors − newNeighbors
  neighbors ← newNeighbors
  mySeqno ← mySeqno + 1
  for dest ∈ known
  {
    if (next_{dest} ∈ disconnected)
    {
      for n ∈ active_{dest}
      {
        neighborcast ([RREP(∞, dest, 1 + sn_{dest}, BAD_LINK_LIFETIME); n], n)
      }
    }
  }
  updateTable ()
}

receive Packet; IP_DEST from sender
{
  if (IP_DEST ≠ me)
  {
    if (IP_DEST ∈ known) then
    {
      active_{IP_DEST} ← active_{IP_DEST} ∪ {sender}
      set lifetime_{IP_DEST} to ACTIVE_ROUTE_TIMEOUT
      set activeTimer_{IP_DEST,sender} to ACTIVE_ROUTE_TIMEOUT
      neighborcast ([Packet; IP_DEST], next_{IP_DEST})    // Forward the packet towards IP_DEST
    } else
    {
      myBcastID ← myBcastID + 1
      broadcast ([RREQ (0, myBcastID, IP_DEST, sn_{IP_DEST}, me, mySeqno)])
      Queue the packet and forward it upon establishing a route to IP_DEST.
    }
```

```
  }
}

timeout (lifetime_dest)
{
  if (hops_dest = ∞) then
  {
    Mark entry for dest as "erasable". Erasable entries can be garbage collected.
    Garbage collecting sets sn_dest to 0, hops_dest and next_dest to some undefined value.
  } else
  {
    hops_dest ← ∞
    known ← known − {dest}
    set lifetime_dest to BAD_LINK_LIFETIME
  }
}

timeout (activeTimer_dest,n)
{
  active_dest ← active_dest − {n}
}
```

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 6.2. 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 6.3.

1. We include the initialization section.

   ```
   initially:
   {
     mySeqno ← 0
     myBcastID ← 0
     known ← ∅
     ∀d. sn_d ← 0
   }
   ```

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); IP_DEST from sender
   {
     if (IP_DEST = me) then
     {
   ```

```
    if (hopCnt = ∞) then
    {
      updateRoute (dest, destSeqno, ∞, sender, BAD_LINK_LIFETIME)
      for n ∈ active_dest
      {
        neighborcast ([RREP (∞, dest, destSeqno, BAD_LINK_LIFETIME); n], n)
      }
    }
    else
    {
      updateRoute (dest, destSeqno, hopCnt, sender, lifetime)
    }
  }
  else
  {
    updateRoute (dest, destSeqno, hopCnt, sender, lifetime)
    neighborcast ([RREP(hopCnt + 1, dest, destSeqno, lifetime); IP_DEST], next_IP_DEST)
  }
}
```

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. Shaded part is the addition.

```
receive NChange
{
  newNeighbors ← computeNeighbors ()
  disconnected ← neighbors − newNeighbors
  neighbors ← newNeighbors
  mySeqno ← mySeqno + 1
  for dest ∈ known
  {
    if (next_dest ∈ disconnected)
    {
      for n ∈ active_dest
      {
        neighborcast ([RREP(∞, dest, 1 + sn_dest, BAD_LINK_LIFETIME); n], n)
        sn_dest ← sn_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.

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

References

[1] Bhargav Bellur and Richard G. Ogier. A reliable, efficient topology broadcast protocol for dynamic networks. In *IEEE Infocomm*. IEEE, 1999.

[2] Dimitri P. Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, 1991.

[3] Karthikeyan Bhargavan, Carl A. Gunter, Moonjoo Kim, Insup Lee, Davor Obradovic, Oleg Sokolsky, and Mahesh Viswanathan. Verisim: Formal analysis of network simulations. In *International Symposium on Software Testing and Analysis (ISSTA)*, Portland, OR, August 2000.

[4] Karthikeyan Bhargavan, Carl A. Gunter, and Davor Obradovic. An assessment of tools used in the verinet project. Technical Report MS-CIS-00-15, University of Pennsylvania, 2000.

[5] Karthikeyan Bhargavan, Carl A. Gunter, and Davor Obradovic. Fault origin adjudication. In *Formal Methods in Software Practice (FMSP)*, Portland, OR, August 2000.

[6] Karthikeyan Bhargavan, Carl A. Gunter, and Davor Obradovic. RIP in SPIN/HOL. In *Theorem Proving in Higher-Order Logics (TPHOLs)*, Portland, OR, August 2000.

[7] Karthikeyan Bhargavan, Carl A. Gunter, and Davor Obradovic. A taxonomy of logical network analysis techniques. Technical Report MS-CIS-00-14, University of Pennsylvania, 2000.

[8] C.-C. Chiang. Routing in clustered multihop, mobile wireless networks with fading channel. In *Proceedings of IEEE SICON '97*, pages 197–211, April 1997.

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

[10] Alan O. Freier, Philip Karlton, and Paul C. Kocher. *Secure Socket Layer*. IETF Draft, November 1996. `home.netscape.com/ eng/ ssl3`.

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

[12] Timothy G. Griffin and Gordon Wilfong. An analysis of BGP convergence properties. In Guru Parulkar and Jonathan S. Turner, editors, *Proceedings of ACM SIGCOMM '99 Conference*, pages 277–288, Boston, August 1999.

[13] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proceedings of the 10th Annual IEEE Conference on COMPuter ASSurance (COMPASS)*, 1995.

[14] Constance Heitmeyer, James Kirby, and Bruce Labaw. Applying the SCR requirements method to a weapons control panel: An experience report. In *Formal Methods in Software Practice*. ACM SIGSOFT, March 1998.

[15] C. Hendrick. Routing information protocol. RFC 1058, IETF, June 1988.

[16] Home page for the HOL interactive theorem proving system. `www.cl.cam.ac.uk/ Research/ HVG/ HOL`.

[17] Gerard J. Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[18] Gerard J. Holzmann. The SPIN model checker. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[19] Christian Huitema. *Routing in the Internet*. Prentice Hall, 1995.

[20] G. Malkin. *RIP Version 2 Carrying Additional Information*. IETF RFC 1388, January 1993.

[21] G. Malkin. Rip version 2 applicability statement. RFC 1722, IETF, November 1994.

[22] G. Malkin. Rip version 2 carrying additional information. RFC 1723, IETF, November 1994.

[23] G. Malkin. Rip version 2 MIB extension. RFC 1724, IETF, November 1994.

[24] G. Malkin. Rip version 2 protocol analysis. RFC 1721, IETF, November 1994.

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

[26] J. C. Mitchell, V. Shmatikov, and U. Stern. Finite-state analysis of SSL 3.0. In *Seventh USENIX Security Symposium*, pages 201–216. USENIX, San Antonio, 1998.

[27] J. Moy. OSPF version 2. RFC 1583, IETF, March 1994.

[28] Shree Murthy and J.J. Garcia-Luna-Aceves. An efficient routing protocol for wireless networks. *ACM Mobile Netowrks and Applications Journal*, October 1996. Special Issue on Routing in Mobile Communication Networks.

[29] Charles E. Perkins and Pravin Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. *Computer Communications Review*, pages 234–244, October 1994.

[30] Charles E. Perkins and Elizabeth M. Royer. Ad hoc on demand distance vector (AODV) routing. Internet-Draft Version 2, IETF, March 1998.

[31] Charles E. Perkins and Elizabeth M. Royer. Ad-hoc on-demand distance vector routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computer Systems and Applications*, pages 90–100, February 1999.

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

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

[34] Y. Rekhter and T. Li. A border gateway protocol 4 (BGP-4). RFC 1771, IETF, March 1995.

[35] Elizabeth M. Royer and Chai-Keong Toh. A review of current routing protocols for ad hoc mobile wireless networks. *IEEE Personal Communications*, pages 46–55, April 1999.

[36] Home page for the SPIN model checker. `http://netlib.bell-labs.com/netlib/spin/whatispin.html`.

[37] K. Varadhan, R. Govindan, and D. Estrin. Persistent route oscillations in inter-domain routing. ISI Technical Report 96-631, USC/Information Sciences Institute, 1996.