# Verisim: Formal Analysis of Network Simulations

Karthikeyan Bhargavan, Carl A. Gunter, Moonjoo Kim, Insup Lee,
Davor Obradovic, Oleg Sokolsky, and Mahesh Viswanathan
Department of Computer and Information Science
University of Pennsylvania
{bkarthik,gunter,moonjoo,lee,davor,sokolsky,maheshv}@saul.cis.upenn.edu

**Abstract**

Network protocols are often analyzed using simulations. We demonstrate how to extend such simulations to check propositions expressing safety properties of network event traces in an extended form of linear temporal logic. Our technique uses the NS simulator together with a component of the MaC system to provide a uniform framework. We demonstrate its effectiveness by analyzing simulations of the Ad Hoc On-Demand Distance Vector (AODV) routing protocol for packet radio networks. Our analysis finds violations of significant properties, and we discuss the faults that cause them. Novel aspects of our approach include modest integration costs with other simulation objectives such as performance evaluation, greatly increased flexibility in specifying properties to be checked, and techniques for analyzing complex traces of alarms raised by the monitoring software.
**Keywords:** Verisim, Formal Analysis, Network, Simulation, Testing, Routing, NS, MaC, AODV, Temporal Logic, Ad Hoc Networks, Packet Radio, Tuning, Population Abstraction, Packet-Type Abstraction.

## 1 Introduction

Network protocols such as routing protocols are difficult to test because meaningful experiments may involve dozens or even thousands of hosts and routers. Developing an adequate testbed would be prohibitively expensive while experiments involving operational systems may be too risky or inconvenient. Thus simulations are widely used as a testing technique for both performance and correctness properties.

The need for validating protocol implementations in simulators has been well recognized. Not only could an improper implementation of the given protocol lead to incorrect simulation results, but if it becomes a part of the simulation suite, it could lead to incorrect results for other protocols simulated with it. Network simulators come with validation test suites for most of the core protocols, so that modified versions of these protocols can be validated to have the same properties. These tests compare the performance of a modified protocol with a pre-computed expected performance chart for the scenario.

There are at least three ways in which testing based only on performance measures is less than one would like for careful analysis of a protocol: such an analysis may not be able to detect certain kinds of bugs in the simulator code, it is desirable to have more support for finding flaws in the protocol itself, and there are flaws of interest that are not immediately manifested as performance problems. Let us consider each of these briefly.

Simulator code can be buggy. An inherent assumption in the validation tests is that any significant bug will show up as a performance degradation, but this need not be true. In particular, a bug may simply alter the overall performance profile. If the aim of the simulation is to find the right parameters to include in the standard specification of the protocol, these parameters may be incorrect because they were learned from a simulation that was incorrectly coded. In particular, there may be poorer-than-expected performance from a deployed system if it implements the protocol properly. Assuming this is even discovered, it may be very painful to reconcile the differences and find the proper parameters, especially if they have been set in stone by the standard.

Suppose the protocol has a design flaw that causes bad performance figures during simulation. The performance figures alone may give only limited information about the nature of the flaw. For a complex protocol that interacts with many other protocols fuller diagnostic information would be invaluable. Current practice involves searching for the flaw by repeated runs of the simulation as informed by manual inspection of the packet trace or processing by ad hoc shell scripts. A structured, logical framework for discovering these flaws can facilitate such interactive discovery.

There are some properties of protocols that do not relate directly to performance. Suppose that a routing protocol also has a security requirement that a packet at a node $n_1$ meant for a neighboring node $n_2$ will never be seen by a third node $n_3$. If this property is violated, the hit on performance is likely to be very small but one would still like to know if the property is violated in any of the simulated scenarios. Even if one is only concerned with performance, there are correctness properties that will impact performance in important circumstances. It may be easier to find these flaws by searching for non-performance-affecting violations rather than by creating scenarios in which these flaws actually cause performance problems. For instance, routing loops can degrade performance, but may also occur without significant impact on performance. Since they are not expected to happen, their occurrence in a simulation would be of interest, even if they did not impact performance in that particular scenario.

In this paper we describe a tool suite called *Verisim* which facilitates the analysis of correctness properties in network simulations. The advantage of Verisim comes from its combination of a popular network simulator tool, NS [13], with the flexible trace-checking component of the MaC system [17]. Traces are generated using NS (version 2) and analyzed to determine whether they satisfy desired properties. These properties are expressed using the Meta-Event Definition Language (MEDL) used in MaC. MEDL is an expressive language extending Linear Temporal Logic; it is able to express a variety of important safety properties of the kind network software is expected to satisfy. With this combination, it is possible to seamlessly integrate flexible testing of such properties into the processes generally used to design and analyze network systems.

We provide an overview of the MaC framework for system analysis, describe its instantiation in Verisim, and illustrate the application of Verisim to existing simulation code for Ad Hoc On-Demand Distance Vector routing (AODV), a protocol for routing in *ad hoc* packet radio networks. Our case study has two parts, based on code we obtained from the Monarch Group at Carnegie-Mellon. The first part illustrates a basic approach for using Verisim to find and correct bugs in the simulation code; the second part shows how the flexibility of Verisim can reduce turn-around time in debugging.

In the first part of our case study we run an NS simulation and create a trace $T$ which is analyzed for properties AODV is expected to satisfy. The properties are expressed by a MEDL formula $\phi$ and the checker produces as its output a *metatrace* $T^{\phi}$ of alarms indicating violations of $\phi$ by the given trace. Our study revealed several bugs in the simulation code, and we use Verisim

to locate each of these and carry out regression testing until they are all removed. The technique is what we call *Repair First Bug (RFB)*. RFB proceeds by taking the trace and analyzing the first alarm to determine what may have caused it. Assuming that the formula $\phi$ is properly expressed, this represents a bug in the simulation code. This bug is repaired and the newly modified program $P_1$ is again run through the simulation to produce an output trace $T_1$, which is again examined to find a second bug. Assuming that three bugs are found, this process generates a program $P_3$ which satisfies the property $\phi$ in the tested scenario. In all, this debugging session required three runs of the simulation.

In the second part of our case study we illustrate how the flexibility of Verisim can be exploited to improve turn-around time for debugging. In this study we attempt to avoid some of steps where the simulation was rerun to generate a new trace for continued debugging. The situation is similar to what one sees in compilers, where an effort is made to produce error messages that are as independent as possible in hope that the several faults in the program can be removed before the compilation needs to be repeated. This is especially useful for simulations, which may run for long periods of time (even days), and where analysis may generate vast, incomprehensible metatraces of alarms. Alarms represent bugs that must be repaired, and it is necessary to repair as many as possible before rerunning the simulation. The automated techniques used by compilers are largely inapplicable since errors generated by routing protocols are quite different in nature. We focus on a mixture of manual and automated techniques we call *tuning*. The metatrace $T^{\phi}$ is manually inspected to find bug classes and then the MEDL property $\phi$ is modified or 'tuned' to produce a formula $\psi$ that ignores one or more bugs recognized in this first manual analysis. Verisim then re-analyzes the *original* $T$ to produce a new metatrace $T^{\psi}$, which is inspected for new bugs. Note that the second step can proceed without rerunning the simulation. This strategy is repeatedly applied until it becomes desirable to fix a collection of bugs and rerun the simulation.

The paper is divided into seven sections. After this introduction we describe Verisim and its components, the MaC framework and the NS system. Then, in the third section, we describe the AODV protocol. Simulator code for this protocol is then analyzed in two case studies in the fourth section. The fifth section describes abstraction techniques that can be used to improve the scalability of checking. The sixth section discusses some of the related work, and the seventh section concludes.

## 2 Verisim

The need to analyze protocol simulation results beyond performance measures led us to design an integrated environment for protocol simulation and analysis. The environment, which we call Verisim, enables us to perform simulations and explore their properties within the same framework. Rather than developing Verisim components from scratch, we envisioned Verisim as a collection of tools that have been tried by researchers and developers and proved to be useful in similar contexts.

We designed Verisim within the MaC monitoring and checking framework [18]. The MaC framework appeared to be a natural choice for the toolset architecture. MaC is designed for the formal analysis of trace-based executions with respect to user-specified properties. This setup exactly matches our needs regarding simulation trace analysis. We designed Verisim as an integrated system that combines the network simulator NS [13] and the checker from an earlier MaC-based monitoring system for Java programs [17]. The individual tools are described below. A simulation trace produced by NS is automatically transformed into the input format of the MaC checker.

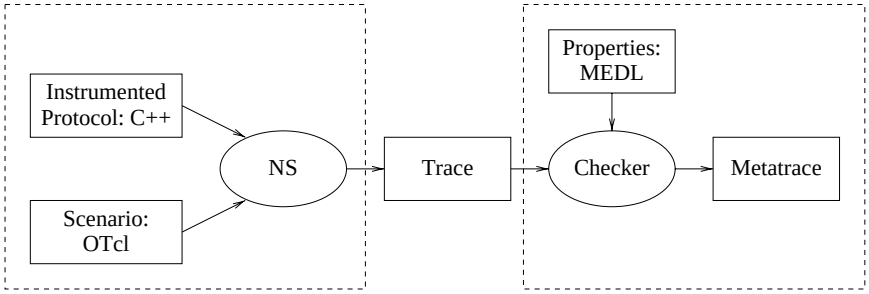


Figure 1: The Architecture of Verisim

The resulting toolset provides the instantiation of the MaC framework depicted in Figure 1. The resulting integrated system enables flexible formal analysis of network simulations where properties are expressed in MEDL, the input language of MaC based on temporal logic, and checked on traces produced by NS.

The use of MEDL allows the users writing a property to be checked to concentrate on *what* needs to be checked rather than *how* to check it. We note that, for simple properties, an *ad hoc* implementation of the checker - say, as a Perl script - may be easy to write and faster than a general-purpose checker. However, the checking algorithm will have to be implemented anew for every property one wants to check and, as properties become more complex, *ad hoc* checkers become more and more error-prone.

### 2.1 NS Network Simulations

Simulator implementations of protocols under development can provide an idea of how the protocols behave in a wide variety of network environments. Typically, a protocol and a suite of scenarios can be generated quickly and the simulation result is then provided as a feedback to the protocol designer. As such, simulator traces often reveal design flaws and potential improvements in the protocol before a laboratory testbed is even considered. Moreover, the simulator code often serves as a reference implementation for the protocol.

The development of a custom simulation framework for a single protocol allows the designer to investigate small topologies and basic characteristics of a new protocol. However, such simulations are limited in their ability to provide data about how a protocol interacts in the larger, multi-protocol environments where it must eventually operate. An extensible, multi-protocol simulation framework allows protocol designers to layer their protocol implementation at the node level and analyze its performance and interaction with other protocols. NS [13] is a discrete event network simulator developed by the VINT Project (`http://netweb.usc.edu/vint`), a collaboration between UC Berkeley, LBL, USC/ISI, and Xerox PARC, that provides such a framework. The system we study in this paper is based on NS, and our case studies use an extension of it by the CMU Monarch group (`http://monarch.cs.cmu.edu`) that adds link-layer and physical layer support for wireless networks.

A block diagram showing the steps in an NS simulation is shown in Figure 2. In order to carry out simulations using NS, one first implements the protocol in C++ using a collection of simulator constructs. A number of well-known protocols have been implemented for NS and can be used in simulations of newer protocols. For instance, the NS release provides TCP, UDP, IP,

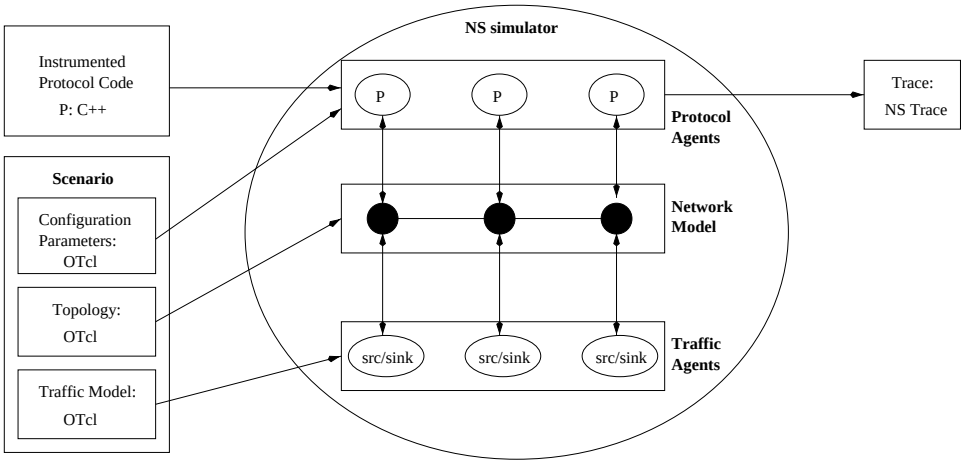


Figure 2: Simulations Using NSv2

and various routing protocols. These protocols are typically implemented as vertical layers on a node. New protocols may be implemented on top of or in between such pre-existing layers. Next, one needs to generate a simulation scenario written as a script. A typical NS scenario consists of a dynamic topology description, a traffic model, and various protocol configuration parameters. The simulator is then compiled with the protocol code and the scenario to produce a protocol-specific simulator. When the simulator is executed, a network model is constructed from the scenario topology, while data sources and sinks are added according to the traffic model. Protocol agents are attached to nodes in the network and their behavior is simulated. The result is a trace of all the packets produced, transported, dropped in the network, and any other diagnostic information directly instrumented into the protocol simulator code. This trace is typically used to analyze the performance of the protocol in terms of metrics like end-to-end delay, queue lengths, bandwidth, network throughput and goodput. It can also be fed into a visualization tool to help understand the network scenario and protocol response.

## 2.2 MaC Monitoring and Checking

*Monitoring and Checking (MaC)* is a framework for dynamic analysis of safety properties of systems with a trace-based semantics. The overall framework is depicted in Figure 3. The frame-

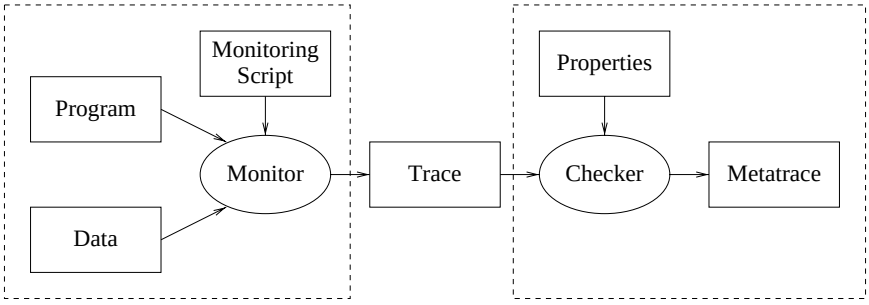


Figure 3: Overview of the MaC Framework

work includes two main phases: (1) before the system is run, its requirement and implementation

specification are used to generate run-time monitoring components; (2) during system execution, information about the running system is collected and matched against the requirements.

A major task in the first phase is to specify requirements formally. In addition, a user identifies aspects of the program execution that must be observed and reported in the trace, so that the desired properties may be checked. This is specified in what we call the *monitoring script*. The primary reason for having a separate monitoring script and requirement specification, is to separate implementation-specific details of monitoring from requirements specification. This separation helps in extending the framework for different implementation languages and specification formalisms, while providing a clean interface to the designer of monitors. In the first phase, run-time components of the MaC framework are generated from a requirement specification and monitoring script automatically; the monitoring script generates a monitor and the requirement specification generates a checker.

The run-time architecture of the MaC framework consists of a *monitor* and a *checker* [17, 18]. The monitor observes the running program and generates a sequence of events which is then examined by the checker. The *metatrace*, which is generated by the checker, contains the timed sequence of properties that have been violated during the execution, along with additional information about the system state, which can then be used for the purposes of debugging. Previously, we developed an implementation of the MaC framework for monitoring and checking of Java programs, called Java-MaC.

Verisim uses the checker component of Java-MaC, while the monitor component of Java-MaC is replaced by NS. The checker is based on the *Meta Event Definition Language (MEDL)*, which is designed to express properties of traces. MEDL is an extension of linear temporal logic (LTL) that captures safety properties (see [25] for a precise characterization of MEDL's expressive power). Safety properties [1] are requirements whose violation can be detected by examining a finite prefix of the execution. Any run-time checking of computation must make decisions about the validity or faultiness of a trace based on what it has seen so far; hence safety properties are the class of properties that can be checked dynamically. MEDL also has additional variables that may be used to record certain aspects of the trace. These variables represent the checker's state when trying to check if the trace conforms to the property. The presence of auxiliary variables in MEDL allows users to overcome certain well known limitations in the expressive power of LTL. For example, within MEDL one can 'count' and so it is possible to express things like '$\mathsf{RREP}$ should happen before the $5th$ occurrence of $\mathsf{RREQ}$'. As in SCR [14], we distinguish between two kinds of data that make up the trace of an execution: things that are true at some instant during the execution (which we call *events*), and facts that hold for a longer duration of time (which are called *conditions*). For example, the return from the method `SendRequest` occurs only at the instant when the control returns from the method, while a boolean condition like ($\mathsf{next\_hop}_d == 2$) holds for as long as $\mathsf{next\_hop}_d$ does not change its value from 2. The distinction between events and conditions is important in terms of what the checker can infer about the execution based on the information extracted by the monitor. The checker assumes that truth values of all conditions remain unchanged between updates from the monitor. For events, the checker makes the dual assumption, namely, that no events (of interest) happen between updates.

Based on this distinction between events and conditions, we have a simple two-sorted logic that constitutes MEDL. The syntax of conditions (C) and events (E) is given in Table 1. Here `e` refers to primitive events that are reported in the trace by the monitor; `c` is either a primitive condition reported in the trace or a boolean condition defined on the auxiliary variables. Guards (G) are

Table 1: MEDL Grammar

```
<C> ::=   c                <E> ::=   e                <G> ::= <E> -> <Statements>
        | [ <E>, <E> )              | start(<C>)
        | ! <C>                    | end(<C>)
        | <C> && <C>               | <E> && <E>
        | <C> || <C>               | <E> || <E>
        | <C> => <C>               | <E> when <C>
```

used to update auxiliary variables that may record something about the history of the execution.

The models for this logic are similar to those for linear temporal logic, in that they are sequences of worlds. The worlds correspond to instants in time at which we have information about the truth values of primitive conditions and events. Intuitively, these worlds correspond to the times when the monitor adds something to the trace. The intuition in describing the semantics of events and conditions based on such models is that conditions retain their truth values in the duration between two worlds, while events are present only at the instants corresponding to certain worlds. The labels on the worlds give the truth values of primitive conditions and events. The semantics for negation (`!c`), conjunction (`c1 && c2`), disjunction (`c1 || c2`) and implication (`c1 => c2`) of conditions is standard. Each condition is associated with two events, one that happens when the condition becomes true (`start(c)`) and the other that happens when the condition becomes false (`end(c)`). Conversely, occurrences of any two events `e1, e2` define an interval of time, and thus form a condition `[e1, e2)` that is true from an occurrence of event `e1` until the first occurrence of `e2`. The event `e when c` is true if `e` occurs and condition `c` is true at that time instant. Finally, a guard `e -> stmt`, is executed when event `e` is true; the effect of the execution is to update the values of the auxiliary variables according to the assignments given in `stmt`. In the assignments, we follow the common practice to denote by `x'` the "next state" value of variable `x`. The formal semantics for the logic is given in [17, 18].

Primitive events sent by the checker may have values associated with them that give detailed information about the event. For example, an event that represents an update of a monitored variable will have the new value of the variable attached to it. The values may be used in assignments to auxiliary variables and event definitions.

Appendix A gives a complete MEDL script for some of the properties used in the Verisim analysis that we describe in subsequent sections. Here we use it to illustrate the concrete syntax of MEDL. The script opens with a script name. The first section identifies primitive events and conditions that are sent to the checker by the monitor with the keyword `import`. For example, `import event eventty` yields one primitive event, which represents the type of the simulation event in the example. The next section gives typed declarations of auxiliary variables. The declaration `var int best_hc[at][dst]` introduces, for each pair of nodes (`at`, `dst`), a variable that represents the best known distance (hop count) between the nodes. Further in the script, events and conditions are defined in terms of other events and conditions and the values of primitive events and auxiliary variables. For example, we represent an event that happens when a packet containing routing information about node `dst` is sent by node `at` as:

```
event sendroute[at][dst] = routeinfo[at][dst] when ((value(eventty,0) == 1)
                                                    && (value(src_hc,0) < 255));
```

Here, `routeinfo` is a previously defined event that occurs when routing information is present in a packet, and the first value associated with the last occurrence of event `eventty` is 1, denoting a send event. Values of primitive events are accessed by their index, starting at 0. Finally, the script contains assignments to the auxiliary variables in response to events. For example, event `init`, which is the first event sent by the monitor, performs initialization of the checker state:
`init -> { best_hc[at][dst]' = 0; }`.

The checker, which is generated automatically from the MEDL script, evaluates the events and conditions described in the script, whenever it reads an element from the trace. There can be dependencies between different events and conditions. For example, an event `e1` that is defined in terms of an auxiliary variable that is updated by event `e2`, must be evaluated after `e2` and the variable have been updated. In order to evaluate events and conditions in a consistent order, we use a DAG data structure that implicitly encodes this dependency and has additional information that allows for fast evaluation of the events and conditions. A very important feature of our checking algorithm is that evaluates events, conditions, and guards on a need basis. This is particularly useful when checking simulations of network protocols. The requirements for a network protocol typically impose constraints on every pair of nodes in the network. However, a simulation event, such as a packet arrival, affects only a few of the nodes and thus only a small subset of the constraints may be violated. The evaluation algorithm re-evaluates only those events and condition that are affected by the incoming data.

The remainder of this paper deals with the validation of Verisim as a test harness for network simulations. To carry out this validation, we perform a case study based on a new protocol currently being standardized by IETF in the Manet Working Group. We present the protocol in the next section, along with some of the properties it is expected to satisfy. For this study, we selected simulation code written by the Monarch group at CMU, one of the research groups working on Manet protocols. As with any complex software, the version of the Monarch code we study has some bugs. We show how to find several of these using Verisim in a simulation of modest complexity.[1] Our first analysis focuses on the use of Verisim as a debugging aid, demonstrating the kinds of bugs that can be found. Our second study focuses on the strategy for using Verisim for debugging, focusing on efficient means for analyzing metatraces to find collections of independent bugs. The aim of the first study is to determine whether Verisim is useful while the aim of the second is to determine whether refinements in methodology can make it more useful.

# 3 AODV Routing

This section describes the AODV routing protocol [21, 22] which we used in our case study. The first part provides a short description of the protocol. The second part discusses some of its requirements—properties that are expected to hold in AODV implementations.

## 3.1 AODV Protocol

The Ad Hoc On-Demand Distance Vector (AODV) routing protocol is used in packet radio networks. A packet radio network consists of a collection of mobile nodes whose link connectivity

[1] We reported these bugs when we found them so they could be removed from subsequent versions of the Monarch simulator code.

frequently changes due to the node movement. Because of dynamic connectivity and a typically low link bandwidth, AODV establishes routes 'on-demand' (that is, only when they are needed).

A route to a destination $d$ contains the following fields:

$\mathsf{next\_hop}_d$: Next node on a path to $d$.

$\mathsf{hop\_cnt}_d$: Distance from $d$, measured in the number of nodes (hops) that need to be traversed to reach $d$.

$\mathsf{seq\_no}_d$: 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:

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

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 dest_seq_no specifies the least sequence number for a route to $d$ that $s$ is willing to accept ($s$ usually uses its own $\mathsf{seq\_no}_d$ for this purpose). 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 $d$ stamped with a sequence number at least as big as dest_seq_no. 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:

$$\mathsf{RREP}(\mathsf{hops\_to\_dest}, d, \mathsf{dest\_seq\_no}, \mathsf{route\_lifetime}).$$

Arguments hops_to_dest, dest_seq_no, and route_lifetime 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 dest_seq_no 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_to_dest 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 the bigger 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 above 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 $\mathsf{next\_hop}_d$ 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 $\mathsf{hops\_to\_dest} = 255$ (infinity), and its $\mathsf{dest\_seq\_no}$ is one more than the previous sequence number for that route. 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.

### 3.2 AODV Properties

Routing protocols are often compared based on performance statistics like speed of convergence, amount of bandwidth and memory needed for control data, and so on. However, the *quality* of the results produced by different protocols may vary. For instance, it is unfair to compare a slow routing protocol that always finds shortest routes with a really fast protocol that sometimes creates routing loops. This is why it is important to know what kind of correctness attributes a given protocol provides when comparing its performance to other protocols.

Our study focuses on analyzing correctness of AODV implementations. This can be studied from two angles: correctness with respect to the *requirements* and correctness with respect to the *standard*.

Requirements are high-level properties that a protocol is supposed to satisfy. They are usually not protocol specific, in the sense that a same requirement property usually makes sense in the context of many different protocols. A common requirement for a routing protocol is *Loop Freedom:* Computed routes never contain loops. It turns out that in the case of AODV it suffices to prove a simple invariant in order to guarantee loop freedom [5]. The loop freedom invariant is described in Table 2. Other typical routing protocol requirements are optimality of the computed routes, convergence to valid routes, etc.

Table 2: AODV Requirement: Loop Freedom

| |
|---|
| **Loop Invariant:** Along every AODV route to a destination $d$, pair $(-\mathsf{seq\_no}_d, \mathsf{hop\_cnt}_d)$ strictly decreases in the lexicographic ordering. |

A protocol standard is a document which gives basic guidelines on how to implement a protocol. Its purpose is two-fold—it ensures interoperability between different implementations and it (supposedly) ensures satisfaction of the requirements. The standard helps the implementors by describing a particular way in which the requirements are supposed to be satisfied. Since each protocol has its own standard specification, properties that describe the standard will be much more protocol-specific than the properties that describe the requirements.

The core part of a protocol standard describes what kinds of events can occur and how are nodes supposed to handle them. Network protocols usually represent reactive systems, which means that every action is carried out in response to an event. Although the standard is written in natural language, one can typically extract the state machine that it is trying to express. For example, the

state machine corresponding to an AODV process is shown in Appendix B. To *monitor* conformance with such a state machine, we convert it to a monitoring specification that gets triggered every time a network event of interest happens. The monitoring specification attempts to keep track of the state of the protocol, and checks that the events generated by the protocol are correct with respect to the state machine.

MEDL, with its explicit notion of events, and its notion of explicit state transformations, proves to be a very practical language for expressing properties of network protocols. Events include, but are not limited to, packet receipts and timeouts. When an event occurs, the state of the protocol is updated and possibly new packets and timers are generated. Table 3 shows some of the properties that test adherence to the AODV standard. These properties were generated from the state machine description in Appendix B. Notice how each property contains an event in its description (denoted by a phrase of the form *when...* or *if...*). We should point out that the set of standard properties listed in Table 3 is not complete—satisfying all of the properties still does not guarantee adherence to the standard. In particular, there are a number of properties about the timing of protocol events that our state machine, and consequently our monitoring specification, does not express. It is generally not feasible to capture the whole standard as a single set of checkable properties. Size and complexity of the standard impose practical limitations on this task. Another limitation is the expressive power of the logic in which the properties are stated, as well as the complexity of checking procedures. For instance, we can only express and check safety properties, but it is conceivable that a standard may include liveness properties as well. Finally, standards sometimes prescribe implementation details whose satisfaction can not be checked by observing protocol runs. Completeness is not essential, since our goal is finding errors, not showing correctness. The richer the set of properties, the more kinds of errors can be detected.

Table 3: AODV Standard Properties

| **Property Name** | **Property Description** |
|---|---|
| Monotone Sequence Numbers | A node's own sequence number never decreases. |
| Destination Stops | When a packet (RREQ, RREP or data) reaches its destination, it should not be forwarded. |
| Correct Forwarding | If a packet addressed to $d$ (RREP or data) is forwarded, it is forwarded along the best unexpired route to $d$ seen so far. |
| Destination Reply | When the destination replies to a route request, the value of the hops_to_dest field of the reply should be 0. |
| Node Reply | When a node sends a route, it sends the best unexpired route seen so far. |
| RREQ Sequence Number | When a node initiates a route request for a destination $d$, the requested sequence number should either be 0, or the last sequence number recorded for $d$ ($\mathsf{seq\_no}_d$). |
| Detect Route Error | If a node detects a broken route, it should use $\mathsf{dest\_seq\_no} = 1 +$ (its own) $\mathsf{seq\_no}_d$ in the unsolicited RREP. |
| Forward Route Error | When a node forwards an unsolicited RREP, it should forward the same sequence number that it received. |

# 4 Checking AODV Simulations

In this section, we analyze AODV simulations using Verisim. Verisim generates a large meta-trace of property violations. We use bug-repairing and tuning to discover errors in the protocol implementation.

## 4.1 AODV properties in MEDL

Our first task is to translate properties given in section 3.2 in MEDL. Generally, all properties are constructed to capture deviations of the *observed* behavior from the ideal (*correct*) behavior. In our framework, observable behavior of a routing protocol is the sequence of packets exchanged between the nodes. Based on the packet sequence, our MEDL property constructs the ideal system state and compares it to the observed system state. For instance, if a RREP packet heading towards a node $u$ is forwarded from node $v$ to node $w$, the observed routing table at $v$ has $\mathsf{next\_hop}_u = w$. However, by monitoring the history of RREP messages received at $v$, we can see whether $v$ was indeed expected to have such a route to $u$.

To give an example, recall the Loop Invariant property from the previous section. Consider some three different nodes: $at$, $nxt$ and $dst$. Assume that the node $at$ has a route to $dst$ through its neighbor $nxt$:

$$\mathsf{next\_hop}_{dst}(at) = nxt.$$

Let $(s(at), h(at))$ be the sequence number and the hop count that node $at$ has for the destination $dst$ (similarly $(s(nxt), h(nxt))$ for the node $nxt$). The Loop Invariant property says:

$$(s(at) \leq s(nxt)) \ \wedge \ (s(at) = s(nxt) \ \Rightarrow \ h(at) > h(nxt)).$$

Therefore, the property is violated exactly when the following holds:

$$(s(at) > s(nxt)) \ \vee \ (s(at) = s(nxt) \ \wedge \ h(at) \leq h(nxt)).$$

Table 4 shows a MEDL alarm that detects this violation in the state of the nodes, as reconstructed from observed events.

Table 4: Loop Invariant in MEDL

```
alarm LoopInv[at][nxt][dst] =
        start((at!=nxt) && (at!=dst) && (nxt!=dst) &&
              (best_next[at][dst] == nxt) &&
              ((best_seq[at][dst] > best_seq[nxt][dst]) ||
               ((best_seq[at][dst] == best_seq[nxt][dst]) &&
                (best_hops[at][dst] <= best_hops[nxt][dst]))))
```

The auxiliary variables: `best_seq`, `best_next`, and `best_hops` keep track of the sequence number, next hop and hop count of a node's current route to the destination $dst$. We compare the states of nodes $at$ and $nxt$ to check if the loop invariant is being violated.

This will be our general strategy for translation—we first encode the ideal state machine in terms of auxiliary variables; then we convert the desired state invariants, and properties of the

outputs into alarms by negation. Table 5 shows AODV properties and their corresponding MEDL alarm names. Appendix A gives the complete MEDL scripts for many of these properties.

Table 5: MEDL Alarms

| **Property** | **MEDL alarm** |
|---|---|
| Monotone Sequence Numbers | MonSeqNo |
| Destination Stops | DestStops |
| Correct Forwarding | CorrectFwd |
| Destination Reply | DestRep |
| Node Reply | NodeRep |
| RREQ Sequence Number | ReqSeqNo |
| Loop Invariant | LoopInv |
| Detect Route Error | DetectRErr |
| Forward Route Error | FwdRErr |

### 4.2 AODV Simulation Case Study

We consider an implementation of AODV written by the CMU Monarch Project(`http://monarch.cs.cmu.edu`) for the network simulator NS. This code was used primarily for performance analysis of AODV in comparison with other routing protocols for mobile, ad hoc networks [9]. In order to carry out this comparison, a number of large random scenarios were constructed as well.

The Monarch implementation is based on the first version of AODV [21], and is known to have bugs—because of incomplete specification in the standard, and due to programmer errors. The code is already instrumented to produce a packet trace for every packet generated, forwarded and dropped by the protocol. We use Verisim to analyze NS simulations of this code on a small network scenario $S$ with 5 nodes, as shown in Figure 4.

Topology: There are 5 nodes initially arranged as in Figure 4 (Phase I). Then node 5 starts moving away from the network, causing the wireless links to break after 2.5s (Phase II). 30s into the simulation, node 5 heads back towards node 1. At 55s it is within the range of node 4 (Phase III), at 70s it is in the range of nodes 2,3, and 4 and finally it is in the range of 1,2, and 3 (Phase IV).

Traffic Model: Nodes 1,2 and 3 are constant bit rate (CBR) sources for node 5. They send a total of 1000 packets of size 512 bytes each, one packet every 0.1s.

AODV parameters: We use the optimal AODV configuration computed by the Monarch group. The configuration involves parameters like route timeout intervals and the number of times a request should be re-tried.

When the AODV protocol is simulated on scenario $S$, NS generates a trace $T$. The initial fragment of a typical trace is shown in Table 6. When a packet send or receive event happens at a node N, there is a line in the trace with the format:

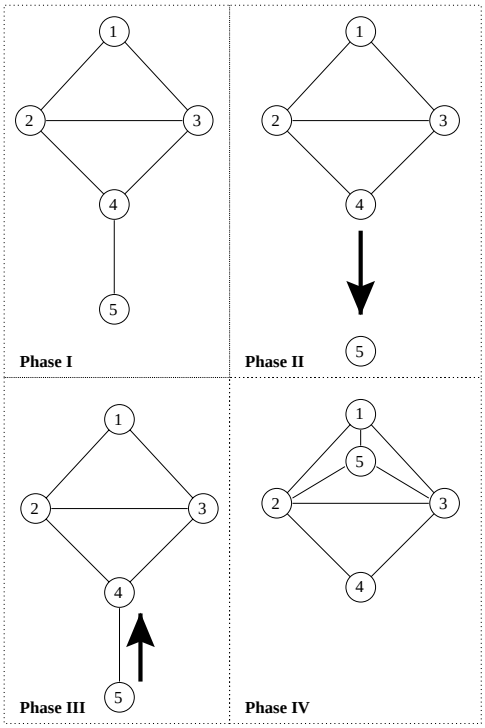


Figure 4: Scenario $S$

```
<send/recv> <time> _N_ RTR --- <Link Layer info> -------  <IP info> <AODV info>
```

For instance, the third line of the trace tells us that at time `0.0`, node `3` broadcast an `AODV REQUEST`, for destination `5`, with hop count `0` and broadcast id `1`. Moreover, node `3`'s current sequence number is `1`, and the last sequence number it heard from the destination (`5`) is `0`. This request eventually reaches the destination `5`, through node `4`. The last line of the trace is node `5`'s `REPLY` to the request which it unicasts to node `3`, via node `4`.

## 4.3 Repair First Bug

We start with Monarch code for AODV ($P$), and simulate it using NS for the scenario $S$ to produce the trace $T$ (Table 6). Verisim then checks whether $T$ satisfies the AODV properties $\phi$, and produces a meta-trace $T^{\phi}$ of property violations (alarms). This meta-trace generation is then repeated, on succeeding versions of $P$. Statistics on the alarms found in these meta-traces are shown in Table 7. We show the results for a representative set of AODV properties. The last column in the table contains the total number of violations of *all* the properties (including the ones not shown). The MEDL specification for these selected AODV properties is given in Appendix A.

### Step I

The first meta-trace $T^{\phi}$ contains 220 alarms, and the initial fragment is as shown in Table 8. This alarm trace has 4 DestRep alarms, 43 instances of LoopInv, 54 DetectRErr alarms, and 38 instances of NodeRep. Incidentally, the first alarm in $T^{\phi}$ is raised at the last event of $T$ shown in Table 6.

Table 6: Typical Trace $T$

```
s 0.000000000 _1_ RTR  --- 0 AODV 52 [0 0 0 0 0] -------  [1:255 -1:255 32 0] [0x2 0 1 [5 0] [1 1]] (REQUEST)
s 0.000000000 _2_ RTR  --- 0 AODV 52 [0 0 0 0 0] -------  [2:255 -1:255 32 0] [0x2 0 1 [5 0] [2 1]] (REQUEST)
s 0.000000000 _3_ RTR  --- 0 AODV 52 [0 0 0 0 0] -------  [3:255 -1:255 32 0] [0x2 0 1 [5 0] [3 1]] (REQUEST)
r 0.000519784 _2_ RTR  --- 0 AODV 52 [20 0 ffff 1 800] ---  [1:255 -1:255 32 0] [0x2 0 1 [5 0] [1 1]] (REQUEST)
r 0.000535386 _3_ RTR  --- 0 AODV 52 [20 0 ffff 1 800] ---  [1:255 -1:255 32 0] [0x2 0 1 [5 0] [1 1]] (REQUEST)
r 0.002002991 _1_ RTR  --- 0 AODV 52 [20 0 ffff 3 800] ---  [3:255 -1:255 32 0] [0x2 0 1 [5 0] [3 1]] (REQUEST)
r 0.002006118 _2_ RTR  --- 0 AODV 52 [20 0 ffff 3 800] ---  [3:255 -1:255 32 0] [0x2 0 1 [5 0] [3 1]] (REQUEST)
r 0.002014489 _4_ RTR  --- 0 AODV 52 [20 0 ffff 3 800] ---  [3:255 -1:255 32 0] [0x2 0 1 [5 0] [3 1]] (REQUEST)
s 0.002360210 _4_ RTR  --- 0 AODV 52 [20 0 ffff 3 800] ---  [4:255 -1:255 31 0] [0x2 1 1 [5 0] [3 1]] (REQUEST)
r 0.002689325 _1_ RTR  --- 0 AODV 52 [20 0 ffff 2 800] ---  [2:255 -1:255 32 0] [0x2 0 1 [5 0] [2 1]] (REQUEST)
r 0.002700822 _4_ RTR  --- 0 AODV 52 [20 0 ffff 2 800] ---  [2:255 -1:255 32 0] [0x2 0 1 [5 0] [2 1]] (REQUEST)
r 0.002708053 _3_ RTR  --- 0 AODV 52 [20 0 ffff 2 800] ---  [2:255 -1:255 32 0] [0x2 0 1 [5 0] [2 1]] (REQUEST)
s 0.002777804 _4_ RTR  --- 0 AODV 52 [20 0 ffff 2 800] ---  [4:255 -1:255 31 0] [0x2 1 1 [5 0] [2 1]] (REQUEST)
r 0.003439172 _2_ RTR  --- 0 AODV 52 [20 0 ffff 4 800] ---  [4:255 -1:255 31 0] [0x2 1 1 [5 0] [3 1]] (REQUEST)
r 0.003449342 _5_ RTR  --- 0 AODV 52 [20 0 ffff 4 800] ---  [4:255 -1:255 31 0] [0x2 1 1 [5 0] [3 1]] (REQUEST)
s 0.003449342 _5_ RTR  --- 0 AODV 44 [0 0 0 0 0] -------  [5:255 3:255 32 4] [0x4 1 [5 2] 600] (REPLY)
```

Table 7: RFB Alarms

| **Meta-trace** | **DestRep** | **DetectRErr** | **NodeRep** | **LoopInv** | **Total alarms** |
|---|---|---|---|---|---|
| $T^{\phi}$ | 4 | 54 | 38 | 43 | 220 |
| $T_1^{\phi}$ | 0 | 54 | 38 | 43 | 216 |
| $T_2^{\phi}$ | 0 | 48 | 39 | 44 | 206 |
| $T_3^{\phi}$ | 0 | 0 | 0 | 0 | 1 |

Table 8: Typical Meta-trace $T^{\phi}$

```
------------------------------------------------------------
Time: 0.003449342s, Alarm DestRep raised at 5 for dest 5
best route at 5 for 5: <seqno: -1,hc: -1,next: -1>
observed route at 5 for 5: <seqno: 2,hc: 1>
------------------------------------------------------------
Time: 0.004823314s, Alarm DestRep raised at 5 for dest 5
best route at 5 for 5: <seqno: 2,hc: -1,next: -1>
observed route at 5 for 5: <seqno: 3,hc: 1>
------------------------------------------------------------
Time: 2.567054284s, Alarm DetectRErr raised at 4 for dest 5
best route at 4 for 5: <seqno: 3,hc: 1,next: 5>
observed route at 4 for 5: <seqno: 3,hc: 255>
------------------------------------------------------------
Time: 2.567054284s, Alarm DetectRErr raised at 4 for dest 5
best route at 4 for 5: <seqno: 3,hc: 1,next: 5>
observed route at 4 for 5: <seqno: 3,hc: 255>
------------------------------------------------------------
```

The first alarm is a DestRep at destination 5, which means that the implementation is not setting the initial hop-count value in an RREP correctly. All four instances of the alarm in $T^\phi$ indicate that the initial value has been set to 1. So we go into the code and correct this simple off-by-one error, changing the initial hop-count from 1 to 0. This produces a new implementation $P_1$, which we use to produce a new trace $T_1$, by running the simulation again.

**Step II**

We run Verisim on $T_1$ and $\phi$ to produce the second meta-trace $T_1^\phi$. $T_1^\phi$ has 216 alarms, and is the same as $T^\phi$ except that the DestRep alarms have been eliminated. The first alarm in the trace is a DetectRErr at node 4, where the node 4 is sending an unsolicited RREP, saying that the destination 5 is unreachable. However, the sequence number in the RREP is not 1 more than the best sequence number at 4. This leads us to suspect that the implementation fails to increment the sequence number at 4 before sending the unsolicited RREP. Looking at other DetectRErr alarms in the trace confirms this bug. We repair $P_1$, to eliminate this bug and produce the third version of our code, $P_2$.

**Step III**

As before we analyze $P_2$ through Verisim to produce $T_2$ and $T_2^\phi$. $T_2^\phi$ has 206 alarms, of which 44 alarms are due LoopInv, 48 are DetectRErr alarms, and 39 are NodeRep alarms. Some of the DetectRErr alarms we detected before are gone, but a number of alarms remain. Interestingly, the NodeRep alarms and the LoopInv alarms increase by 1. This is because in the old trace, when the incorrect route errors are received by nodes, the MEDL formula assumes they are ignored. However, in the new trace, the generated route errors have the correct hop-count, so $\phi$ recognizes that they will be acknowledged by the recipients. This leads to more errors being recognized.

The first alarm is a NodeRep at node 3, which advertises a route with hop-count 2 for the destination 5 even though it no longer has a route to the destination. It is in effect advertising outdated routes. We conclude that the conditions that check whether an RREP should be sent are buggy and that routes are not deleted properly in the code. Indeed we find, when we look at the code, that the RREP generation code has multiple errors in it. We need to change 3 conditional expressions in the code, to make it conform to our properties. Finally, we again run Verisim on this new implementation $P_3$ to produce a trace $T_3$ and meta-trace $T_3^\phi$.

**Step IV**

The fourth meta-trace just contains one alarm, which is raised because of an unexpected buffering event at a lower layer protocol in the simulation. Essentially, a packet $p_n$ received at node 3 is buffered at a lower layer while the protocol responds to an older packet $p_o$. However, our MEDL formula, which does not model lower layer protocols, assumes that $p_n$ has already been seen and processed by the protocol, causing the alarm. As such, $T_3$ is 'correct' with respect to the AODV properties that we modeled in MEDL.

### 4.4 Tuning

The previous section demonstrated the repair first bug technique for bug-hunting, involving new simulations every time a bug was discovered. In this section, we demonstrate tuning for MEDL, which allows us to discover multiple bugs in every simulation run. We first simulate $P$ with $S$ to get $T$, which is analyzed with the MEDL formula $\phi$ to get the meta-trace $T^{\phi}$. As before, we start our analysis by looking at $T^{\phi}$. However, when we find a bug, we tune our MEDL formula $\phi$ instead of repairing the protocol code $P$. After this tuning, we re-run the checking part of Verisim on $T$ along with the new MEDL formula to generate the next meta-trace. The alarm statistics for tuning are as shown in Table 9. The MEDL script for the properties in the table is included, along with the tuning modifications, in Appendix A.

Table 9: Tuning Alarms

| **Meta-trace** | **DestRep** | **DetectRErr** | **NodeRep** | **LoopInv** | **Total alarms** |
|---|---|---|---|---|---|
| $T^{\phi}$ | 4 | 54 | 38 | 43 | 220 |
| $T^{\phi_1}$ | 0 | 54 | 38 | 43 | 216 |
| $T^{\phi_2}$ | 0 | 0 | 38 | 50 | 166 |
| $T^{\phi_3}$ | 0 | 0 | 21 | 0 | 30 |

#### Step I

As before the first alarm in $T^{\phi}$ is a DestRep at destination 5, which initializes the hop-count in the RREP to 1. This probably means that the code is initializing a node's self-hop-count to 1 instead of 0. So we modify the alarm DestRep to check whether a node ever emits a hop-count other than 1 (instead of 0). Then we run Verisim on $T$ and this new MEDL formula $\phi_1$ to get the meta-trace $T^{\phi_1}$. All the DestRep alarms disappear in the new meta-trace which validates our assumption and identifies the first bug in the code.

#### Step II

The second meta-trace $T^{\phi_1}$ has 216 alarms and is the same as $T^{\phi}$ except that the DestRep alarms no longer appear. After looking at the meta-trace, we guess that the first alarm, DetectRErr, is because a node that discovers a route error fails to increment the destination sequence number. As before, we can modify the alarm DetectRErr to ignore this case. However, according to the meta-trace, route error information still seems to propagate through the network. This means that the implementation of the route error packet handler must be incorrect, but in a way that allows the route error to be propagated. So, in order to ignore alarms related to the route error messages, we modify the route error packet handling routine in the MEDL formula as well. Note that by making this modification, we are making the MEDL formula 'incorrect'—we are changing the ideal state so that it becomes the same as the observed state. This change generates the third version, $\phi_2$, which is used to produce the meta-trace $T^{\phi_2}$. Indeed, $T^{\phi_2}$ seems to not have the kinds of DetectRErr alarms and follow-up alarms as noticed before.

**Step III**

$T^{\phi_2}$ has 166 alarms, of which 50 are LoopInv alarms and 38 are NodeRep alarms. Both DestRep and DetectRErr have been eliminated. Observe that the LoopInv alarms have increased because the modified MEDL state allows more alarms to be identified. As before, we look at the meta-trace and conclude that the way replies are generated in the protocol code is incorrect. In particular, even when a node has lost a route, it keeps its hop-count around and when an RREQ is received, it incorrectly replies as if it has a route. We imitate this behavior by changing the MEDL formula to assume the same by allowing hop counts to stay even after the route has been lost. We run Verisim on this formula $\phi_3$ and generate the fourth meta-trace $T^{\phi_3}$.

**Step IV**

The new meta-trace $T^{\phi_3}$ still has 30 alarms, with 21 NodeRep alarms that are difficult to interpret. Essentially, at this point, too much information has been filtered out of the trace to make any good guesses about the origin of the errors. So we go back to the code to repair the three bugs detected above. When we look at the code for the RREP generation, we realize that the implementation has multiple bugs causing it to behave highly unexpectedly. These bugs explain the alarms remaining in $T^{\phi_3}$. We repair $P$ to produce a new implementation $P_f$., which is analyzed through Verisim to produce $T_f^{\phi}$. $T_f^{\phi}$ has a total of 1 alarm due to a packet buffering event at a node.

### 4.5 Analysis

We discovered 3 errors in the AODV implementation, which altogether required rewriting 18 lines of the Monarch code. Of these, the RREP generation problem is particularly interesting. This error causes the AODV implementation to actually form loops, which we detected in our simulation. In general, loop formation is difficult to detect by other analyses. Indeed, our previous manual analyses of AODV simulations failed to detect the existence of loop or the RREP generation bugs that cause it. The automation provided by Verisim was crucial to detect and wade through property violations in the simulation.

It must be emphasized that the intuition that allows one to tune MEDL formulas is highly protocol specific. One must have a good understanding of the protocol, and conduct a manual analysis of the meta-trace before the faults that caused the errors can be guessed. It is often the case that there are several useful ways to tune a formula. We have demonstrated that Verisim is flexible enough to allow our guesses to be validated without even re-running the simulation, let alone looking at or modifying the code.

## 5 Abstractions and 'Off-The-Shelf' Simulations

In order to see how well our techniques scale up to simulations usually analyzed to measure the performance of a network protocol, we applied our techniques to the largest trace made available by the CMU Monarch group [9]. This 'Off-The-Shelf' (OTS) trace was generated by AODV simulation on a site of size $1500 \times 300$ meters with 50 nodes constantly moving at 20 meters per second. There were 150 data connections transmitting four 64 byte packets every second. The simulation and our Verisim analyses of the trace were carried out on a dual Pentium-III 550Mhz Xeon processors

Table 10: Results of MonSeqNo Property on Trace

| **Exp** | **Trace [# of events]** | **Property [size in bytes]** | **Time (in secs)** | **Rate (time/events/prop)** |
|---|---|---|---|---|
| A | $T$ [6,446,316] | $\mu$ [1,476,638] | $>4$ days | N/A |
| B | $T$ [6,446,316] | $F_\pi(\mu)$ [14,543] | 51,045 | $0.54\mu s$ |
| C | $E_\tau(T)$ [706,753] | $\mu$ [1,476,638] | $>4$ days | N/A |
| D | $E_\tau(T)$ [706,753] | $F_\pi(\mu)$ [14,543] | 5,440 | $0.53\mu s$ |
| E | $P_{\pi'}(T)$ [631,253] | $F_{\pi'}(\mu)$ [145,178] | 85,012 | $0.93\mu s$ |
| F | $P_\pi(T)$ [69,411] | $F_\pi(\mu)$ [14,543] | 556 | $0.55\mu s$ |
| G | $E_\tau(P_\pi(T))$ [6,812] | $F_\pi(\mu)$ [14,543] | 51 | $0.55\mu s$ |

Table 11: Results of LoopInv Property on Trace

| **Exp** | **Trace [# of events]** | **Property [size in bytes]** | **Time (in secs)** | **Rate (time/events/prop)** |
|---|---|---|---|---|
| H | $P_\pi(T)$ [69,411] | $F_\pi(\lambda)$ [75,508] | 8064 | $1.54\mu s$ |
| I | $E_{\tau'}(P_\pi(T))$ [48,735] | $F_\pi(\lambda)$ [75,508] | 5912 | $1.61\mu s$ |

machine with one gigabyte of memory. The OS was Red Hat Linux 6.1 with the 2.2.12-20 SMP Kernel. We used NS version 2.1b1 and MACSware 0.99 implemented in IBM JDK 1.1.8 for Linux and running on the JVM. The NS simulation itself required about 5220 seconds to complete and generated 6,446,316 events. This is much larger than the traces analyzed by Verisim in the previous section, which all had less than 10,000 events. A naive effort to use Verisim to analyze MonSeqNo, a relatively simple property, on this trace was prohibitively time-consuming. We estimate that the time required to check the desired relationship after each of 6,446,316 events between each pair of nodes (2500 relations) to be more than 100 days based on extrapolating a four-day run of the analysis. On the bright side, errors with MonSeqNo were detected in the first 4 days of analysis. More significantly, there are a number of optimizations that will find an error with considerably less effort. The results of analyzing the OTS simulation with various optimizations for the MonSeqNo (called $\mu$) property are given in Table 10. Two additional optimizations were tested on the LoopInv (called $\lambda$) property, and these results are provided in Table 11. The OTS trace is called $T$ in the tables. The naive analysis is Experiment A, recorded in the first line of Table 10.

The experiments measure the effects of various abstractions that one may perform on either the trace or the property to make the analysis feasible, while also finding errors in the code. There were two abstractions that we chose to apply: *population abstraction* and *packet-type abstraction*. Population abstraction focuses only on a small set of nodes, while ignoring the others. We can apply this abstraction to either the property being tested or to the trace. For example, when applied to the property MonSeqNo, it would mean that we check that only certain nodes satisfy the MonSeqNo property. When we apply this to the trace, we prune the trace to consist of only events sent or received by these nodes. In our case study, we looked at two population abstractions. In one we focused on packets at nodes 6 through 10 for the destinations 6 through 10 (25 relations). We call this $\pi$. In the other population abstraction, called $\pi'$, we also considered only packets at nodes 6 through 10, but we let the destination be any of the 50 nodes (250 relations). The result of

applying the population abstraction $\pi$ to a formula $\varphi$ is denoted by $F_\pi(\varphi)$. When the population abstraction is applied to a trace $T$, we denote it by $P_\pi(T)$. Population abstraction is applied to either the property or the trace in Experiments B, D, E, F, G, H, I of Tables 10 and 11.

In packet-type abstraction, we prune the trace to include only events that directly affect the property we are interested in. For example, for the MonSeqNo property, this abstraction (denoted by $E_\tau$) when applied to the trace, removes all events except for the `sendroute[at][dst]` event. The corresponding abstraction for the LoopInv property (denoted by $E_{\tau'}$), removes a different set of events from the trace. In experiments C,D,G, and I a packet type abstraction was applied.

It is important to make the distinction between population abstractions and packet-type abstractions. These two classes of abstractions have essential differences that one needs to be aware of when interpreting results of the abstracted simulations. Packet-type abstractions, if applied properly, are *complete*. This means that all errors from the original trace will still exist in the abstracted trace. Because of this, it is generally always useful to perform packet-type abstractions. They will more than likely improve the performance, while producing the same result as checking the original trace. In contrast, population abstractions can miss errors. This can happen if an error occurs outside the observed population. However, this is not very likely to happen with network protocols where all nodes run identical processes. Generally, both population abstractions and packet-type abstractions are *sound*—every error present in the abstracted simulation indicates an error in the original simulation. Formally, if we use the notation $T \models \varphi$ to indicate that a trace $T$ satisfies a formula $\varphi$, the following will hold for every event-type abstraction $\tau$ and a population abstracton $\pi$:

$$\begin{array}{lcl} T \models \varphi & \iff & E_\tau(T) \models \varphi \\ T \models \varphi & \implies & P_\pi(T) \models \varphi \\ T \models \varphi & \implies & T \models F_\pi(\varphi) \end{array}$$

Our case study revealed two things: linear growth in complexity and significant benefits from abstractions. First, the time taken to process the trace depends only linearly on the length of the trace and the size of the formula; this can be seen from the fact that the last column of our tables is nearly constant. The reason why the rates in Table 11 are three times more than those in Table 10 is because the property of LoopInv is more complicated and has a 3 alternations between `&&` and `||`. Second, abstractions can significantly improve the time taken in performing the analysis. For example, after applying both population and packet type abstractions, the time for the analysis went from more than 4 days (Experiment A) to 51 seconds (Experiment G). Moreover, this optimization did not excessively compromise our ability to discover bugs in the trace: the alarms associated with nodes 6 to 10 that would have been generated had we analyzed the entire trace are still generated when we test the much smaller trace we get after applying the abstractions.

# 6 Related Work

While there has been a great deal of research on the formal verification of communication systems, these efforts have generally been limited in two respects. First, they generally prove properties of the protocol and therefore may not be helpful in finding problems in protocol implementations. Second, few efforts have focused on multi-party protocols like routing, where proving a property of a fixed number of routers limits the scope of the proof drastically. [15] describes a method for studying behavior of multi-party protocols (such as PIM-SM) in 'stressful' conditions. (See [5] for a general discussion of verifying routing protocols.) These two problems are partially addressed by

the Verisim strategy of analyzing trace runs from simulations. First, the simulation code is closer to the implementation code and therefore the Verisim tests are more likely to reveal problems with the deployed system. Second, the ease of creating simulations makes it possible to test a large variety of configurations, thus partially addressing the problem that all configurations cannot be tested. In any event, Verisim analysis is complementary to both static and dynamic analysis, so it can be useful as long as it is convenient. Integration with NS contributes to this objective since simulations created for other reasons like performance analysis can easily be subjected to Verisim analysis as well.

A large body of related research work concentrates on automated generation of *test oracles* from the requirements. A general methodology for doing this is discussed in [24], together with examples in Real Time Interval Logic (RTIL) and Z. Papers [7, 6, 8] describe a trace analysis tool for LOTOS requirements, while [12] describes a similar tool for Estelle requirements. Generating test oracles for Graphical Interval Logic (GIL) is discussed in [11, 20]. An equivalent problem for a safe fragment of Linear Temporal Logic is discussed in [16]. This fragment is expressively similar to the requirements language of Verisim. However, an important feature that distinguishes Verisim from most of the above work is its focus on integration of simulation and testing. Another toolset that follows this idea is the simulation and monitoring platform MTSim [10], based on the graphical real-time specification language Modechart. An advantage of Verisim is that instead of using formal models, it uses off-the-shelf network simulators already designed for prototyping, performance evaluation and other purposes.

There is similarity between Verisim formal analysis of protocol simulations and network Intrusion Detection Systems (IDS's). IDS's aim to detect anomalies in network traffic to enable operators to discover problems or trigger automated responses. Examples include Next-generation Intrusion Detection Expert System (NIDES) [2], which performs both statistical analysis and rule-based signature analysis on audit records and Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD) [23], which detects malicious activity through and across large networks. Although IDS's often focus on detecting statistical anomalies like unusual volumes of certain kinds of traffic, at least some are able to check properties of the kind we describe in MEDL. Although we are not aware of any efforts to do so, such systems could perhaps be used in the way we have used Verisim to produce metatraces as a debugging aid for analyzing simulations. For instance, the rule-based analysis language (P-BEST language) [19] used in [2, 23] is as expressive as MEDL.

Additional information about related work can be obtained from [4], which describes a taxonomy for logical analysis of networks and uses this to classify some of the literature. A survey of tools used in the Verinet project (including Verisim) can be found in [3].

# 7 Conclusion

We have demonstrated an integrated system called Verisim consisting of a network simulator and a logic-based checker for traces of events. This combination provides a flexible approach to studying correctness properties of network simulations. We have shown the usefulness of the tool by demonstrating how it can find flaws in non-trivial simulator code. We have also shown how its flexibility can be exploited through the concept of tuning to improve the turn-around time in debugging. We believe that the approach is practical and scalable and can be used as a productive adjunct to standard network protocol engineering practices.

## Acknowledgments

We would like to express thanks to Mike Berry and Sampath Kannan for their early involvement in this project. We are also grateful to the Monarch group at CMU for making their code available to us; clearly this open code generosity was important to our study. This research was partially supported by: ARO DAAG55-98-1-0393, ARO DAAG55-98-1-0466, DARPA Contract F30602-98-2-0198, NSF CCR-9619910, ONR N00014-97-1-0505 (MURI), NSF CCR-9988409, NSF CISE-9703220, and DARPA ITO MOBIES F33615-00-C-1707.

## References

[1] B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[2] Debra Anderson, Thane Frivold, and Alfonso Valdes. Next-generation intrusion detection expert system (NIDES) : A summary. Technical report, SRI, May 1995. SRI-CSL-95-07.

[3] 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. `http://www.cis.upenn.edu/verinet/papers/tool-assessment.ps`.

[4] 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. `http://www.cis.upenn.edu/verinet/papers/taxonomy.ps`.

[5] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter. Formal verification of standards for distance vector routing protocols, February 2000. `http://www.cis.upenn.edu/~hol/papers/rip.ps`.

[6] G.v. Bochmann and O. Bellal. Test result analysis with respect to formal specifications. In *Proc. 2-nd Int. Workshop on Protocol Test Systems, Berlin*, pages 272–294, October 1989.

[7] G.v. Bochmann, D. Desbiens, M. Dubuc, D. Ouimet, and F. Saba. Test result analysis and validation of test verdicts. In *Proc. Workshop on Protocol Test Systems (IFIP)*, 1990.

[8] G.v. Bochmann, R. Dssouli, and J.R. Zhao. Trace analysis for conformance and arbitration testing. *IEEE Tr. on Soft. Eng.*, 15(11):1347–1356, November 1989.

[9] Josh Broch, David A. Maltz, David B. Johnson, Yih-Chun Hu, and Jorjeta Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *Proceedings of the Fourth Annual ACM/IEEE International Conference on Mobile Computing and Networking*, October 1998.

[10] Monica Brockmeyer, Farnam Jahanian, Constance Heitmeyer, and Bruce Labaw. A flexible, extensible environment for testing real-time specifications. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS)*, 1997.

[11] Laura K. Dillon and Q. Yu. Oracles for Checking Temporal Properties of Concurrent Systems. In *Proceedings of the 2nd ACM SIGSOFT Symposium on Foundations of Software Engineering*

*(SIGSOFT'94)*, volume 19, pages 140–153, December 1994. Proceedings published as Software Engineering Notes.

[12] S.A. Ezust and G.v. Bochmann. An Automatic Trace Analysis Tool Generator for Estelle Specifications. *Computer Communication Review*, 25(4):175–184, October 1995. Proceedings of ACM SIGCOMM 95 Conference.

[13] Kevin Fall and Kannan Varadhan. *ns Notes and Documentation.* The VINT Project, February 2000.

[14] Constance Heitmeyer, Alan Bull, Carolyn Gasarch, and Bruce Labaw. SCR*: A toolset for specifying and analyzing requirements. In *Proc. of COMPASS*, 1995.

[15] Ahmed Helmy and Deborah Estrin. Simulation-based 'STRESS' Testing Case Study. In *Sixth International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, July 1998.

[16] L. J. Jagadeesan, A. Porter, C. Puchol, J. C. Ramming, and L.G.Votta. Specification-based testing of reactive software: Tools and experiments. In *Proceedings of the International Conference on Software Engineering*, May 1997.

[17] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. Formally specified monitoring of temporal properties. In *Proceedings European Conference on Real-Time Systems*, 1999.

[18] I. Lee, S. Kannan, M. Kim, O. Sokolsky, and M.Viswanathan. Runtime assurance based on formal specifications. In *Proceedings International Conference on Parallel and Distributed Processing Techniques and Applications*, 1999.

[19] Ulf Lindqvist and Phillip A. Porras. Detecting computer and network misuse through the production-based expert system toolset (P-BEST). In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, Oakland, California, May 1999.

[20] T.O. O'Malley, D.J. Richardson, and L.K. Dillon. Efficient Specification-Based Test Oracles. In *Second California Software Symposium (CSS'96)*, April 1996.

[21] Charles Perkins. Ad hoc on-demand distance vector (AODV) routing. Internet-Draft Version 00, IETF, November 1997.

[22] 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.

[23] Phillip A. Porras and Peter G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In *National Information Systems Security Conference*, 1997.

[24] D.J. Richardson, S. Leif Aha, and T.O. O'Malley. Specification-Based Oracles for Reactive Systems. In *14th International Conference on Software Engineering*, May 1992.

[25] Mahesh Viswanathan. *Foundations for the Run-time Analysis of Software Systems.* PhD thesis, University of Pennsylvania, December 2000.

# A The MEDL specification of AODV properties

To illustrate the use of MEDL in specification of AODV properties, we show a complete MEDL script that contains all the properties that are discussed in the paper. These are the MonSeqNo, DestRep, DetectRErr, NodeRep and LoopInv properties.

```
ReqSpec AODVSpec

/* imported events: packet fields */
import event atnode, fordest, src, src_seq, src_hc, dest, dest_seq, bcastid;
import event prev, next_hop, init, eventty, pktty, pkt_rcv;

/* state variables for each pair of nodes */
var int best_seq[at][dst], best_hc[at][dst], best_next[at][dst];

/* packet with routing information is detected */
event routeinfo[at][dst] = pkt_rcv when
                          ((value(atnode,0)==at) && (value(src,0)==dst) &&
                           (value(pktty,0) > 0))
/* packet with routing information is received by node at */
event recvroute[at][dst] = routeinfo[at][dst] when ((value(eventty,0) == 0) &&
                                                    (value(src_hc) < 255));
/* a better route is received by node at */
event recvbetter[at][dst] = recvroute[at][dst]
                            when ((value(src_seq,0) > best_seq[at][dst]) ||
                                 ((value(src_seq,0) == best_seq[at][dst]) &&
                                  (value(src_hc,0) < best_hc[at][dst])));
/* route error received */
event recverr[at][dst] = routeinfo[at][dst] when ((value(eventty,0) == 0) &&
                                                  (value(pktty,0) == 2) &&
                                                  (value(src_hc,0) >= 255));
/* received route error means that route is broken */
event recvbettererr[at][dst] = recverr[at][dst] when
                            when (value(src_seq,0) > best_seq[at][dst]);

/* packet with routing information is sent by node at */
event sendroute[at][dst] = routeinfo[at][dst] when ((value(eventty,0) == 1) &&
                                                    (value(src_hc,0) < 255));
/* route error sent */
event senderr[at][dst] = routeinfo[at][dst] when ((value(eventty,0) == 1) &&
                                                  (value(pktty,0) == 2) &&
                                                  (value(src_hc,0) >= 255))
/* initial route error packet is sent by node at */
event sendiniterr[at][dst] = senderr[at][dst]
                          when (best_hc[at][dst] < 255);

/* increased sequence number is sent by destination dst */
event sendincseq[at][dst] = sendroute[at][dst]
                         when ((dst == at) &&
                               (value(src_seq,0) > best_seq[at][at]));
```

```
/* Alarm: sequence number sent by a node has decreased */

alarm MonSeqNo[at][dst] = sendroute[at][dst]
                          when ((dst == at) &&
                                (value(src_seq,0) < best_seq[at][at]));


/* Alarm: destination's reply has wrong hop count */

alarm DestRep[at][dst] = sendroute[at][dst] when
                                ((dst==at) &&
                                  (value(src_hc,0) > 0));


/* Alarm: unsolicited route error message does not have
   incremented sequence number */

alarm DetectRErr[at][dst] = sendiniterr[at][dst] when
                                 (value(src_seq,0) != best_seq[at][dst]+1);

/* Alarm: node does not send best route */

alarm NodeRep[at][dst] = sendroute[at][dst] when
                               ((dst != at) &&
                               ((value(src_hc,0) != best_hc[at][dst] + 1) ||
                                (value(src_seq,0) != best_seq[at][dst])));

/* Alarm: loop invariant is violated */

alarm LoopInv[at][nxt][dst] =
        start((at!=nxt) &&
              (at!=dst) &&
              (nxt!=dst) &&
              (best_next[at][dst] == nxt) &&
              ((best_seq[at][dst] > best_seq[nxt][dst]) ||
               ((best_seq[at][dst] == best_seq[nxt][dst]) &&
                (best_hops[at][dst] <= best_hops[nxt][dst]))))


/* initialization: reset all state variables */
init -> {
  best_seq[at][dst]' = 0; best_hc[at][dst]' = 0; best_next[at][dst]' = 0;
}

/* new route established: update checker state */
recvbetter[at][dst] -> {
 best_seq[at][dst]' = value(src_seq,0);  best_hc[at][dst]' = value(src_hc,0);
 best_next[at][dst]' = value(prev,0);
}

/* route error received: update checker state */

recvbettererr[at][dst] -> {
```

```
 best_seq[at][dst] = value(src_seq,0);
 best_hc[at][dst] = 255;
 best_next[at][dst] = 0;

}

/* dst(at) sent a larger sequence number: update checker state */
sendincseq[at][dst] -> { best_seq[at][at]' = value(src_seq,0); }

/* at sent a route error: update checker state */
sendiniterr[at][dst] -> {
 best_seq[at][dst]'  = value(src_seq,0); best_hc[at][dst]' = 255;
 best_next[at][dst]' = 0;
}

End
```

## A.1 Tuning

**Step I** We change DestRep as follows

```
alarm DestRep[at][dst] = sendroute[at][dst] when
                                ((dst==at) &&
                                  (value(src_hc,0) != 1));
```

**Step II** We change the DetectRerr alarm, and the event-handling conditions for route error packets as follows

```
alarm DetectRErr[at][dst] = senderr[at][dst] when
                                ((best_hc[at][dst] < 255) &&
                                 (value(src_seq,0) != best_seq[at][dst]));

event recvbettererr[at][dst] = recverr[at][dst] when
                            when (value(prev,0) == best_next[at][dst]);
```

**Step III** We change the state machine as follows

```
sendiniterr[at][dst] -> {
 best_seq[at][dst]'  = value(src_seq,0);
 best_next[at][dst]' = 0;
}

recvbettererr[at][dst] -> {
 best_seq[at][dst] = value(src_seq,0);
 best_next[at][dst] = 0;
}
```

# B The AODV State Machine

The AODV Specification [21] is an evolving document published by the MANET working group at the IETF (`http://www.ietf.org`). The document describes the various packets and network events that an AODV process needs to respond to. Here we present the reactive state machine that an implementation of AODV version 0 is supposed to implement. There are two control states corresponding to the presence or absence of a route to the destination. In addition, for each destination AODV keeps track of the best known route: seq_no, hop_cnt, next_hop, and lifetime. An AODV node runs a state machine for each destination; the state machine for the destination dst is shown in Table B. We have left out some details of timouts and link error events, which the protocol needs to handle as well. The state machine presented here captures the major packet events and their relation to the state at an AODV process.

| **STATE: No Route** | | |
|---|---|---|
| **Condition** | **Action** | **Next State** |
| TimeOut | seq_no $\leftarrow 0$ | No Route |
| Recv from $p$:<br>RREQ($d$, hops_to_src, dest_seq_no, $s$, src_seq_no)<br>$\wedge\ d =$ dst | dest_seq_no $\leftarrow \max($seq_no, dest_seq_no$)$;<br>Broadcast RREQ($d$, hops_to_src $+1$,<br>dest_seq_no, $s$, src_seq_no) | No Route |
| Recv from $p$:<br>RREQ($d$, hops_to_src, dest_seq_no, $s$, src_seq_no)<br>$\wedge\ s =$dst $\wedge$ src_seq_no $\geq$ seq_no | next_hop $\leftarrow p$; hop_cnt $\leftarrow$ hops_to_src $+1$;<br>seq_no $\leftarrow$ src_seq_no;<br>lifetime $\leftarrow$REV_ROUTE_LIFE; | Has Route |
| Recv from $p$:<br>RREP(hops_to_dest, $d$, dest_seq_no, route_lifetime)<br>$\wedge\ d =$dst $\wedge$ dest_seq_no $\geq$ seq_no | next_hop $\leftarrow p$; hop_cnt $\leftarrow$ hops_to_dest $+1$;<br>seq_no $\leftarrow$ dest_seq_no;<br>lifetime $\leftarrow$ route_lifetime | Has Route |

| **STATE: Has Route** | | |
|---|---|---|
| **Condition** | **Action** | **Next State** |
| TimeOut | seq_no $\leftarrow$ seq_no $+1$;<br>next_hop $\leftarrow 0$; hop_cnt $\leftarrow 255$<br>Send to active neighbors:<br>RREP(255,dst, seq_no,<br>BAD_LINK_LIFETIME) | No Route |
| Recv from $p$:<br>RREQ($d$, hops_to_src, dest_seq_no, $s$, src_seq_no)<br>$\wedge\ s =$dst<br>$\wedge$ [src_seq_no, hops_to_src] is better than<br>[seq_no, hop_cnt] | next_hop $\leftarrow p$;<br>hop_cnt $\leftarrow$ hops_to_src $+1$;<br>seq_no $\leftarrow$ src_seq_no;<br>lifetime $\leftarrow$REV_ROUTE_LIFE | Has Route |
| Recv from $p$:<br>RREP(hops_to_dest, $d$, dest_seq_no, route_lifetime)<br>$\wedge\ d =$dst<br>$\wedge$ [dest_seq_no, hops_to_dest] is better than<br>[seq_no, hop_cnt] | next_hop $\leftarrow p$;<br>hop_cnt $\leftarrow$ hops_to_dest $+1$;<br>seq_no $\leftarrow$ dest_seq_no;<br>lifetime $\leftarrow$ route_lifetime | Has Route |
| Recv from $p$:<br>RREP($255$, $d$, dest_seq_no, route_lifetime)<br>$\wedge\ d =$dst<br>$\wedge$ dest_seq_no $>$ seq_no | next_hop $\leftarrow 0$;<br>hop_cnt $\leftarrow 255$;<br>seq_no $\leftarrow$ dest_seq_no;<br>lifetime $\leftarrow$BAD_LINK_LIFETIME | Has Route |
| Recv from $p$:<br>RREQ($d$, hops_to_src, dest_seq_no, $s$, src_seq_no)<br>$\wedge\ d =$ dst $\wedge$ dest_seq_no $\leq$ seq_no | Unicast from me for $s$ :<br>RREP(hop_cnt, $d$, seq_no,<br>MY_ROUTE_TIMEOUT) | Has Route |
| Recv unicast from $p$ for $dst$: DATA | Send to next_hop : DATA | Has Route |
| Recv unicast from $p$ for $dst$:<br>RREP(hops_to_dest $+1$, $d$, dest_seq_no,<br>route_lifetime) | Send to next_hop :<br>RREP(hops_to_dest, $d$,<br>dest_seq_no, route_lifetime) | Has Route |