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

June 6, 2000

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 Java 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.

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. 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, with the flexible trace-checking component of the Java MaC system. 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) from Java 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.

In this paper, we provide an overview of the MaC framework for system analysis, describe its instantiation in Verisim, and illustrate the application of Verisim on 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 turnaround 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 ϕ and the checker produces as its output a *metatrace* T^{ϕ} of alarms indicating violations of ϕ 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 ϕ 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 ϕ. 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 turnaround 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 for 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 mixture of manual and automated techniques we call *tuning*. The metatrace T^{ϕ} is manually inspected to find bug classes and then the MEDL property ϕ is modified or 'tuned' to produce a formula ψ 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^{ψ}, which is inspected for new bugs. Note that the second step can proceed without rerunning the simulation, since it precedes from the original

*Contact author: gunter@cis.upenn.edu

trace. This strategy is repeatedly applied until it becomes desirable to fix a collection of bugs and rerun the trace.

The paper is divided into eight sections. After this introduction we describe in three sections the MaC framework, the NS system, and Verisim. Then, in the fifth section we describe the AODV protocol. Simulator code for this protocol is then analyzed in two case studies in the sixth section. The seventh section discusses some of the related work, and the eighth section concludes.

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 1. The first component of the MaC framework is the *Monitor*, which extracts a *Trace*; this trace is input to the second component, the *Checker*, which uses it to produce a *Metatrace*. The metatrace is examined by a human or another program to determine system status, including deviations from expected behavior. Inputs to the monitor include the *Program* and its *Data* (that is, inputs from its environment) and possibly a *Monitoring Script* which aids the trace extraction. The goal of the checker is to find deviations of the trace from a set of *Properties* that are also input to the checker.

The MaC framework provides an architecture for analyzing systems formally and flexibily using runtime information. This is illustrated by a prototype implementation of the framework for Java programs described in [6, 7, 8] and illustrated in Figure 2. This system, which we will call *Java MaC* provides a general approach to the analysis of Java programs based on trace information obtained from instrumented Java programs. A distinguished feature of the system is the ability of its monitor component to aid the instrumentation of Java programs in order to produce traces. Very briefly, a *Primitive Event Definition Language (PEDL)* defines low-level events and is used to generate an instrumented Java byte code program. This program produces a *Partial State Sequence (PSS)*, which is supplied to an *Event Recognizer* to produce the desired trace.

Our focus in this paper is on the use of the *checker* component of Java MaC. The monitor component of Java MaC will be replaced by NS to obtain the Verisim system, which is the main focus of the paper. To understand Verisim it is therefore necessary to have some familiarity with the checker component of Java MaC, so we provide a brief explanation here and refer the reader to other sources for details. The checker is based on a *Meta Event Definition Language (MEDL),* which is used to express properties of traces. Violations of these properties are obtained by running the checker with MEDL properties and an input trace to produce a metatrace. We now overview MEDL.

MEDL is based on an extension of linear temporal logic (LTL) with auxiliary variables. Auxiliary variables are 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 'RREP should happen before the $5th$ occurence of RREQ'. Like in SCR [5], 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 ($\texttt{next_hop}_d$ == 2) holds for as long as $\texttt{next_hop}_d$ 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 remains unchanged between updates from the monitor. For events, the checker makes the dual assumption, namely, that no events (of interest) happen between updates.

Table 1: MEDL grammar

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

<G> ::= <E> -> <Statements>
```

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 it is a boolean condition defined on the auxiliary variables. Guards (G) are 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. Each world is, therefore, labeled by the time instant it corresponds to and the set of primitive conditions and events that are true at that instant. 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 defined naturally; so `!c` is true when `c` is false, `c1 && c2` is true only when both `c1` and `c2` are true, `c1 || c2` is true when either `c1` or `c2` is true, and `c1 => c2` is true if `c2` is true whenever `c1` is true. Conjunction (`e1 && e2`) and disjunction (`e1 || e2`) on events is defined similarly. Now, since conditions are true from some time until just before the instant when they become false, two events can naturally be associated with a condition, namely the instant when the condition becomes true (`start(c)`) and the instant when the condition becomes false (`end(c)`). Any pair of events define an interval of time, and forms a condition `[e1, e2)` that is true from event `e1` until `e2`. The event `e when c` is true if `e` occurs and condition `c` is true at that time instant. Finally, a

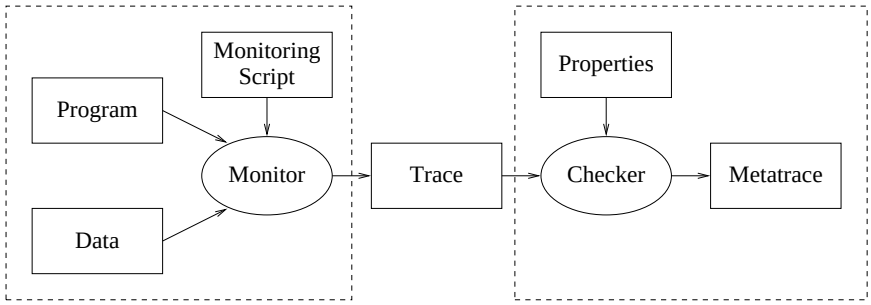

Figure 1: Overview of the MaC framework

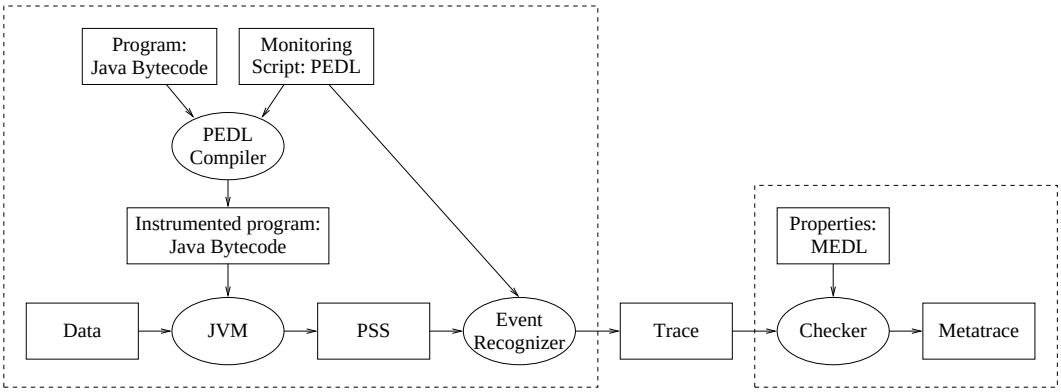

Figure 2: Overview of the MaC Prototype for Java

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`. The formal semantics for this logic is given in [6, 8].

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. The evaluation of individual events and conditions is fairly standard based on the semantics of the logic. However, there are 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. Hence, the checker must evaluate the events and conditions in a consistent order. In our implementation 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. Details of this algorithm can be found in [6].

3 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 results can inform the protocol design. 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 [4] 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 3. 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, 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 an OTcl 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

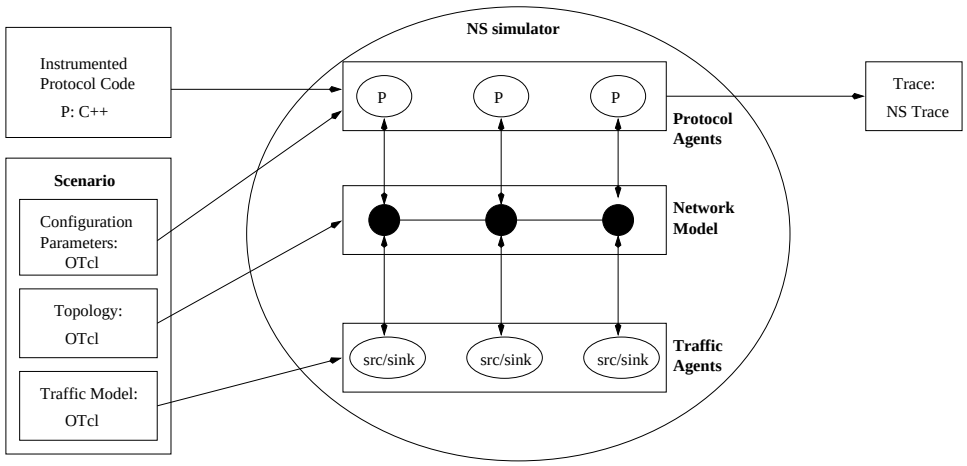

Figure 3: Simulations Using NSv2

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.

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. NS comes with a validation test suite 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 degaradation, 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. If they are not expected to happen, then their occurance in a simulation would be of interest, even if they did not impact performance in that particular scenario.

4 Verisim

Verisim is the integrated system obtained by using NS and the checker of Java MaC to provide the instantiation of the MaC framework depicted in Figure 4. The resulting integrated system enables flexible formal analysis of network simulations where properties are expressed in MEDL and checked on traces produced by NS.

The remainder of this paper is focused on 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. This protocol is described 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

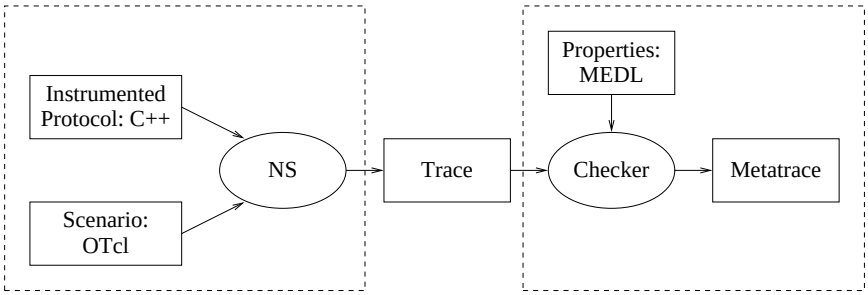

Figure 4: The Architecture of Verisim

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.

5 AODV Routing

This section describes the AODV routing protocol [10, 11] which we used in our case study. The first part provides a short protocol description of the protocol. The second part discusses some of its requirements—properties that are expected to hold in AODV implementations.

5.1 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 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.

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

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:

$$\mathsf{RREQ}(d, \mathsf{hops_to_src}, \mathsf{seq_no}, s, \mathsf{src_seq_no}).$$

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 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 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:

$$\mathsf{RREP}(\mathsf{hop_cnt}_d, d, \mathsf{seq_no}_d, \mathsf{lifetime}_d).$$

Arguments $\mathsf{hop_cnt}_d$, $\mathsf{seq_no}_d$, 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 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 hop_cnt 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 $\mathsf{hop_cnt} = 255$ (infinity), and its 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.

5.2 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 hard 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. These attributes are sometimes high-level requirements that can be asked about any protocol. A common requirement for a routing protocol is

Loop Freedom: Computed routes never contain loops.

Other examples include optimality of the routes, convergence properties, maximum route length, and so on. Other attributes are protocol-specific. These test whether a given implementation has expected behavior, usually with respect to the standard. For instance, the standard may prescribe that after some event, certain fields in the routing table must have positive values. In some cases we may want to test even a stronger hypothesis, stating that the standard is satisfied in a particular way (e.g. the value of the field is not only positive, but also an even number).

Here are examples of some AODV-specific properties:

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 Route: 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 hop_cnt 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 ($\mathsf{seq_no}_d$).

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.[2]

Detect Route Error: If a node detects a broken route, it should use $\mathsf{seq_no} = 1 +$ (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.

[2]This property is an important invariant that is sufficient (but not necessary) for loop freedom, as shown in [2].

6 Checking AODV Simulations

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

6.1 AODV properties in MEDL

Our first task is to translate properties given in section 5.2 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 2 shows a MEDL alarm that detects this violation in the observed state.

Table 2: Loop Invariant in MEDL

```
alarm LoopInv[at][nxt][dst] = sendroute[at][dst] when
        ((at!=nxt) && (at!=dst) && (nxt!=dst) &&
         (obs_nexthop[at][dst] == nxt) &&
         ((obs_seqno[at][dst] > obs_seqno[nxt][dst]) ||
          ((obs_seqno[at][dst] == obs_seqno[nxt][dst]) &&
           (obs_hopcnt[at][dst] <= obs_hopcnt[at][dst]))))
```

Event `sendroute[at][dst]` is generated whenever the node at sends a control packet containing the route information for dst (either a RREP carrying route information for dst, or a RREQ originated at dst). By inspecting the contents of the packet, we observe the route that at uses for dst. At this point the checker will re-evaluate the Loop Invariant condition to check for violations.

This will be our general strategy for translation—we convert the desired properties into alarms by negation. Table 3 shows properties and their corresponding MEDL alarm names.

Table 3: 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

6.2 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 [3]. 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 [10], 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 5.

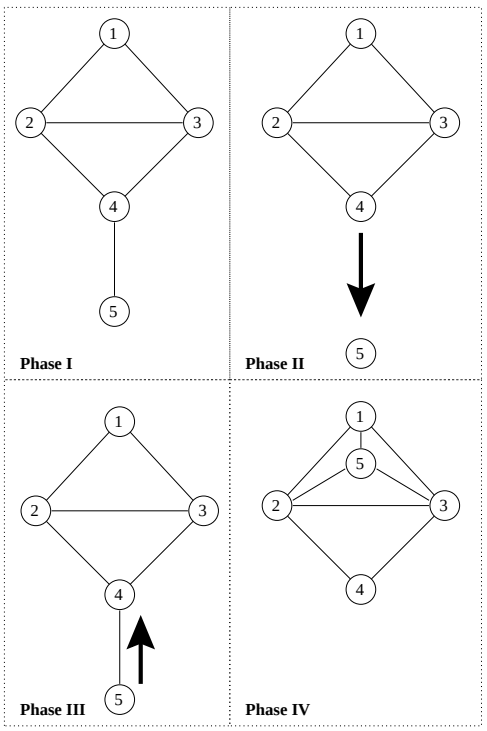

Figure 5: Scenario S

Topology: There are 5 nodes initially arranged as in Figure 5 (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 4. When a packet send or receive event happens at a node N, there is a line in the trace with the format:

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

Table 4: 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 ffffffff 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 ffffffff 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 ffffffff 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 ffffffff 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 ffffffff 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 ffffffff 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 ffffffff 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 ffffffff 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 ffffffff 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 ffffffff 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 ffffffff 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 ffffffff 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)
```

6.3 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 4). Verisim then checks whether T satisfies the AODV properties ϕ, and produces a meta-trace T^{ϕ} 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 5.

Table 5: RFB Alarms

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

Table 6: Typical Meta-trace T^{ϕ}

```
------------------------------------------------------------
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>
------------------------------------------------------------
```

Step I The first meta-trace T^{ϕ} contains 220 alarms, and the initial fragment is as shown in Table 6. 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^{ϕ} is raised at the last event of T shown in Table 4. 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^{ϕ} 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 ϕ to produce the second meta-trace T_1^{ϕ}. T_1^{ϕ} has 216 alarms, and is the same as T^{ϕ} 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^{ϕ}. T_2^{ϕ} 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 ϕ 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^{ϕ}.

Step IV The fourth meta-trace just contains one alarm, which is raised because of an unexpected buffering 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.

6.4 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 ϕ to get the meta-trace T^{ϕ}. As before, we start our analysis by looking at T^{ϕ}. However, when we find a bug, we tune our MEDL formula ϕ 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 7.

Step I As before the first alarm in T^{ϕ} 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 go into the MEDL formula and 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 ϕ_1 to get the meta-trace T^{ϕ_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^{ϕ_1} has 216 alarms and is the same as T^{ϕ} except that the DestRep alarms no longer

Table 7: Tuning Alarms

Meta-trace	DestRep	DetectRErr	NodeRep	LoopInv	Total alarms
T^{ϕ}	4	54	38	43	220
T^{ϕ_1}	0	54	38	43	216
T^{ϕ_2}	0	0	38	50	166
T^{ϕ_3}	0	0	21	0	30

appear. We see that the first alarm, DetectRErr, is again due to the inaccurate incrementing of sequence numbers at nodes. Having identified the error, we can modify the alarm to ignore this case. However, if a node fails to increment its sequence number on sending an unsolicited RREP, all future alarms are also affected because the node has an incorrect state. So we need to modify the MEDL formula not to increment its sequence number 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, ϕ_2, which is used to produce the meta-trace T^{ϕ_2}. Indeed, T^{ϕ_2} seems to not have the kinds of DetectRErr alarms and follow-up alarms as noticed before.

Step III T^{ϕ_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 conclude that the ways replies are generated in the protocol code must be 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 changing the conditions under which a RREP can be sent and allowing hop counts to stay even after the route has been lost. We run Verisim on this formula ϕ_3 and generate the fourth meta-trace T^{ϕ_3}.

Step IV The new meta-trace T^{ϕ_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 firm conclusions 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^{ϕ_3}. We repair P to produce a new implementation P_f., which is analyzed through Verisim to produce T_f^{ϕ}. T_f^{ϕ} has a total of 1 alarm due to packet buffering at a node.

6.5 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. The loop formation itself is not a very easy property to detect. 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.

7 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. (See [2] 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.

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) [1], which performs both statistical analysis and rule-based signature analysis on audit records and Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD) [12], which detects malicious activity through and across large networks. Although IDS's often focus on detecting statistical anomolies 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) [9] used in [1, 12] describes a rule as a pair of consisting of an antecedent and and a consequent on a factbase. The consequent can update the checker state by updating auxiliary variables and the antecedent recognizes certain conditions. We are currently investigating the expressiveness of such a language in order to compare it with MEDL.

8 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 debugging turnaround time. 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, and ONR N00014-97-1-0505 (MURI).

References

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

[2] 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`.

[3] 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.

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

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

[6] Moonjoo Kim, Mahesh Viswanathan, Hanêne Ben-Abdallah, Sampath Kannan, Insup Lee, and Oleg Sokolsky. A framework for run-time correctness assurance of real-time systems. Technical Report MS-CIS-98-37, University of Pennsylvania, 1998.

[7] 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.

[8] 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.

[9] 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.

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

[11] 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.

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