# Incorporating Bounded Model Checking in Network Simulation: Theory, Implementation and Evaluation

Ahmed Sobeih, Mahesh Viswanathan and Jennifer C. Hou

***Abstract*— Existing network simulators perform reasonably well in evaluating the performance of network protocols, but lack the capability of verifying the correctness of network protocols. In this paper, we present our ongoing research on extending J-Sim — an open-source, component-based compositional network simulation environment — with the model checking capability to explore the state space created by a network protocol in order to find a violation of a desirable safety property and/or to find a witness for a desirable liveness property if any exists. This paper shows how J-Sim can model-check the Ad-Hoc On-Demand Distance Vector (AODV) routing protocol, a fairly complex network protocol with thousands of lines of Java code. We also exploit protocol-specific properties in the process of exploring the state space, to reduce the size of the state space and to guide the (best-first) search towards paths that can potentially locate violations/witnesses in less time. The experimental results presented in this paper show that a best-first search strategy can provide several orders of magnitude reduction in both the time and space overheads needed to find violations/witnesses.**

***Index Terms*— Network Simulation, Protocol Verification, Model Checking.**

## I. Introduction

Modern data communication networks are extremely complex and do not lend well to theoretical analysis. With computer/network entities and techniques interacting and interfering with one another, theoretical network analysis can be rigorously made only after leaving out several (sometimes subtle) details that cannot be easily captured in the analysis [1], [2], [3], [4]. As a result, it may be more feasible to carry out simulation to study and evaluate the performance of network entities and protocols, and interaction among them. Several existing network simulators (e.g., ns-2 [5] and J-Sim [6]) provide an environment in which a network protocol designer can build a protocol prototype, evaluate its performance with respect to preselected networking metrics (e.g., system throughput, packet delivery ratio, and end-to-end delay) and re-design/refine the protocol if needed.

One major deficiency of existing network simulators is, however, that they only evaluate the performance of network protocols in several scenarios, but can *not* exhaustively analyze all possible scenarios for correctness. If all the corner cases do not appear (and hence cannot be investigated) in the scenarios studied, subtle errors in the protocol specification/implementation may not be easily located in the simulation, and may manifest themselves after the protocol has been implemented and deployed. This could lead to serious, and sometimes disastrous, problems. For example, a routing protocol that suffers from routing loops may cause data packets to loop in the network and not reach their destinations. Another example arises in the area of network security, where "holes" for security attacks may only be discovered after protocol implementation and deployment, causing severe damage to computer systems.

In the current practice, to check whether or not a network protocol contains any errors, a prototype that was specifically written for verification purposes has to be built (e.g., in an interactive theorem prover and/or a model checker). This process is an onerous, time-consuming and error-prone task. An interesting question is then whether or not we can employ a single, integrated tool to provide both the *performance evaluation* and *verification* of network protocols. With such a tool, only *one* prototype will be built and used for the two purposes, thus saving both the time and effort of network protocol designers.

Motivated thus, we have extended J-Sim [6] with capability to explore the state space created by a network protocol in order to find a violation of a desirable safety property and/or to find a witness for a desirable liveness property if any exists.

J-Sim [6] is an open-source network simulation and emulation environment developed entirely in Java. It is implemented on top of a component based software architecture, called the *autonomous component architecture (ACA)*, that closely mimics the digital-circuit design. The basic entities in the ACA are *components*, which communicate with one another via sending/receiving data at their *ports*. How components behave (in terms of how a component handles and responds to data that arrive at a port) is specified at system design time in *contracts*, but their binding does not take place until the time when the system is being "composed." With the separation of contract binding (at system design time) from component binding (at system integration time), a component can be individually implemented and tested independently. When data arrives at a port of a component, the component processes the data immediately in an independent execution context (e.g., *thread* in Java). The interference between different data handled simultaneously by the same component is thus minimal, and is only subject to synchronization and mutual exclusion (in order to ensure the integrity of shared data). In some sense, the ACA realizes the notion of *software IC* because of this *message-passing, independence execution* model [6], [7]. By closing the gap between hardware and software ICs, the ACA realizes the objectives of composability, extensibility and loose coupling between individual components [8]. All of these features enable new components to be included into J-Sim in a plug-and-play fashion. For these reasons, we chose J-Sim as the network simulator (rather than any other network simulator; e.g., ns-2 [9], *NEST* [10], *REAL* [11]) to be augmented with the protocol verification capability.

Ahmed Sobeih, Mahesh Viswanathan and Jennifer C. Hou are with the Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801. E-mail: {sobeih,vmahesh,jhou}@uiuc.edu.

In this paper, we present the theory, implementation and evaluation of using bounded model checking (BMC) [12], [13] in order to enable the J-Sim model checker to model-check a fairly complex network protocol with thousands of lines of Java code; namely, the Ad-Hoc On-Demand Distance Vector (AODV) routing protocol [14]. We implemented such a model checker (written in Java so that it can be readily integrated with J-Sim), and incorporated it into J-Sim in a way that enables the model checker to take control of the simulation of a network protocol in order to explore (almost) the entire state space, rather than just exploring one single execution path as J-Sim does. One important feature is that the implementation of the model checker does *not* require the core design and implementation of J-Sim to be altered. Furthermore, the model checker has been instrumented to take advantage of best-first search strategies that exploit properties inherent to the network protocol being checked, in order to guide the search towards paths that can potentially locate violations/witnesses in less time. We compare the performance of the various search strategies and show that the best-first search strategy can locate counterexamples[1] of a safety property in a more timely and space-efficient manner.

The rest of the paper is organized as follows. In Section II, we provide background information on formal reasoning and introduce concepts and definitions that lay out the theoretical basis of BMC. We give the implementation of BMC in J-Sim in Section III. Following that, we present our performance results in Section IV. Finally, we provide a taxonomy of related work in Section V and conclude the paper in Section VI with a list of future research work.

## II. Bounded Model Checking

In general, there are two basic approaches towards formal reasoning of software and hardware systems: *theorem proving* and *model checking*. In theorem proving, a formal technique (e.g., deductive methods and induction) is used to prove that the implementation of a system under study meets its specification. Model checking, on the other hand, checks a finite state machine model of the system in order to verify whether a (safety or liveness) property holds. Model checking [15], [16], by state space exploration, starts from an initial state of the system and recursively generates successor system states by executing the transitions (e.g., events) of the system. This process continues until either the entire state space is explored or an error is discovered. (Hence, in systems of infinite state space, model checking is used for locating errors, rather than proving the absence of errors.)

As compared to theorem proving, model checking has several important advantages. First, it can be built into existing tools and automated. Second, model checking does not require a deep understanding of complex mathematical concepts. Third, when the desired property fails to hold, a model checker provides a counterexample of the sequence of events that led to the error, thus helping in understanding why the error occurred and how it can be fixed.

[1]A counterexample is a sequence of states starting from the initial state and ending with the state in which the safety property is violated. A formal definition will be given later in the paper.

One of the major challenges of using model checking is the well-known *state space explosion* problem, i.e., the state space of the system can be so prohibitively large that a model checker may run out of memory. Several approaches to handling the state space explosion problem (e.g., partial order reduction, abstraction, just to name a few) can be found in [15]. In principle, bounded model checking (BMC) aims at solving the state explosion problem by exploring the state space up to a (configurable) maximum depth in order to disprove a safety property and/or to prove a liveness property. In what follows, we formally define concepts that lay out the theoretical basis of BMC.

*Definition:* The *set of system variables* $G = \{v_1, v_2, ..., v_k\}$ is a finite set of variables which describe states of a system or appear in formulas specifying properties of a system. Each system variable $v_i \in G$ has a type, which may be either simple (e.g., boolean, integer, etc.) or composite (e.g., array of integers, class, array of classes, etc.). The type of a variable indicates the domain over which the variable ranges.

*Definition:* A *state* $s$ is a function that assigns to each system variable $v \in G$ a value $s(v)$ over its domain. We denote by $S$ the set of all states. It should be noted that although $G$ is finite, $S$ may be infinite because the domains over which the system variables range may be infinite.

*Definition:* An *expression* is constructed from the system variables and constants (e.g., 1) to which functions (e.g., +) are applied. An *atomic formula* consists of propositions (boolean variables) and formulas constructed by applying predicates (e.g., $>$) to expressions. A *state formula* (also called *assertion*) is constructed by applying boolean connectives (e.g., $\vee$, $\wedge$) and quantification to atomic formulas constructing first-order formulas. For a state formula $\varphi$ and a state $s$, we say $s \models \varphi$ if $\varphi$ holds on state $s$; i.e., if $\varphi$ evaluates to ***true*** in $s$. Otherwise, we say $s \not\models \varphi$.

*Definition:* A *transition* $t$ is a function $t : S \mapsto 2^S$ that maps each state $s \in S$ into a (possibly empty) set of states $t(s) \subseteq S$. Each state in $t(s)$ is called a *t-successor* of $s$. We say that the transition $t$ is *enabled* on the state $s$ if $t(s) \neq \emptyset$. Otherwise, we say that $t$ is *disabled* on $s$.

*Definition:* A *system* $P$ is defined as a tuple

$$P = < G, I, T >$$

where $G$ is a finite set of system variables; $I$ is the initial condition, which is a state formula characterizing all the initial states, i.e., states at which the system can start; A state is defined to be *initial* if it satisfies $I$; and $T$ is a finite set of transitions $T = \{t_1, t_2, ..., t_m\}$.

*Definition:* An *execution path* $\pi$ of a system $P$ is a function $\pi : \mathbb{N} \mapsto S$ that defines an infinite sequence of states $\pi(0), \pi(1), \pi(2), ...$ that satisfies the following two requirements:

1) Initiation: $\pi(0) \models I$; i.e., $\pi(0)$ is initial.
2) Consecution: $\forall j \in \mathbb{N}$, $\exists t \in T : \pi(j+1) \in t(\pi(j))$; i.e., the state $\pi(j+1)$ is a $t$-successor of the state $\pi(j)$.

Next, we define both the syntax and semantics of a *temporal logic* which allows specification of properties such as safety properties (ensuring that a desired property invariantly holds along an execution path) and liveness properties (ensuring that a desired property holds at least once on an execution

path). These properties are related to the infinite behavior of a system. Although there are different temporal logics (see [15]), we focus on *linear temporal logic* (LTL) [17] because of its widespread use and well-developed proof methods.

*Definition: LTL Syntax:* LTL formulas are built using state formulas, boolean connectives ($\neg$ (negation) and $\vee$ (disjunction)) and temporal operators (e.g., $\bigcirc$ (next), $\Box$ (henceforth) and $\Diamond$ (eventually)[2]). The LTL formulas are recursively defined as follows:

- If $\varphi$ is a state formula, then $\varphi$ is an LTL formula.
- If $\varphi$ and $\psi$ are LTL formulas, then $\neg\varphi$ and $\varphi \vee \psi$ are LTL formulas.
- If $\varphi$ is an LTL formula, then $\bigcirc\varphi$, $\Box\varphi$ and $\Diamond\varphi$ are LTL formulas.

The other boolean connectives, such as $\wedge$ (conjunction), $\rightarrow$ (implication) and $\leftrightarrow$ (equivalence) can be defined in terms of $\neg$ and $\vee$. Specifically, $\varphi \wedge \psi = \neg((\neg\varphi) \vee (\neg\psi))$, $\varphi \rightarrow \psi = (\neg\varphi) \vee \psi$ and $\varphi \leftrightarrow \psi = (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$.

*Definition: LTL Semantics:* LTL formulas are interpreted over an execution path. Given an execution path $\pi$ and an LTL formula $\varphi$, we specify the recursive definition for the notion of $\varphi$ holding at position $j \in \mathbb{N}$ in $\pi$, denoted by $(\pi, j) \vDash \varphi$ as follows:

- For a state formula $\varphi$, $(\pi, j) \vDash \varphi \iff \pi(j) \vDash \varphi$.
- $(\pi, j) \vDash \neg\varphi \iff (\pi, j) \nvDash \varphi$; i.e., not $(\pi, j) \vDash \varphi$.
- $(\pi, j) \vDash \varphi \vee \psi \iff (\pi, j) \vDash \varphi$ or $(\pi, j) \vDash \psi$.
- $(\pi, j) \vDash \bigcirc\varphi \iff (\pi, j+1) \vDash \varphi$. Therefore, $\bigcirc\varphi$ holds at position $j$ if and only if $\varphi$ holds at the next position $j+1$.
- $(\pi, j) \vDash \Box\varphi \iff \forall k \geq j, (\pi, k) \vDash \varphi$. Therefore, $\Box\varphi$ holds at position $j$ if and only if $\varphi$ holds at position $j$ and all following positions.
- $(\pi, j) \vDash \Diamond\varphi \iff \exists k \geq j, (\pi, k) \vDash \varphi$. Therefore, $\Diamond\varphi$ holds at position $j$ if and only if $\varphi$ holds at some position $k \geq j$.

Next, we formally define BMC. In principle, the concept of BMC is to consider only a *finite prefix* of an execution path. We focus on safety properties (LTL formulas of type $\Box\varphi$) and liveness properties (LTL formulas of type $\Diamond\varphi$) with state formulas $\varphi$. Such formulas have the advantage that they can be recursively defined by the following equations:

$$\Box\varphi = \varphi \wedge \bigcirc\Box\varphi,$$
$$\Diamond\varphi = \varphi \vee \bigcirc\Diamond\varphi.$$

This leads to the following equivalences:

$$(\pi, j) \vDash \Box\varphi \iff \bigwedge_{i=j}^{\infty} \pi(i) \vDash \varphi,$$
$$(\pi, j) \vDash \Diamond\varphi \iff \bigvee_{i=j}^{\infty} \pi(i) \vDash \varphi.$$

Although it is not feasible to consider all positions along an infinite execution path, given an execution path $\pi$ and a bound $k$, it is easy to see that the following implications hold:

$$\neg \bigwedge_{i=j}^{k} \pi(i) \vDash \varphi \Longrightarrow (\pi, j) \nvDash \Box\varphi,$$
$$\bigvee_{i=j}^{k} \pi(i) \vDash \varphi \Longrightarrow (\pi, j) \vDash \Diamond\varphi.$$

Therefore, it suffices to consider (sufficiently long) finite prefixes of a given execution path $\pi$ to disprove a safety property $\Box\varphi$ and to prove a liveness property $\Diamond\varphi$. In light of this observation, bounded model checking can be formally defined as follows:

*Definition: Bounded Model Checking (BMC):* Given a system $P = < G, I, T >$, a state $\pi(0) \vDash I$ and a state formula $\varphi$, we have the following:

- If for some $k$, the formula $\bigwedge_{i=0}^{k} \pi(i) \vDash \varphi$ is not satisfiable, then $(\pi, 0) \nvDash \Box\varphi$. In this case, the prefix $\pi(0), \pi(1), \ldots, \pi(k)$ is called a *counterexample*, which disproves the safety property $\Box\varphi$. We call $\pi(k)$ an *error state*.
- If for some $k$, the formula $\bigvee_{i=0}^{k} \pi(i) \vDash \varphi$ is satisfiable, then $(\pi, 0) \vDash \Diamond\varphi$. In this case, the prefix $\pi(0), \pi(1), \ldots, \pi(k)$ is called a *witness*, which proves the liveness property $\Diamond\varphi$. We call $\pi(k)$ a *witness state*.

In practice, BMC is performed by progressively increasing the bound until a counterexample or a witness is found. However, this only yields a semi-decision procedure because the validity of a formula can not be inferred from the absence of bounded length counterexamples. Nevertheless, completeness can be achieved for finite state systems [18] because for every finite state system, there exists a bound $K$ such that the absence of counterexamples of length $K$ or less implies that the formula holds.

## III. Implementation of Bounded Model Checking in J-Sim

In this section, we discuss how we implement the BMC framework in J-Sim.

There are three major design goals in building the BMC framework in J-Sim (or any other network simulators):

1) The core implementation of J-Sim should not be modified.
2) Only a minimal modification to the J-Sim code that implements the network protocol is required.
3) The network protocol designer should not be overburdened with the details of the model checking process. He/she should only provide the protocol specification (in particular, the assertions that should be maintained in the course of verification and validation) and define the state of the protocol being model-checked, the set of events that may trigger state transition, and how the events are handled. Overall, the role of the network protocol designer in model checking should be to provide sufficient protocol specifics for the checking process to proceed.

To realize the above design goals, we have implemented a model checker in Java as a component, in the ACA of J-Sim. The model checker checks a network protocol by executing the J-Sim simulation code of that network protocol *directly* and exploring the state space on the fly up to a maximum depth *MAX_DEPTH*. Figure 1 illustrates the overall framework of incorporating bounded model checking into J-Sim. As shown in Figure 1, the model checker interacts with instances of J-Sim classes, $P_1$, $P_2$, ... , $P_n$, that implement the entities comprising the network protocol being model-checked. The network protocol designer specifies the safety and liveness properties that need to be checked. The meaning of the safety and liveness properties depends on the network protocol itself.

[2]For the syntax and semantics of other temporal operators (e.g., $U$ (until) and $W$ (waiting-for)), the interested reader may refer to [17].

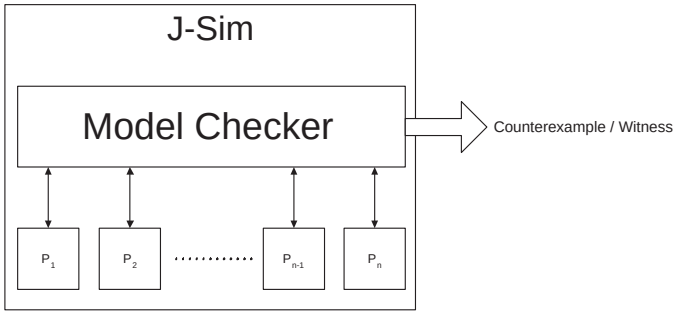


Fig. 1. Overall framework of bounded model checking in J-Sim.

For example, in case of a security protocol, a safety property is that unauthorized users do not get access to the system and a liveness property is that authorized users will be able to access the system (e.g., a denial of service attack does not take place). A state formula φ is implemented as a function (in Java) whose input is a state *s* and output is true/false. After the safety and liveness properties are properly specified and the state defined, the model checker starts from an initial state $\pi(0)$ and recursively generates successor system states by executing the transitions $T$ of the system. This process continues until either a counterexample disproving a safety property or a witness proving a liveness property is located (as shown in Figure 1) or the state space is explored up to the specified maximum depth.

*Definition of states:* In order to explore the state space created by a network protocol, the notion of the "state" has to be adequately defined. As mentioned in Section II, the definition of a state entails specifying the set of system variables *G*. To this end, we have defined and implemented another class, *GlobalState*, which includes the set of system variables and the depth of the state as data members. A state is an instance of the *GlobalState* class that assigns to each data member (i.e., system variable) a value as explained in Section II. It should be noted that the implementation of *GlobalState* differs from one network protocol to another; hence, it is the responsibility of the protocol designer to provide an implementation of *GlobalState*. In addition, the protocol designer should specify the initial condition $I$ and construct an initial state $\pi(0) \models I$.

The model checking procedure, shown in Figure 2, keeps track of three instances of *GlobalState*; namely, *initialState* (the initial state $\pi(0)$ of the network protocol), *currentState* (the current state being explored) and *nextState* (one of the possible successors of the current state). As shown in Figure 2, the two major data structures are *NonVisitedStates* (which stores the states that have not yet been visited) and *AlreadyVisitedStates* (which stores the states that have already been visited). Figure 2 presents a stateful search that avoids visiting a state that has already been visited before (i.e., a state that already exists in *AlreadyVisitedStates*).

*State transition:* In each state in the state space, some transitions (e.g., events) may or may not occur. Examples of events in a network protocol are: packet arrival, packet loss and timeout. It is also the responsibility of the network protocol designer to specify (a) the set of transitions $T$ that exist in the network protocol, (b) when each transition occurs (e.g., a packet arrival event occurs at a node *n* only if the network contains a packet whose destination is *n*), and (c) how each event is handled (i.e., an event handler function that makes a transition from one state to another). Note that the network protocol designer has to write the event handlers in order to have a working prototype of the network protocol in J-Sim, even if he/she does not intend to model check the protocol.

To help the protocol designer in defining the events that trigger the state transition, we have made use of the *reflection* facilities [19] of the Java language and implemented a *Transition* class as follows:

```
class Transition
{
   /* check if the event can take place */
   java.lang.reflect.Method  EnablingFunction ;

   /* define how each event is handled */
   java.lang.reflect.Method  EventHandler ;
   ...............
}
```

For each possible event in the network protocol, the protocol designer needs only to (1) create an instance of the *Transition* class, (2) use the *java.lang.Class.getMethod()* function to return a *Method* object that reflects the event's enabling function and event's event handler, and (3) use the *java.lang.reflect.Method.invoke()* function to invoke the enabling function (Figure 2, line 19) and the event handler (Figure 2, line 24).

As mentioned in Section II, a transition $t$ maps a state $s$ into a set of states $t(s)$. Therefore, a transition may generate multiple successor states. For example, a packet arrival event may generate multiple successor states. This is because if the network contains two packets $m_1$ and $m_2$ whose destination is node *n*, two successor states can be generated depending on whether node *n* receives $m_1$ first and then $m_2$ or receives $m_2$ first and then $m_1$. On the other hand, a node reboot event generates only one successor state (namely, the state of the node after reboot). In the model checking procedure, the enabling function (Figure 2, line 19) should return the number of possible successor states (zero if the event is disabled), i.e., the cardinality of $t(s)$.

*The model checking process:* The model checking procedure shown in Figure 2 works in one of two modes: *DISPROVE_SAFETY* or *PROVE_LIVENESS*. In the former, the model checking procedure searches the state space for a counterexample. In the latter, the model checking procedure searches the state space for a witness. If the model checking procedure is in the *DISPROVE_SAFETY* mode, it first checks if a safety property is violated in the initial state (Figure 2, line 2). If so, the procedure terminates after notifying the protocol designer that the initial state violates a safety property. Similarly, if the model checking procedure is in the *PROVE_LIVENESS* mode and a liveness property holds in the initial state (Figure 2, line 7), the procedure terminates after notifying the protocol designer that the initial state satisfies a liveness property.

If the initial state does not violate a safety property or satisfy a liveness property, the model checking procedure starts with an empty *AlreadyVisitedStates* (Figure 2, line 11) and *Non-*

```
Procedure modelCheck()
{
    /* The following are static data members of the ModelChecking class:
       AlreadyVisitedStates, NonVisitedStates, initialState, currentState, nextState.
       Depth of initialState = 0 */
 1. if ( OperationMode == DISPROVE_SAFETY ) { /* if searching for a counterexample */
 2.    if ( verifySafety(initialState) == false ) {
 3.       Print("modelCheck: Safety property is violated at the initial state.") ;
 4.       initialState.printState() ;  /* Print the error state */
 5.       exit ;
       } /* end if safety property is violated at the initial state */
    } /* end if searching for a counterexample */
 6. else if ( OperationMode == PROVE_LIVENESS ) { /* if searching for a witness */
 7.    if ( verifyLiveness(initialState) == true ) {
 8.       Print("modelCheck: Liveness property holds at the initial state.") ;
 9.       initialState.printState() ;  /* Print the witness state */
10.       exit ;
      } /* end if liveness property holds at the initial state */
    } /* end else if searching for a witness */
11. AlreadyVisitedStates = { } ;
12. NonVisitedStates = { initialState } ;
13. while ( | NonVisitedStates | > 0 ) {
14.    currentState = NonVisitedStates.removeFirst() ;
      /* Explore currentState only if it has not been visited before */
15.    if ( currentState does not exist in AlreadyVisitedStates ) {
16.       AlreadyVisitedStates = AlreadyVisitedStates ∪ { currentState } ;
17.       for ( all protocol entities p ) { /* for all the protocol entities in the network protocol */
18.          for ( all possible events e ) { /* for all events that may take place */
             /* Calculate the number of successor states that event e can generate for protocol entity p. */
             /* For example, how many packets are destined to node p. */
             /* Enabling function returns zero if event e is disabled. */
19.             NumberOfNextStates = e.EnablingFunction(p) ;
20.             for ( int i = 0 ; i < NumberOfNextStates ; i++ ) { /* for all possible successor states */
                /* Copy the relevant state information from currentState to the protocol entities */
21.                CopyFromModelToEntities(currentState) ;
22.                nextState = currentState ;  /* Start with nextState equal to currentState */
23.                nextState.depth += 1 ;  /* Increment the depth of nextState */
24.                e.EventHandler(p) ;  /* Invoke e's event handler */
               /* Copy the new relevant state information from the protocol entities to nextState */
25.                CopyFromEntitiesToModel(nextState) ;
26.                if (nextState does not exist in AlreadyVisitedStates) {
27.                   if ( OperationMode == DISPROVE_SAFETY ) { /* if searching for a counterexample */
28.                      if ( verifySafety(nextState) == false ) {
29.                         Print("modelCheck: FOUND ERROR STATE ") ;
30.                         nextState.printState() ;  /* Print the error state */
31.                         Print("COUNTEREXAMPLE ") ;
32.                         printPath(nextState) ;  /* Print the counterexample */
33.                         exit ;
                         } /* end if safety property is violated at the next state */
34.                      else {
35.                         if ( nextState.depth < MAX_DEPTH ) { /* Check if nextState needs to be explored */
36.                            NonVisitedStates = NonVisitedStates ∪ { nextState } ;
                            }
                         } /* end else */
                      } /* end if searching for a counterexample */
37.                   else if ( OperationMode == PROVE_LIVENESS ) { /* if searching for a witness */
38.                      if ( verifyLiveness(nextState) == true ) {
39.                         Print("modelCheck: FOUND WITNESS STATE ") ;
40.                         nextState.printState() ;  /* Print the witness state */
41.                         Print("WITNESS ") ;
42.                         printPath(nextState) ;  /* Print the witness */
43.                         exit ;
                         } /* end if liveness property holds at the next state */
44.                      else {
45.                         if ( nextState.depth < MAX_DEPTH ) { /* Check if nextState needs to be explored */
46.                            NonVisitedStates = NonVisitedStates ∪ { nextState } ;
                            }
                         } /* end else */
                      } /* end else if searching for a witness */
                   } /* end if nextState does not exist in AlreadyVisitedStates */
                } /* end for all possible successor states */
             } /* end for all the events that may take place */
          } /* end for all the protocol entities in the network protocol */
       } /* end if currentState has not been visited before */
    } /* end while NonVisitedStates is not empty */
47. Print("No error or witness states were found. Ending model checking") ;
48. exit ;
}
```

Fig. 2. Bounded and stateful on-the-fly model checking procedure.

*VisitedStates* initially contains the initial state only (Figure 2, line 12). As long as *NonVisitedStates* is not empty, the model checking procedure removes a state from *NonVisitedStates* and sets it to *currentState* (Figure 2, line 14) and starts exploring

*currentState* if it has not been visited before (Figure 2, line 15). For each state being explored (*currentState*), the model checking procedure generates all the possible successor states (*nextState*) by executing the event handlers of the events that can occur in *currentState*. However, since an event handler is only invoked from the model checking procedure but actually executed inside the protocol entities themselves, the model checking procedure must first restore the state of the protocol entities to the state reflected in *currentState* before the execution of the event handler. This is achieved by the *CopyFromModelToEntities()* function call (Figure 2, line 21). After the execution of the event handler (Figure 2 , line 24), the *CopyFromEntitiesToModel()* function is called (Figure 2, line 25) to extract the new state information from the protocol entities and copy them to *nextState*.

If *nextState* has not been visited before (Figure 2, line 26), the model checking procedure then checks whether *nextState* violates a safety property (Figure 2, line 28) or satisfies a liveness property (Figure 2, line 38) in the *DISPROVE_SAFETY* or *PROVE_LIVENESS* mode respectively. If not, *nextState* is added to *NonVisitedStates* (Figure 2, line 36 and line 46 respectively) in order to be explored later if its depth is strictly less than *MAX_DEPTH*; otherwise, a counterexample or a witness is printed by calling the *printPath()* function (Figure 2, line 32 and line 42 respectively). The *printPath()* function is a recursive function that traces the state space backwards from *nextState* until *initialState* is reached.

*Implementation:* We have encountered two major implementation problems in the course of incorporating the model checker into J-Sim: one is related to how network protocol entities communicate with each other, with the model checker in between; and the other is related to the ACA timers. We describe below each of them and how we solved them while keeping our design goals met.

Without model checking, protocol entities communicate with each other via ports. However, when the network protocol is model-checked and the model checker is used as shown in Figure 1, the protocol entities need to communicate with each other via the model checker. Initially, we simply connected the ports of each protocol entity to those of the model checker, but then found that protocol-specific data/control messages generated by the protocol entities during the execution of an event handler may not be forwarded to the model checker at the required time. This is because the model checker does not wait until the protocol entities finish executing an event handler. This may cause the model checker to exclude some of the new state's information in *nextState*. We solved this problem by setting the ports that are involved in the interaction between the model checker and the protocol entities to the *function-call execution model* instead of the default *independence execution model* [8]. In the function-call execution model, the model checker *waits* until the protocol entities finish executing an event handler; therefore, this solution ensures that all the new state's information will be included in *nextState*. Although this solution requires modest modification to the J-Sim simulation code of the network protocol, we believe that this modification is minimal. Alternatively, one may make the modification in a subclass of the J-Sim class of a network protocol entity, thus keeping the original parent J-Sim class unmodified.

The second problem is related to the ACA timers. Without model checking, a protocol entity that uses an ACA timer sets the timer to a pre-determined time interval. Upon timer expiration, a timeout event is triggered if the timer is still active. If the network protocol is to be model-checked, the model checker should explore all the possible transitions from a given state, and should not be limited to a single timeout value for each timer[3]. Instead, the model checker should trigger the timeout event when that event may occur in the real world. For example, a typical retransmission timer in a reliable unicast protocol may expire at any time as long as there is a pending data message that has been sent but not yet acknowledged. This problem can be easily solved by modifying the core implementation of J-Sim. However, this violates our first design goal. Therefore, we chose to make this modification also at a subclass of the J-Sim class of a network protocol entity.

## IV. Evaluation and Results

In this section, we explain how we use the model checking framework to model-check the implementation of the Ad-Hoc On-Demand Distance Vector (AODV) routing protocol in J-Sim. We first summarize the AODV protocol and then elaborate on how we built the AODV model. Following that, we present our experimental results.

### A. Ad-Hoc On-Demand Distance Vector (AODV) Routing

The Ad-Hoc On-Demand Distance Vector (AODV) routing protocol [20] is a well-known and widely-used reactive multihop routing protocol for ad hoc networks. AODV is reactive in the sense that a route to a given destination is established via a route discovery process only when it is needed by a source node (i.e., traffic-driven). In this section, we describe the implementation of AODV in J-Sim (which is based on AODV Draft (version 11) [14]).

Each node $n$ in the ad hoc network maintains a routing table. A routing table entry for a destination $d$ contains, in addition to other fields, a next hop address $nexthop_{n,d}$ (the address of the node to which packets destined to destination $d$ are forwarded), a hop count $hops_{n,d}$ (the number of hops needed to reach the destination $d$) and a destination sequence number $seqno_{n,d}$ (a measure of the freshness of the route information). Each routing table entry has a lifetime. Periodically, every *ROUTE_CACHE_FREQUENCY* seconds, a route timeout event is triggered causing the invalidation (but not deletion) of all the routing table entries that have not been used (e.g., to send or forward packets to the destination) for a time interval that is greater than the lifetime. Invalidating a routing table entry involves incrementing $seqno_{n,d}$ and setting $hops_{n,d}$ to $\infty$. If a new route to destination $d$ is offered to node $n$, node $n$ compares the destination sequence number of the offered route to the destination sequence number of the current route, and accepts the route with the greater sequence number. If the sequence numbers are the same, the offered route is accepted

[3]We assume that setting of the interval of a timer may differ from one run of the protocol to another; otherwise, this approach may suffer from excessive false positives.

only if it has a smaller hop count than the hop count in the routing table entry.

Each node $n$ also maintains two monotonically increasing counters: a node sequence number $seqno_n$ and a broadcast ID $bid_n$. When a node $n$ requires a route to a destination $d$, if it does not already have a routing table entry to the destination $d$, it creates an invalid routing table entry to the destination $d$ with $hops_{n,d}$ set to $\infty$. Node $n$ then *broadcasts* a route request (RREQ) packet containing the following fields $< n, seqno_n, bid_n, d, seqno_{n,d}, hopCount_q >$ and then increments $bid_n$. The $hopCount_q$ field is initialized to 1. The pair $< n, bid_n >$ uniquely identifies a RREQ packet.

When a node receives a RREQ, if it has already received a RREQ with the same source address and broadcast ID, it discards the RREQ; otherwise, it either satisfies the RREQ by *unicasting* a route reply (RREP) packet back to the requesting node if it has a fresh enough route to the destination (or it is the destination itself) or rebroadcasts the RREQ to its own neighbors after incrementing the $hopCount_q$ field if it does not have a fresh enough route to the destination (nor is it the destination itself). An intermediate node determines whether it has a fresh enough route to the destination by comparing the destination sequence number in its own routing table entry to the $seqno_{n,d}$ field in the RREQ packet. Each intermediate node also records a reverse route to the requesting node; this reverse route can be used to send/forward route replies to the requesting node. The requesting node's sequence number $seqno_n$ is used to maintain the freshness of the reverse route.

A RREP packet sent by an intermediate node $m$ contains the following fields $< hopCount_p, d, seqno_{m,d}, n >$. The $hopCount_p$ field is initialized to $1 + hops_{m,d}$. If it is the destination $d$ that sends the RREP packet, it first increments $seqno_d$ (i.e., $seqno_d \leftarrow seqno_d + 2$) and then sends a RREP packet containing the following fields $< 1, d, seqno_d, n >$. The unicast RREP travels back to the requesting node via the reverse route. Each intermediate node along the reverse route sets up a forward pointer to the node from which the RREP came, thus establishing a forward route to the destination, increments the $hopCount_p$ field and forwards the RREP packet to the next hop towards the requesting node. Each node receiving a RREQ keeps the pair $< n, bid_n >$ in a broadcast ID cache so that it can later check if it has already received a RREQ with the same source address and broadcast ID. Each entry in this cache has a lifetime. Periodically, every *BCAST_ID_SAVE* seconds, a broadcast ID timeout event is triggered causing the deletion of all the entries in the cache that have expired (i.e., remained in the cache for a duration that is at least equal to the lifetime).

In a mobile ad hoc network, the topology of the network may change due to node mobility. As the nodes move relative to each other, existing wireless links between nodes may get broken and new links may be established. The implementation of AODV in J-Sim supports two mechanisms to detect broken links: a link layer mechanism and a network layer mechanism. In the link layer mechanism, the link layer (e.g., IEEE 802.11) detects a broken link (e.g., by the absence of a link layer ACK each time a packet is transmitted to the next hop; or the failure to get a CTS after sending an RTS each time a packet needs to be transmitted to the next hop and the retry count exceeds the maximum retry limit). Upon detecting a broken link, the link layer notifies the AODV process. In turn, the AODV process executes a broken link event handler. Specifically, if the broken link is closer to the destination than the source, the node attempts a local repair by sending a RREQ to discover a route to the destination; otherwise, the node invalidates the routing table entries to all the destinations that have become unreachable due to the broken link, and broadcasts a route error (RRER) packet announcing the node IDs of all these unreachable destinations. On the other hand, in the network layer mechanism (which is optional in both the AODV Draft (version 11) [14] and the implementation of AODV in J-Sim), each node has a *neighbors list* that contains the node IDs of its neighbors. Neighboring nodes exchange HELLO packets to establish and maintain the neighborhood information. Each entry in the neighbors list has a lifetime. On receiving a HELLO packet, a node creates an entry (or refreshes the lifetime of an existing entry if one already exists) for the source node in the neighbors list. Periodically, every $HELLO\_INTERVAL$ seconds, a neighbor timeout event is triggered, causing the deletion of all the entries in the neighbors list that have expired.

### B. Building the AODV Model

In this section, we explain the steps that we followed in order to build the AODV model. These steps constitute a generic methodology for building a model for any other network protocol that needs to be model-checked in J-Sim.

*Definition of a state:* For AODV, we define *GlobalState* as a tuple that has two components; namely, the protocol state and the network cloud. The protocol state includes data members that represent the state of the protocol entities; i.e., the wireless nodes in the ad hoc network. Specifically, the protocol state includes each node's sequence number $seqno_n$, broadcast ID $bid_n$, routing table and the broadcast ID cache that keeps the information of the previously received RREQ packets. The network cloud models the network as a black box that contains the AODV packets (e.g., RREQ, RREP) and also maintains the neighborhood information. A broadcast AODV packet whose source is node $s$ is modeled as a set of packets, each of which is destined to one of the neighbors of $s$.

In the initial global state, the network does not contain any packets and each of the wireless nodes is initialized as specified by the constructor of the *AODV* class in J-Sim. Specifically, each wireless node starts with an empty routing table, empty broadcast ID cache, $seqno_n = 2$ and $bid_n = 1$.

*Safety and liveness properties:* In AODV, an important safety property is the *loop-free* property. Consider two nodes $n$ and $m$ such that $m$ is the next hop of $n$ to some destination $d$; i.e., $nexthop_{n,d} = m$. The loop-free property can be expressed formally in LTL notation as follows [21], [22]:

$$(\pi, 0) \models \Box \; ( \; (seqno_{n,d} < seqno_{m,d}) \vee (seqno_{n,d} == seqno_{m,d} \wedge hops_{n,d} > hops_{m,d})).$$

Informally, the loop-free property requires that either $m$ has a more fresh route to $d$ than $n$ (i.e., a route with a greater destination sequence number) or $m$ has a shorter route that is equally fresh.

A desirable liveness property in a routing protocol, such as AODV, is that nodes that require a route to a destination $d$ will eventually have a valid routing table entry to the destination $d$. This property can be expressed formally in LTL notation as follows:

$$(\pi,0) \models \bigwedge_{n \neq d} ReqRoute(n,d) \rightarrow \Diamond hops_{n,d} \neq \infty$$

where $ReqRoute(n,d)$ is true if node $n$ requires a route to a destination $d$. Assuming all nodes $n \neq d$ require a route to $d$, the liveness property can be stated as follows:

$$(\pi,0) \models \bigwedge_{n \neq d} \Diamond hops_{n,d} \neq \infty$$

*State transitions:* The following step is to specify the set of transitions, when each transition occurs, and how each transition is handled. The state transitions can be classified into two categories: node transitions (i.e., transitions that are triggered inside a node) and network transitions (i.e., transitions that are triggered inside the network). The transitions in each category can be listed as follows:

1) Node Transitions
   a) Initiation of a route request to a destination $d$: This event is enabled if the node does not have a valid routing table entry to the destination $d$. The event is handled by broadcasting a RREQ packet.
   b) Restart of the AODV process at node $n$: This event may take place because of a node reboot. The event is always enabled and is handled by reinitializing the state of the AODV process at node $n$. Specifically, the AODV process at node $n$ restarts with an empty routing table, empty broadcast ID cache, $seqno_n = 2$ and $bid_n = 1$.
   c) Broadcast ID timeout event at node $n$: This event is enabled if there is at least one entry in the broadcast ID cache of node $n$. The event is handled by deleting the entries in the broadcast ID cache that have expired. The broadcast ID timeout event may generate multiple successor states depending on which entries in the cache are deleted.
   d) Timeout of route to destination $d$ at node $n$: This event is enabled if node $n$ has a valid routing table entry to destination $d$ in its routing table. The event is handled by invalidating this routing table entry.
   e) Neighbor timeout event at node $n$: This event is enabled if there is at least one entry in the neighbors list of node $n$. The event is handled by deleting the entries in the neighbors list that have expired. The neighbor timeout event may generate multiple successor states depending on which entries in the neighbors list are deleted.
2) Network Transitions
   a) Delivering an AODV packet to node $n$: This event is enabled if the network contains at least one AODV packet such that $n$ is the destination of the packet (or the next hop towards the destination) and $n$ is one of the neighbors of the source of the packet. The event is handled by removing one of these packets from the network and forwarding it to the node $n$ in order to be processed according to the AODV implementation. As mentioned above, this is an example of an event that may generate multiple successor states.
   b) Loss of an AODV packet destined to node $n$: This event is enabled if the network contains at least one AODV packet such that $n$ is the destination of the packet. The event is handled by removing one of these AODV packets from the network. Similar to the reception of an AODV packet event, this event may generate multiple successor states depending on which packet in the network is lost.

### C. Model Checking the AODV Protocol

In this section, we discuss how we made use of protocol-specific properties and heuristics to facilitate a best-first search strategy and to handle the state space explosion problem. In addition, we present several counterexamples and witnesses that the model checker can find, and discuss the effect of the search strategy on the time and space overheads needed to find these counterexamples and witnesses. All of the experiments presented in this section were run on a Pentium 4 2.66 GHz machine running Red Hat Linux 8.0 kernel version 2.4.18 with 1 GB memory.

*Use of protocol-specific properties to facilitate a best-first search strategy:* While model checking AODV, we made use of three search strategies in exploring the state space: breadth-first, depth-first and best-first. Although breadth-first search is the most straightforward search strategy and is guaranteed to find the shortest path to an error/witness, it may run out of memory before reaching any error/witness state. Depth-first search tries to alleviate this problem at the price of finding a (possibly) longer counterexample/witness in (possibly) more time. The purpose of best-first search is to explore states that may potentially lead to an error/witness state first, thereby reducing the time needed to locate error/witness states. Protocol-specific metrics are exploited to specify how a state is considered "better" or more likely to lead to an error/witness state.

A suitable best-first search strategy for exploring the state space of AODV can be obtained by inspecting the loop-free property stated in Section IV-B. A node, which does not have a routing table entry to the destination $d$ or has an invalid routing table entry to the destination $d$, does not affect the truth value of the loop-free property. Therefore, a suitable best-first search strategy (which we will call Best-first Search 1) is to consider a state $s_1$ better than another state $s_2$ if the number of *valid* routing table entries in $s_1$ is greater than the corresponding quantity in $s_2$. This best-first search strategy turns out to be also useful when searching for a witness for the liveness property stated in Section IV-B as will be shown later.

In order to study the effect of the choice of the best-first search strategy, we investigate another best-first search strategy (which we will call Best-first Search 2) that was also obtained by inspecting the loop-free property stated in Section IV-B. The loop-free property can be rewritten as follows:

$$(\pi,0) \models \Box\ (\ ((seqno_{n,d} - seqno_{m,d}) < 0) \vee (seqno_{n,d} == seqno_{m,d} \wedge ((hops_{m,d} - hops_{n,d}) < 0)))\ .$$

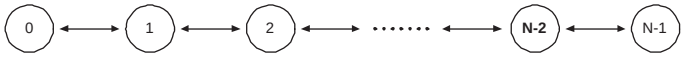


Fig. 3. An ad hoc network consisting of a chain of N nodes.

Therefore, the greater the quantity $(seqno_{n,d} - seqno_{m,d})$ and/or the greater the quantity $(hops_{m,d} - hops_{n,d})$ in a state, the more likely this state is close to an error state. Hence, in Best-first Search 2, we consider a state $s_1$ better than another state $s_2$ if the following summation

$$S = \sum_{n \neq d}((seqno_{n,d} - seqno_{m,d}) + (hops_{m,d} - hops_{n,d}))$$

is greater in $s_1$ than the corresponding summation in $s_2$, where $nexthop_{n,d} = m$. The summation $S$ includes only the nodes $n$ and $m$ that have valid routing table entries to the destination $d$. If none of the nodes have a valid routing table entry to the destination $d$, $S$ is set to $-\infty$ (i.e., least interesting).

*Use of protocol-specific heuristics to handle the state space explosion:* Obviously, the state space of the AODV protocol is infinite. Furthermore, there is an infinite number of possible initial states depending on the number of nodes and the neighborhood information. In order to handle the state space explosion problem, we made use of the following heuristics:

1) For the initial state, we consider two ad hoc network topologies only: a chain topology (Figure 3) and a grid topology (Figure 4). All wireless links are assumed to be bidirectional. In Figures 3- 4, a bidirectional arrow exists between nodes $i$ and $j$ if and only if $i$ is within the transmission radius of $j$ and $j$ is within the transmission radius of $i$; i.e., both $i$ and $j$ can reach each other in one hop.
2) We consider only one destination node $d$; namely, $d = N-1$ in the chain topology and $d = M^2 - 1$ in the grid topology. Therefore, all RREQ packets request a route to $d$ and the route timeout event invalidates the routing table entry to $d$ only. Furthermore, the loop-free property checks the absence of routing loops to $d$ only and the liveness property requires the presence of valid routing table entries to $d$ only at all the other nodes.
3) Since the network layer mechanism of detecting broken links is optional in both the AODV Draft (version 11) [14] and the implementation of AODV in J-Sim, we ignore the neighbor timeout event. This reduces both the number of events and the number of states in the state space.

Clearly, pruning the state space due to ignoring events and/or states and exploring the state space up to a maximum depth may cause the model checker to miss errors/witnesses (e.g., errors that may manifest themselves only in large networks, but not in small networks). However, in spite of pruning the state space, the model checker was still able to find interesting errors and witnesses as explained below.

*Effect of the search strategy:* In this subsection, we study the effect of the search strategy on the performance of the model checker with respect to the time and space overheads required for finding three counterexamples and one witness. The initial state of the ad hoc network is a chain topology consisting of

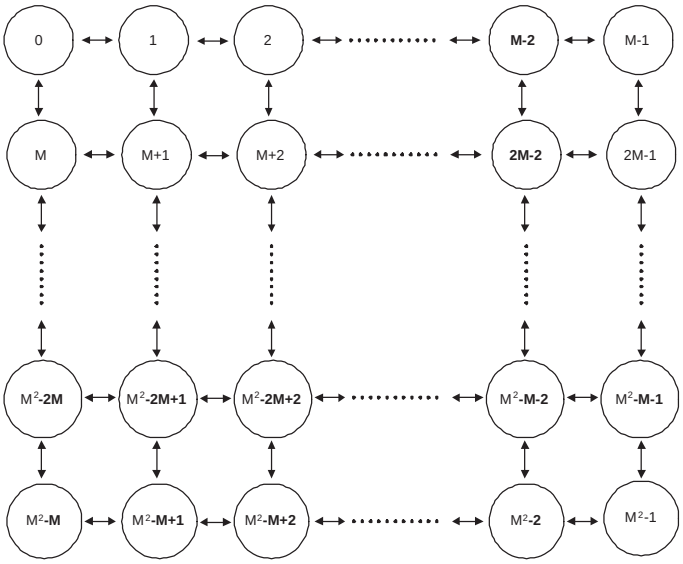


Fig. 4. An ad hoc network consisting of a grid of M $\times$ M nodes.

3 nodes: $n0$, $n1$ and $n2$. Although this initial state is simple, it ensures that $n0$ requires a multihop route to reach $n2$; i.e., AODV multihop routing is needed. In addition, if an error may occur in a small network, it may also occur in a large network. $MAX\_DEPTH$ was set to 15.

**Counterexample 1** - Using a breadth-first search strategy with all the state transitions mentioned in Section IV-B, the model checker finds an error in the J-Sim implementation, whose trace is shown in Figure 5, in 65.79 minutes. When the packet loss and route timeout events are ignored, the breadth-first search strategy finds the error in 13.45 minutes. Table I and Table II respectively give the time overhead and the space overhead (in terms of the number of states in the stateful search) needed to locate the error for a breadth-first search strategy. As shown in Tables I- II, ignoring some events (namely, the packet loss and route timeout events) may provide 79.55% reduction in time overhead and 52.25% reduction in space overhead.

The counterexample, shown in Figure 5, can be explained as follows. State 1 is the initial state. In state 2, $n0$ initiates a route request to the destination $n2$ by broadcasting a RREQ packet. Similarly, in state 3, $n1$ initiates a route request to the destination $n2$ by broadcasting a RREQ packet. In state 4, $n1$ receives the RREQ packet sent by $n0$ and since neither does it have a route to the destination nor is it the destination itself, it rebroadcasts the RREQ packet. In state 5, $n2$ receives the RREQ packet sent by $n1$ in state 3 and since it is the destination itself, it responds by unicasting a RREP packet after incrementing $seqno_2$. In state 6, $n2$ receives the RREQ packet sent by $n0$ in state 2 and since it is the destination itself, it responds by unicasting a RREP packet after incrementing $seqno_2$. In state 7, $n1$ receives the RREP that is destined to $n0$ and forwards it to $n0$. In addition, $n1$ sets up a forward pointer to the node from which the RREP came (i.e., $n2$), thus establishing a valid routing table entry to the destination $n2$ (note the change of the hop count field from $\infty$ in state 6 to 1 in state 7 and the change of the sequence number field from 0 in state 6 to 6 in state 7).

```
modelCheck: FOUND ERROR
        Depth = 9
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=2 seqno=6 next=1
        Node 1 seqno=2; bid=1; RTEntry: dst=2 hops=1 seqno=4 next=2
        Node 2 seqno=6; bid=1; BcastID: src=1, id=1, BcastID: src=0, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0)

COUNTEREXAMPLE
State 1 Depth = 0
        Node 0 seqno=2; bid=1
        Node 1 seqno=2; bid=1
        Node 2 seqno=2; bid=1
        Network Empty
State 2 Depth = 1
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=2; bid=1
        Node 2 seqno=2; bid=1
        Network RREQ(src:0--dest:1)
State 3 Depth = 2
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 2 seqno=2; bid=1
        Network RREQ(src:0--dest:1), RREQ(src:1--dest:0), RREQ(src:1--dest:2)
State 4 Depth = 3
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1; RTEntry: dst=2 hops=∞ seqno=0 next=0, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=2; bid=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:2), RREQ(src:1--dest:0), RREQ(src:1--dest:2)
State 5 Depth = 4
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1; RTEntry: dst=2 hops=∞ seqno=0 next=0, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; BcastID: src=1, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREQ(src:1--dest:2), RREP(src:2--dest:1--seqno:4)
State 6 Depth = 5
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1; RTEntry: dst=2 hops=∞ seqno=0 next=0, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=6; bid=1; BcastID: src=1, id=1, BcastID: src=0, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREP(src:2--dest:1--seqno:4), RREP(src:2--dest:0--seqno:6)
State 7 Depth = 6
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1; RTEntry: dst=2 hops=1 seqno=6 next=2, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=6; bid=1; BcastID: src=1, id=1, BcastID: src=0, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREP(src:2--dest:1--seqno:4), RREP(src:1--dest:0--seqno:6)
State 8 Depth = 7
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=2 seqno=6 next=1
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1; RTEntry: dst=2 hops=1 seqno=6 next=2, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=6; bid=1; BcastID: src=1, id=1, BcastID: src=0, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREP(src:2--dest:1--seqno:4)
State 9 Depth = 8
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=2 seqno=6 next=1
        Node 1 seqno=2; bid=1
        Node 2 seqno=6; bid=1; BcastID: src=1, id=1, BcastID: src=0, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREP(src:2--dest:1--seqno:4)
State 10 Depth = 9
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=2 seqno=6 next=1
        Node 1 seqno=2; bid=1; RTEntry: dst=2 hops=1 seqno=4 next=2
        Node 2 seqno=6; bid=1; BcastID: src=1, id=1, BcastID: src=0, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0)
Total Time = 3947493.0 msecs.
```

Fig. 5. Trace of counterexample 1 obtained using breadth-first search.

In state 8, $n0$ receives the RREP packet and establishes a valid routing table entry to the destination $n2$. In state 9, the AODV process in $n1$ restarts and in state 10, $n1$ receives the RREP packet that was sent by $n2$ in state 5 in which $seqno_2$ was set to 4. $n1$ establishes a valid routing table entry to the destination $n2$. Nevertheless, as shown in Figure 5, $nexthop_{0,2} = 1$ but $seqno_{0,2} > seqno_{1,2}$; i.e., a routing loop is created.

It should be noted, however, that if the restart of the AODV process at $n1$ in state 9 was because of a node reboot, the link layer of $n0$ may be able to detect that the link between $n0$ and $n1$ was broken causing the invocation of the broken link event handler (Section IV-A), and preventing the routing loop from taking place. On the other hand, if the link layer does not detect broken links fast enough (e.g., if the link layer issues a link layer notification of a broken link only if it has data packets that need to be transmitted to an active next hop, then a node may reboot while its neighbor does not detect it because the rate of sending data packets may be too low), then the routing loop may occur.

As for the network layer mechanism of detecting broken links, according to AODV Draft (version 11) [14], if a node does not receive any packets (HELLO or otherwise) from one of its neighbors for more than $ALLOWED\_HELLO\_LOSS \times HELLO\_INTERVAL$, it should assume that the link to this neighbor is currently lost and invalidate the routing table entries to the destinations that became unreachable because of this broken link and broadcast a RERR packet. However, we discover that the AODV implementation in J-Sim does not

TABLE I

TIME OVERHEAD OF FINDING COUNTEREXAMPLE 1: TIME NEEDED TO LOCATE THE ERROR (IN SEC.).

| | Breadth-first Search | Depth-first Search | Best-first Search 1 | Best-first Search 2 |
|---|---|---|---|---|
| Considering all state transitions | 3947.493 | 5513.528 | 1.656 | 234.374 |
| Ignoring packet loss and route timeout events | 807.185 | 574.555 | 0.157 | 53.540 |

TABLE II

SPACE OVERHEAD OF FINDING COUNTEREXAMPLE 1: NUMBER OF STATES IN THE STATEFUL SEARCH.

| | Breadth-first Search | Depth-first Search | Best-first Search 1 | Best-first Search 2 |
|---|---|---|---|---|
| Considering all state transitions | 26013 | 4758 | 384 | 1792 |
| Ignoring packet loss and route timeout events | 12421 | 2536 | 102 | 2304 |

follow this part of specification. Specifically, in the AODV implementation in J-Sim, if a node does not receive any packets (HELLO or otherwise) from one of its neighbors for more than $1.5 \times ALLOWED\_HELLO\_LOSS \times HELLO\_INTERVAL$, it deletes the neighbor's information from the neighbors list mentioned in Section IV-A, *without* invalidating the routing table entries to the destinations that became unreachable because of the broken link or broadcasting a RERR packet. Furthermore, as stated in AODV Draft (version 11) [14], after a node reboot, a node waits for $DELETE\_PERIOD$. During this time, the node does not transmit any RREP packets, and if it receives a data packet for some other destination, it should broadcast a RERR packet and must reset the waiting timer to expire after the current time plus $DELETE\_PERIOD$. We found that the AODV implementation in J-Sim does not follow this part of specification either. It was shown in [23] that by the time the rebooted node comes out of the waiting phase and becomes an active router again, none of its neighbors will be using it as an active next hop any more.

Table I gives the time needed to locate the error in all the three search strategies, while Table II shows the space overhead in terms of the number of states in the stateful search. Best-first Search 1 can reduce both the time and space overhead by several orders of magnitude. However, this comes at the price of a (possibly) longer counterexample, which was 16 states in the best-first search compared to 10 states in the breadth-first search. Nevertheless, the choice of the best-first search strategy affects both the time and space overheads needed to locate the error. As shown in Tables I- II, Best-first Search 2 incurs a two order of magnitude increase in time overhead and a one order of magnitude increase in space overhead, when compared to Best-first Search 1. The reason why Best-first Search 2 performed worse than Best-first Search 1 is that Best-first Search 2 requires a node (and its next hop towards the destination) to have valid routing table entries to the destination. This may not be true at the first stages (i.e., lower depths) of the search space. Therefore, in the first stages of the search, the nonvisited states may look equally interesting and thus, Best-first Search 2 may not be able to explore the states that are most likely to lead to the error first. This causes Best-first Search 2 to take more time and explore more states than Best-first Search 1. We leave the question of how to determine the "best" best-first search strategy to future research work.

To demonstrate the powerfulness of BMC and to further study the performance of the various search strategies, we manually injected the following two errors, in addition to the error found above.

**Counterexample 2** - The first error that we manually injected is not to increment the destination sequence number when invalidating a routing table entry. In order not to get the same counterexample shown in Figure 5, we required that the counterexample should contain at least one state that is generated due to a route timeout event because the route timeout event triggers the invalidation of a routing table entry. Using a breadth-first search strategy with all the state transitions mentioned in Section IV-B, the model checker found an error in 1.5792 hour using space overhead of 29083 states. Using a depth-first search strategy, the model checker finds the error, whose trace is shown in Figure 6, using space overhead of only 9068 states but at the price of taking 13.7788 hours. Furthermore, the length of the counterexample found using the depth-first search strategy was 16 states, but the length of the counterexample found using the breadth-first search strategy was only 10 states.

The counterexample, shown in Figure 6, can be explained as follows. State 1 is the initial state. In state 2, $n1$ initiates a route request to the destination $n2$ by broadcasting a RREQ packet. In state 3, the RREQ packet that was destined to $n2$ is lost. In state 4, the AODV process at $n1$ restarts. In state 5, $n1$ initiates a route request to the destination $n2$ by broadcasting a RREQ packet. In state 6, the RREQ packet that was destined to $n2$ is lost. In state 7, the AODV process at $n1$ restarts. In state 8, $n0$ initiates a route request to the destination $n2$ by broadcasting a RREQ packet. In state 9, $n1$ receives the RREQ packet sent by $n0$ and since neither does it have a route to the destination nor is it the destination itself, it rebroadcasts the RREQ packet. In state 10, $n2$ receives the RREQ packet sent by $n0$ in state 8 and since it is the destination itself, it responds by unicasting a RREP packet after incrementing $seqno_2$. In state 11, $n1$ receives the RREP that is destined to $n0$ and forwards it to $n0$. In addition, $n1$ sets up a forward pointer to the node from which the RREP came (i.e., $n2$), thus establishing a valid routing table entry to the destination $n2$. In state 12, the route timeout event takes place at $n1$ causing the invalidation of the routing table entry to $n2$ (note the change of the hop count field from 1 in state 11 to $\infty$ in state 12) without incrementing the destination sequence number. In state 13, the AODV process at $n0$ restarts. In state 14, $n0$ receives the RREP packet and establishes a valid routing table entry to the destination $n2$. In state 15, $n0$ receives a RREQ packet sent by $n1$ and since it has a fresh enough route to the destination $n2$, it responds by unicasting a RREP packet. In state 16, $n1$ receives the RREP packet and since the destination sequence number field in the RREP packet is equal to the destination sequence number field in the routing table entry to $n2$ and the hop count field in the RREP packet is less than the hop count field in the routing table

TABLE III

TIME OVERHEAD OF FINDING COUNTEREXAMPLE 2: TIME NEEDED TO LOCATE THE ERROR (IN SEC.).

| | Breadth-first Search | Depth-first Search | Best-first Search 1 | Best-first Search 2 |
|---|---|---|---|---|
| Considering all state transitions | 5685.178 | 49603.651 | 0.132 | 232.978 |

TABLE IV

SPACE OVERHEAD OF FINDING COUNTEREXAMPLE 2: NUMBER OF STATES IN THE STATEFUL SEARCH.

| | Breadth-first Search | Depth-first Search | Best-first Search 1 | Best-first Search 2 |
|---|---|---|---|---|
| Considering all state transitions | 29083 | 9068 | 93 | 1786 |

entry to $n2$, $n1$ accepts the new offered route and establishes a valid routing table entry to the destination $n2$. Nevertheless, as shown in Figure 6, $nexthop_{0,2} = 1$ and $nexthop_{1,2} = 0$; i.e., a routing loop is created.

Tables III and IV give, respectively, the time and space overheads needed to locate the error using the three search strategies. Similar to the results presented above, a best-first search strategy can provide several orders of magnitude reduction in both time and space overheads.

**Counterexample 3** - The second error that we manually injected is to delete (instead of invalidating) a routing table entry when its lifetime expires. In order not to get the same counterexample shown in Figure 5, we required that the counterexample should contain at least one state that is generated due to a route timeout event because the route timeout event triggers the deletion of a routing table entry. Using a breadth-first search strategy with all the state transitions mentioned in Section IV-B, the model checker found an error in 59.31 minutes using space overhead of 24966 states. Using Best-first Search 1, the model checker finds the error, whose trace is shown in Figure 7, in only 0.137 second using space overhead of only 93 states but at the price of finding a longer counterexample – the length of the counterexample found using the best-first search strategy was 16 states, but the length of the counterexample found using the breadth-first search strategy was only 10 states.

States 1-11 of the counterexample can be explained as explained in the previous errors. In state 12, both $n0$ and $n1$ have valid routing table entries to $n2$. In state 13, the route timeout event takes place at $n1$ causing the deletion of the routing table entry to $n2$. In state 14, $n1$ initiates a route request to the destination $n2$ by broadcasting a RREQ packet. In state 15, $n0$ receives the RREQ packet sent by $n1$ and since it has a fresh enough route to the destination $n2$, it responds by unicasting a RREP packet. In state 16, $n1$ receives the RREP packet and since the destination sequence number field in the RREP packet is greater than the destination sequence number field in the routing table entry to $n2$, $n1$ accepts the new offered route and establishes a valid routing table entry to the destination $n2$. Nevertheless, as shown in Figure 7, $nexthop_{0,2} = 1$ and $nexthop_{1,2} = 0$; i.e., a routing loop is created.

Tables V and VI give, respectively, the time and space overheads needed to locate the error using the three search strategies. Similar to the results presented above, a best-first search strategy can reduce both the time and space overhead by several orders of magnitude.

TABLE V

TIME OVERHEAD OF FINDING COUNTEREXAMPLE 3: TIME NEEDED TO LOCATE THE ERROR (IN SEC.).

| | Breadth-first Search | Depth-first Search | Best-first Search 1 | Best-first Search 2 |
|---|---|---|---|---|
| Considering all state transitions | 3558.614 | 5046.795 | 0.137 | 231.961 |

TABLE VI

SPACE OVERHEAD OF FINDING COUNTEREXAMPLE 3: NUMBER OF STATES IN THE STATEFUL SEARCH.

| | Breadth-first Search | Depth-first Search | Best-first Search 1 | Best-first Search 2 |
|---|---|---|---|---|
| Considering all state transitions | 24966 | 4637 | 93 | 1783 |

In addition to evaluating the ability of the J-Sim model checker to find a counterexample of a safety property, we have studied its ability to find a witness for the liveness property specified in Section IV-B. Tables VII and VIII give, respectively, the time and space overheads needed to find the witness using the three search strategies. A best-first search strategy can reduce both the time and space overhead, needed to find a witness for the liveness property, by several orders of magnitude.

*Effect of the size and/or the topology of the network:* In this subsection, we study the effect of the size and/or the topology of the network on the performance of the model checker with respect to the time and space overheads required for finding a counterexample and a witness. To study the effect of the size of the network, Table IX gives the time and space overheads needed to find Counterexample 3 in a chain topology consisting of $N$ nodes using Best-first Search 1. To study the effect of the topology and the size of the network, Table X gives the time and space overheads needed to find a witness in a grid topology consisting of $M \times M$ nodes using Best-first Search 1.

## V. RELATED WORK

Conventional model checkers (e.g., SPIN [24], SMV [25], Murphi [26]) require that the system be first specified using a high-level modeling language. The process of describing the system in a high-level modeling language is a time-consuming, painstaking, and error-prone process. To deal with this, there has been some work (e.g., [27], Java PathFinder [28], and Bandera [29]) on translating programming languages (e.g., Java) into the input modeling languages of several conventional model checkers. The idea is to automatically extract an abstract model out of an application written in Java and then use conventional model checking to analyze this abstract model. However, this may not be always feasible, as it requires that each language feature of Java must have a corresponding one in the destination modeling language.

TABLE VII

TIME OVERHEAD OF FINDING A WITNESS FOR THE LIVENESS PROPERTY SPECIFIED IN SECTION IV-B: TIME NEEDED TO LOCATE THE WITNESS (IN SEC.).

| | Breadth-first Search | Depth-first Search | Best-first Search 1 | Best-first Search 2 |
|---|---|---|---|---|
| Considering all state transitions | 0.61 | 4.047 | 0.098 | 1.26 |

TABLE VIII

SPACE OVERHEAD OF FINDING A WITNESS FOR THE LIVENESS PROPERTY SPECIFIED IN SECTION IV-B: NUMBER OF STATES IN THE STATEFUL SEARCH.

| | Breadth-first Search | Depth-first Search | Best-first Search 1 | Best-first Search 2 |
|---|---|---|---|---|
| Considering all state transitions | 552 | 240 | 58 | 154 |

TABLE IX

TIME (SEC.) AND SPACE (NUMBER OF STATES IN THE STATEFUL SEARCH) OVERHEADS OF FINDING COUNTEREXAMPLE 3 IN A CHAIN TOPOLOGY CONSISTING OF $N$ NODES USING BEST-FIRST SEARCH 1.

| N | Time (sec.) | Number of states |
|---|---|---|
| 3 | 0.137 | 93 |
| 4 | 3.946 | 575 |
| 5 | 427.404 | 3817 |
| 6 | 582.150 | 3013 |
| 7 | 1803.807 | 3846 |

TABLE X

TIME (SEC.) AND SPACE (NUMBER OF STATES IN THE STATEFUL SEARCH) OVERHEADS OF FINDING A WITNESS IN A GRID TOPOLOGY CONSISTING OF $M \times M$ NODES USING BEST-FIRST SEARCH 1.

| $M \times M$ | Time (sec.) | Number of states |
|---|---|---|
| $2 \times 2$ | 0.334 | 140 |
| $3 \times 3$ | 12.375 | 2014 |
| $4 \times 4$ | 209.196 | 10112 |

Conventional model checkers have also been used in formal reasoning of distributed systems. In [30], SMV is used to verify three cache coherence protocols used in distributed file systems; however, each cache coherence protocol has to be modeled using the SMV input language and then SMV checks that model rather than the actual implementation. In [31], the process of writing the model is automated by using an extensible compiler, xg++, that can automatically extract a model (described in the Murphi input language) from the original implementation code. Compared to [31], our goal is not to extract a model, but instead to model check the J-Sim implementation code itself. Teapot [32] is a domain-specific language for writing cache coherence protocols, and offers further improvement over [30] and [31]. The Teapot compiler can translate a protocol specification to both executable C code and code that can be input to Murphi, and hence potential discrepancies between the specification and the actual executable code can be eliminated. Our approach differs from Teapot in that it does not require an input protocol specification and does not generate an output executable code. Furthermore, since the model checker is built in J-Sim, there is no need to use an existing model checker such as Murphi.

As mentioned above, the J-Sim model checker checks a network protocol by executing the J-Sim implementation code of that network protocol *directly* and exploring the state space *on the fly*. This is inspired by the previous work on model checking the implementation code directly (e.g., CMC [22] and Verisoft [33]) for C and C++. Although CMC has been applied to model check implementations of networking code (namely, the AODV routing protocol [14]), our approach differs from CMC in two aspects: (a) our goal is to model check the network protocol while it is being designed (using J-Sim) rather than after it is implemented; and (b) we focus on Java rather than C or C++. Several errors found using CMC were caused by implementation errors (e.g., using the result of a routing table lookup without checking for NULL or using invalid routing table entries to send routing updates); such errors do not exist in the implementation of AODV in J-Sim.

A more recent version of Java PathFinder [34] performs model checking at the bytecode level. However, this involves building a new Java Virtual Machine $JVM^{JPF}$, which is called from the model checker, to interpret bytecode generated by a Java compiler. Compared to [34], our approach has an important advantage of not requiring any modifications to the Java Virtual Machine. As far as formal analysis of network simulation is concerned, the only existing work is Verisim [35], which was developed based on a collection of pre-existing tools, i.e., ns-2 [5] and the MaC monitoring and checking framework [36]. Verisim replaces the monitor component of MaC by ns-2 and uses the checker component of MaC to verify user-defined properties on traces produced by ns-2. It should be noted, however, that not all errors may manifest themselves in a trace because ns-2 does not explore all possible execution paths during a simulation run.

Maude [37] is a reflective language and system that supports both equational and rewriting logic specification and programming. Maude is extremely powerful and can be used to create *executable* specifications for a wide range of applications (e.g., other languages, theorem provers, concurrent systems). In fact, Maude can even be used to build language extensions for Maude itself. For example, Full Maude is implemented in Maude as an extension of (Core) Maude [38]. Concurrent object-oriented systems can be specified in Full Maude by means of object-oriented modules that can be executed and also model-checked with a Linear Temporal Logic (LTL) model checker. In [39], we have compared the performance of the J-Sim model checker against that of Maude, and shown that the former outperforms the latter in model checking a stop-and-wait ARQ protocol.

Bhargavan *et al.* provide in [21] a complete automated proof, using the SPIN model checker and the HOL theorem prover, that no routing loops will be formed by AODV if all the nodes in the ad hoc network (a) always immediately detect when a neighbor restarts its AODV process and the restart is treated as if all links to the neighbor have broken, (b) increment the destination sequence number of a routing table entry to a destination when the route to that destination has expired or broken, and (c) never delete routing table entries. In this paper, we have shown how the model checker in J-Sim can

discover routing loops caused by violation of each of these three conditions.

## VI. Conclusions and Future Work

This paper documents our ongoing research on extending J-Sim, an open-source component-based compositional network simulation and emulation environment, with the capability of validating network protocols using dynamic on the fly bounded model checking. We demonstrate the ability of the model checker, that we have built and incorporated into J-Sim, to model-check the Ad-Hoc On-Demand Distance Vector (AODV) routing protocol, a fairly complex network protocol with thousands of lines of Java code. Experimental results have shown that the model checker in J-Sim is able to find counterexamples of a safety property and witnesses for a liveness property within acceptable time. Furthermore, protocol-specific properties and heuristics that are inherent to network protocols expedite the process of locating these counterexamples and witnesses. Use of a best-first search strategy (that leverages protocol specific metrics) also reduces the time and space overheads by several orders of magnitude.

We believe we have made a case for incorporating model checking into simulation, so as to provide an integrated performance evaluation and validation environment. We have identified several research avenues for future work. First, we are currently extending the J-Sim model checker to check general LTL formulae. We will also enable the model checker to handle timeout events depend on actual timer values. Second, we will study whether or not and how the models/classes in J-Sim can be formally analyzed using JavaFAN (the Java Formal Analyzer) [40], [41], a recent tool that uses executable semantic definitions of Java and the JVM written in Maude in order to symbolically simulate, search the state space of, and model check Java concurrent programs at both the source-code and byte-code levels.

## References


[1] F. Calí, M. Conti, and E. Gregori, "Dynamic tuning of the IEEE 802.11 protocol to achieve a theoretical throughput limit," *IEEE/ACM Trans. on Networking*, vol. 8, no. 6, pp. 785–799, December 2000.

[2] V. Misra, W. Gong, and D. Towsley, "Stochastic differential equation modeling and analysis of TCP windowsize behavior," in *Proc. of Performance'99*, October 1999.

[3] J. Padhye, V. Firoiu, D. Towsley, and J. Kurose, "Modeling TCP throughput: A simple model and its empirical validation," in *Proc. of ACM SIGCOMM'98*, September 1998.

[4] S. Shakkottai and R. Srikant, "How good are deterministic fluid models of internet congestion control?," in *Proc. of IEEE INFOCOM'02*, June 2002.

[5] Ns-2, "http://www.isi.edu/nsnam/ns/," .

[6] J-Sim, "http://www.j-sim.org/," .

[7] H.-Y. Tyan and J. C. Hou, "*JavaSim*: A component-based compositional network simulation environment," in *Proc. of Western Simulation Multiconference, CNDS'01*, January 2001.

[8] H.-Y. Tyan, *Design, Realization and Evaluation of a Component-based Compositional Software Architecture for Network Simulation*, Ph.D. thesis, Department of Electrical Engineering, The Ohio State University, 2002.

[9] S. Bajaj, L. Breslau, D. Estrin, K. Fall, S. Floyd, P. Haldar, M. Handley, A. Helmy, J. Heidemann, P. Huang, S. Kumar, S. McCanne, R. Rejaie, P. Sharma, K. Varadhan, Y. Xu, H. Yu, and D. Zappala, "Improving simulation for network research," Tech. Rep. 99-702, University of Southern California, 1999.

[10] D. J. Schwartz, Y. Yemini, and D. Bacon, "NEST: A network simulation and prototyping testbed," *Communications of the ACM*, vol. 33, no. 10, pp. 63–74, October 1990.

[11] S. Keshav, "REAL: A network simulator," Tech. Rep. 88/472, University of California, Berkeley, 1988.

[12] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu, "Symbolic model checking without BDDs," in *Proc. of TACAS'99*, March 1999.

[13] Tobias Schuele and Klaus Schneider, "Bounded model checking of infinite state systems: Exploiting the automata hierarchy," in *Proc. of ACM-IEEE MEMOCODE'04*, June 2004.

[14] C. E. Perkins, E. M. Royer, and S. Das, "Ad hoc on demand distance vector (aodv) routing," IETF Draft, January 2002.

[15] E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 1999.

[16] B. Berard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, and P. McKenzie, *Systems and Software Verification: Model-Checking Techniques and Tools*, Springer Verlag, 2001.

[17] Zohar Manna and Amir Pnueli, *The Temporal Logic of Reactive and Concurrent Systems - Specification*, Springer-Verlag, 1992.

[18] Edmund M. Clarke, Daniel Kroening, Joel Ouaknine, and Ofer Strichman, "Completeness and complexity of bounded model checking," in *Proc. of VMCAI'04*, January 2004.

[19] D. Flanagan, *Java In A Nutshell*, O'Reilly, 1997.

[20] C. E. Perkins and E. M. Royer, "Ad-hoc on-demand distance vector routing," in *Proc. of IEEE WMCSA'99*.

[21] Karthikeyan Bhargavan, Davor Obradovic, and Carl A. Gunter, "Formal verification of standards for distance vector routing protocols," *Journal of the ACM*, vol. 49, no. 4, pp. 538–576, July 2002.

[22] M. Musuvathi, D. Y.W. Park, A. Chou, D. R. Engler, and D. L. Dill, "CMC: A pragmatic approach to model checking real code," in *Proc. of OSDI'02*.

[23] Karthikeyan Bhargavan, Carl A. Gunter, and Davor Obradovic, "Fault origin adjudication," in *Proc. of the Workshop on Formal Methods in Software Practice*, August 2000.

[24] G. J. Holzmann, "The model checker SPIN," *IEEE Trans. on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997.

[25] K. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.

[26] D. L. Dill, A. J. Drexler, A. J. Hu, and C. H. Yang, "Protocol verification as a hardware design aid," in *Proc. of IEEE ICCD'92*.

[27] D. Y.W. Park, U. Stern, J. U. Skakkebæk, and D. L. Dill, "Java model checking," in *Proc. of IEEE ASE'00*.

[28] K. Havelund, "Java Pathfinder, a translator from Java to Promela," in *Proc. of SPIN'99*.

[29] J. Corbett, M. Dwyer, J. Hatcliff, C. Păsăreanu, Robby, S. Laubach, and H. Zheng, "Bandera: Extracting finite state models from Java source code," in *Proc. of ICSE'00*, June 2000.

[30] J. M. Wing and M. Vaziri-Farahani, "Model checking software systems: a case study," in *Proc. of ACM SIGSOFT'95*, October 1995.

[31] D. Lie, A. Chou, D. Engler, and D. Dill, "A simple method for extracting models from protocol code," in *Proc. of ISCA'01*.

[32] S. Chandra, B. Richards, and J. R. Larus, "Teapot: a domain-specific language for writing cache coherence protocols," *IEEE Trans. on Software Engineering*, vol. 25, no. 3, pp. 317–333, May-June 1999.

[33] P. Godefroid, "Model checking for programming languages using VeriSoft," in *Proc. of ACM POPL'97*.

[34] W. Visser, K. Havelund, G. Brat, and S.Park, "Model checking programs," in *Proc. of IEEE ASE'00*.

[35] K. Bhargavan, C. A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan, "Verisim: formal analysis of network simulations," *IEEE Trans. on Software Engineering*, vol. 28, no. 2, pp. 129–145, February 2002.

[36] M. Kim, M. Viswanathan, H. Ben-Abdallah, S. Kannan, I. Lee, and O. Sokolsky, "Formally specified monitoring of temporal properties," in *Proc. of ECRTS'99*.

[37] Maude, "http://maude.cs.uiuc.edu/," .

[38] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *Maude 2.0 Manual*, July 2003.

[39] Ahmed Sobeih, Mahesh Viswanathan, and Jennifer C. Hou, "Check and Simulate: A case for incorporating model checking in network simulation," in *Proc. of ACM-IEEE MEMOCODE'04*, June 2004.

[40] Azadeh Farzan, Feng Chen, Jose Meseguer, and Grigore Rosu, "Formal analysis of Java programs in JavaFAN," in *Proc. of CAV'04*, July 2004.

[41] Azadeh Farzan, Jose Meseguer, and Grigor Rosu, "Formal JVM code analysis in JavaFAN," in *Proc. of AMAST'04*, July 2004.

[42] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The Maude LTL model checker," in *Proc. of WRLA'02*.

```
modelCheck: FOUND ERROR
        Depth = 15
        Node 0 seqno=2; bid=1; BcastID: src=1, id=1; RTEntry: dst=2 hops=2 seqno=4 next=1, RTEntry: dst=1 hops=1 seqno=4 next=1
        Node 1 seqno=2; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=1 seqno=4 next=0, RTEntry: dst=2 hops=3 seqno=4 next=0
        Node 2 seqno=4; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0)

ERROR TRACE
State 1 Depth = 0
        Node 0 seqno=2; bid=1
        Node 1 seqno=2; bid=1
        Node 2 seqno=2; bid=1
        Network Empty
State 2 Depth = 1
        Node 0 seqno=2; bid=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 2 seqno=2; bid=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:2)
State 3 Depth = 2
        Node 0 seqno=2; bid=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 2 seqno=2; bid=1
        Network RREQ(src:1--dest:0)
State 4 Depth = 3
        Node 0 seqno=2; bid=1
        Node 1 seqno=2; bid=1
        Node 2 seqno=2; bid=1
        Network RREQ(src:1--dest:0)
State 5 Depth = 4
        Node 0 seqno=2; bid=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 2 seqno=2; bid=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREQ(src:1--dest:2)
State 6 Depth = 5
        Node 0 seqno=2; bid=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 2 seqno=2; bid=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0)
State 7 Depth = 6
        Node 0 seqno=2; bid=1
        Node 1 seqno=2; bid=1
        Node 2 seqno=2; bid=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0)
State 8 Depth = 7
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=2; bid=1
        Node 2 seqno=2; bid=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREQ(src:0--dest:1)
State 9 Depth = 8
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=2; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=2; bid=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREQ(src:1--dest:2)
State 10 Depth = 9
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=2; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREP(src:2--dest:0--seqno:4)
State 11 Depth = 10
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=2; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=1 seqno=4 next=0, RTEntry: dst=2 hops=1 seqno=4 next=2
        Node 2 seqno=4; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREP(src:1--dest:0--seqno:4)
State 12 Depth = 11
        Node 0 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=2; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=1 seqno=4 next=0, RTEntry: dst=2 hops=∞ seqno=4 next=0
        Node 2 seqno=4; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREP(src:1--dest:0--seqno:4)
State 13 Depth = 12
        Node 0 seqno=2; bid=1
        Node 1 seqno=2; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=1 seqno=4 next=0, RTEntry: dst=2 hops=∞ seqno=4 next=0
        Node 2 seqno=4; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREP(src:1--dest:0--seqno:4)
State 14 Depth = 13
        Node 0 seqno=2; bid=1; RTEntry: dst=2 hops=2 seqno=4 next=1
        Node 1 seqno=2; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=1 seqno=4 next=0, RTEntry: dst=2 hops=∞ seqno=4 next=0
        Node 2 seqno=4; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREQ(src:1--dest:0)
State 15 Depth = 14
        Node 0 seqno=2; bid=1; BcastID: src=1, id=1; RTEntry: dst=2 hops=2 seqno=4 next=1, RTEntry: dst=1 hops=1 seqno=4 next=1
        Node 1 seqno=2; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=1 seqno=4 next=0, RTEntry: dst=2 hops=∞ seqno=4 next=0
        Node 2 seqno=4; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0), RREP(src:0--dest:1--seqno:4)
State 16 Depth = 15
        Node 0 seqno=2; bid=1; BcastID: src=1, id=1; RTEntry: dst=2 hops=2 seqno=4 next=1, RTEntry: dst=1 hops=1 seqno=4 next=1
        Node 1 seqno=2; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=1 seqno=4 next=0, RTEntry: dst=2 hops=3 seqno=4 next=0
        Node 2 seqno=4; bid=1; BcastID: src=0, id=1; RTEntry: dst=0 hops=2 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:0)
Total Time = 4.9603651E7 msecs.
```

Fig. 6. Trace of counterexample 2 obtained using depth-first search.

```
modelCheck: FOUND ERROR
        Depth = 15
        Node 0 seqno=4; bid=2; BcastID: src=1, id=2; RTEntry: dst=1 hops=1 seqno=6 next=1, RTEntry: dst=2 hops=2 seqno=4 next=1
        Node 1 seqno=6; bid=3; RTEntry: dst=0 hops=1 seqno=4 next=0, RTEntry: dst=2 hops=3 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:2)

ERROR TRACE
State 1 Depth = 0
        Node 0 seqno=2; bid=1
        Node 1 seqno=2; bid=1
        Node 2 seqno=2; bid=1
        Network Empty
State 2 Depth = 1
        Node 0 seqno=2; bid=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 2 seqno=2; bid=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:2)
State 3 Depth = 2
        Node 0 seqno=2; bid=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 2 seqno=4; bid=1; BcastID: src=1, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREP(src:2--dest:1--seqno:4)
State 4 Depth = 3
        Node 0 seqno=2; bid=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=1 seqno=4 next=2
        Node 2 seqno=4; bid=1; BcastID: src=1, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0)
State 5 Depth = 4
        Node 0 seqno=2; bid=1; BcastID: src=1, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=1 seqno=4 next=2
        Node 2 seqno=4; bid=1; BcastID: src=1, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:0--dest:1)
State 6 Depth = 5
        Node 0 seqno=2; bid=1; BcastID: src=1, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=1 seqno=4 next=2
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:0--dest:1)
State 7 Depth = 6
        Node 0 seqno=2; bid=1; BcastID: src=1, id=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=1 seqno=4 next=2
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network Empty
State 8 Depth = 7
        Node 0 seqno=2; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=1 seqno=4 next=2
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network Empty
State 9 Depth = 8
        Node 0 seqno=4; bid=2; RTEntry: dst=1 hops=1 seqno=4 next=1; RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=1 seqno=4 next=2
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:0--dest:1)
State 10 Depth = 9
        Node 0 seqno=4; bid=2; RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1; RTEntry: dst=2 hops=1 seqno=4 next=2, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREP(src:1--dest:0--seqno:4)
State 11 Depth = 10
        Node 0 seqno=4; bid=2; RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=2 hops=2 seqno=4 next=1
        Node 1 seqno=4; bid=2; BcastID: src=0, id=1; RTEntry: dst=2 hops=1 seqno=4 next=2, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network Empty
State 12 Depth = 11
        Node 0 seqno=4; bid=2; RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=2 hops=2 seqno=4 next=1
        Node 1 seqno=4; bid=2; RTEntry: dst=2 hops=1 seqno=4 next=2, RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network Empty
State 13 Depth = 12
        Node 0 seqno=4; bid=2; RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=2 hops=2 seqno=4 next=1
        Node 1 seqno=4; bid=2; RTEntry: dst=0 hops=1 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network Empty
State 14 Depth = 13
        Node 0 seqno=4; bid=2; RTEntry: dst=1 hops=1 seqno=4 next=1, RTEntry: dst=2 hops=2 seqno=4 next=1
        Node 1 seqno=6; bid=3; RTEntry: dst=0 hops=1 seqno=4 next=0, RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:0), RREQ(src:1--dest:2)
State 15 Depth = 14
        Node 0 seqno=4; bid=2; BcastID: src=1, id=2; RTEntry: dst=1 hops=1 seqno=6 next=1, RTEntry: dst=2 hops=2 seqno=4 next=1
        Node 1 seqno=6; bid=3; RTEntry: dst=0 hops=1 seqno=4 next=0, RTEntry: dst=2 hops=∞ seqno=0 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:2), RREP(src:0--dest:1--seqno:4)
State 16 Depth = 15
        Node 0 seqno=4; bid=2; BcastID: src=1, id=2; RTEntry: dst=1 hops=1 seqno=6 next=1, RTEntry: dst=2 hops=2 seqno=4 next=1
        Node 1 seqno=6; bid=3; RTEntry: dst=0 hops=1 seqno=4 next=0, RTEntry: dst=2 hops=3 seqno=4 next=0
        Node 2 seqno=4; bid=1; RTEntry: dst=1 hops=1 seqno=4 next=1
        Network RREQ(src:1--dest:2)
Total Time = 137.0 msecs.
```

Fig. 7. Trace of counterexample 3 obtained using Best-first Search 1.