J-Sim: An Integrated Environment for Simulation and Model Checking of Network Protocols

Ahmed Sobeih, Mahesh Viswanathan, Darko Marinov, and Jennifer C. Hou

University of Illinois at Urbana-Champaign
Department of Computer Science
Urbana, IL 61801 USA
{sobeih, vmahesh, marinov, jhou}@cs.uiuc.edu

Abstract

In this paper, we report our work [24, 26] on extending the J-Sim network simulator [13] to be an integrated environment for both simulation and model checking of network protocols. We also present a case study in which we model-checked AODV in J-Sim.

1. Introduction

One major deficiency of traditional network simulators is that they only evaluate the performance of network protocols in scenarios provided by the protocol designer but can *not* exhaustively analyze possible scenarios for correctness. For example, a network simulator can evaluate the performance of a routing protocol but cannot check whether this protocol may suffer from routing loops. In the current practice, to check whether or not a network protocol contains any errors, a prototype that was solely written for verification purposes has to be built (e.g., in an interactive theorem prover and/or a model checker). In general, 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* (e.g., via simulation) and *verification* (e.g., via model checking) 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* [13, 27, 28]—a composable and extensible open-source network simulation environment that is developed entirely in Java—with the model checking [4] capability to explore the state space created by a network protocol up to a (configurable) maximum depth in order to find violations of a safety property (e.g., the absence of routing loops in a routing protocol). The basic idea is to execute the simulation code in order to pinpoint errors in the network protocol. Specifically, we have implemented a model checker (written in Java so that it can be readily integrated with *J-Sim*), and incorporated this model checker into *J-Sim*.

Design of special-purpose model checkers for network simulator code enjoys several benefits over using general-purpose verification tools. First, it saves the protocol designer the task of building a special-purpose model of the protocol for verification and a separate model for performance analysis. Since building a formal model of a protocol is an onerous, time-consuming and error-prone task, by designing special-purpose model checkers for network simulator code, we ensure that verifying a protocol is easier for the designer (and hence something he/she is more likely to perform). Second, using a model checker for C or Java (like [9, 18, 1, 3, 10, 7]) to verify the protocol code *along* with the simulator code might likely be intractable due to the complexity of the general-purpose simulator code.

To extend *J-Sim* with the model checking capability, we have addressed several issues. First, we have laid a framework that enables the model checker to take control of the simulation of a network protocol in order to explore the (entire) state space, rather than just exploring one single execution path as *J-Sim* traditionally does. Second, we ensured that the implementation of this model checking framework did not require the core design and implementation of *J-Sim* to be altered. Third, we instrumented the model checker to make use of a best-first search (BeFS) strategy, which exploits *properties inherent to the network protocol and the safety property being checked*, in order to guide the search towards paths that can potentially locate errors in less time before running out of memory due to the well-known *state*

1-4244-0910-1/07/$20.00

space explosion problem in model checking.

The rest of the paper is organized as follows. In Section 2, we give an overview of network simulation in *J-Sim*. In Section 3, we give an overview of the model checking framework in *J-Sim*. In Section 4, we present the performance results of one of our case studies. In Section 5, we discuss related work. Finally, we conclude the paper in Section 6.

2. Network Simulation in *J-Sim*

J-Sim is implemented on top of a component-based software architecture, called the *autonomous component architecture (ACA)*, that closely mimics the integrated circuit (IC) design. The basic entities in the ACA are *components*, which communicate with one another via sending/receiving data at their *ports*. 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 software architecture of the ACA is motivated by the belief that software design cannot achieve the same level of modularity as IC design due to the fact that the object-oriented (OO) programming paradigm is fundamentally different from hardware design in *component binding*. Specifically, in OO programming, a class makes direct references to other class instances and makes function calls to those exposed by other class instances. The binding is "strong" in the sense that the caller has to know the exact names of the callees. In the course of debugging, one cannot obtain a clear view of binding relations without delving into the implementation details and tracing codes line by line. This yields unpredictability in software development and high maintenance cost, and is usually termed as software crisis. In contrast, an IC chip is a blackbox fully specified by the function specification and the input/output signal patterns in the databook. Changes in input signals trigger an IC chip to perform certain function, and change, after a certain delay, its outputs according to the chip specification. The fact that an IC chip is interfaced with other chips/modules/systems only through its pins (and is otherwise shielded from the rest of the world) allows IC chips to be designed, implemented, and tested, independently of everything else. In other words, at design time, an IC chip is bound with a certain specification in the databook, instead of being bound to components that interact with it. Component binding is thus deferred to the time when a system (e.g., ALU) is being composed.

Following the same line of design principles, how components in the ACA behave (in terms of how a component handles and responds to data that arrives at a port) is specified at system design time in *contracts*, but component binding does not take place until the system integration time when the system is being "composed." A contract specifies how an initiator (caller) and a reactor (callee) fulfill a certain function. It simply specifies the causality of information exchange between components but *not* the components that may participate in information exchange. Two components, acting respectively as the initiator and the reactor, are bound at system integration time to fulfill the contract. In some sense, the ACA realizes the notion of *software IC* where an IC corresponds to a component, pins correspond to ports, signals correspond to data that arrives at a port of a component, and an IC specification corresponds to a contract.

With the separation of contract binding (at system design time) from component binding (at system integration time), *J-Sim* provides a loosely-coupled component architecture, i.e., a component can be individually designed, implemented and tested independently [27]. By closing the gap between hardware and software ICs, the ACA realizes the objectives of composability and extensibility of network simulation [28]. All of these features enable new components to be included into *J-Sim* in a plug-and-play fashion. On top of the ACA, a generalized packet-switched internetworking framework (called INET) has been laid based on common features extracted from the various layers in the protocol stack. Both the ACA and the INET have been implemented in Java, and the resulting code, along with its scripting framework and GUI interfaces, is called *J-Sim*. Finally, an essential suite of wired and wireless network components and protocols have been implemented in *J-Sim*.

3. Model Checking Framework in *J-Sim*

The model checker, that we implemented as a component in *J-Sim*, is an explicit-state model checker [4] that checks a network protocol by executing the *J-Sim* simulation code of that network protocol *directly* and exploring the state space on-the-fly until either a counterexample disproving a safety property is found or the state space is explored up to a maximum depth (MAX_DEPTH).

In order to explore the state space created by a network protocol, the notion of the "state" has to be adequately defined. To this end, the model checker makes use of the *GlobalState* class. A state is an instance of *GlobalState*. The model checking procedure modelCheck, shown in Figure 1, interacts via ports with three instances of *GlobalState*; namely, *initialState* (the initial state 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 1, 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 1 presents a stateful search that avoids visiting a state if another equivalent state has already been visited before (i.e., a state that al-

ready exists in *AlreadyVisitedStates*). *AlreadyVisitedStates* stores concrete states, and two states s_1 and s_2 are considered equivalent if $s_1.equals(s_2)$ returns true.

In each state in the state space, some transitions (i.e., events) may or may not be enabled, and an enabled transition may generate multiple successor states. For instance, 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. In modelCheck, the enabling function (Figure 1, line 9) returns the number of possible successor states (zero if the event is disabled). For each state being explored (*currentState*), modelCheck generates all the possible successor states (*nextState*) by executing the event handlers of the events that are enabled in *currentState*. However, since an event handler is only invoked from modelCheck but actually executed inside the protocol entities (i.e., the classes that implement the network protocol being model-checked) themselves, modelCheck 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 (line 11) where the model checker interacts via ports with the protocol entities.

After the execution of the event handler (line 14), the *CopyFromEntitiesToModel()* function is called (line 15) to extract the new state information from the protocol entities and copy them to *nextState*. If *nextState* has not been visited before (line 16), modelCheck then checks whether *nextState* violates a safety property (line 17). (The network protocol designer specifies the safety property that needs to be checked as a Java method whose output is true/false.) If so, a counterexample is printed by calling the *printPath()* function (line 18); otherwise, *nextState* is added to *NonVisitedStates* (line 20) in order to be explored later if its depth is strictly less than MAX_DEPTH. Adding a state to *NonVisitedStates* (line 20) or *AlreadyVisitedStates* (line 6) needs a function that creates a copy of a state (e.g., $clone()$).

For more implementation details, the interested reader is referred to [24]. It should be mentioned that the user needs to do a specific set of a few programming tasks (e.g., providing an implementation of *GlobalState*) before initiating the model checking process [26].

4. Evaluation and Results

In [26], we demonstrated the ability of the model checking framework in *J-Sim* to model-check two network protocols: (a) Ad-Hoc On-Demand Distance Vector (AODV) routing [22, 23] for wireless ad hoc networks, and (b) Directed Diffusion [12] for wireless sensor networks. These

```
procedure modelCheck()
 1. AlreadyVisitedStates = { } ;
 2. NonVisitedStates = { initialState } ;
 3. while ( | NonVisitedStates | > 0 ) {
 4.    currentState = NonVisitedStates.remove() ;
 5.    if ( currentState does not exist in AlreadyVisitedStates ) {
 6.       AlreadyVisitedStates = AlreadyVisitedStates ∪ { currentState } ;
 7.       for ( all protocol entities p ) { /* for all protocol entities */
 8.          for ( all possible events e ) { /* for all events */
 9.             NumberOfNextStates = e.EnablingFunction(p) ;
10.             for ( int i = 0 ; i < NumberOfNextStates ; i++ ) {
11.                CopyFromModelToEntities(currentState) ;
                   /* Start with nextState equal to currentState */
12.                nextState = currentState ;
                   /* Increment the depth of nextState */
13.                nextState.depth += 1 ;
14.                e.EventHandler(p) ;  /* Invoke e's event handler */
15.                CopyFromEntitiesToModel(nextState) ;
16.                if (nextState does not exist in AlreadyVisitedStates) {
17.                   if ( nextState.verifySafety() == false ) {
18.                      printPath(nextState) ; exit ;
                      } /* end if safety property is violated at nextState */
19.                   else if ( nextState.depth < MAX_DEPTH )
20.                        NonVisitedStates = NonVisitedStates ∪ { nextState } ;
                   }
                }
             }
          }
       }
    }
```

Figure 1. Stateful model checking procedure.

are reasonably complex network protocols whose *J-Sim* implementations (not including the *J-Sim* library) have about 1200 and 1400 lines of code, respectively. Our choice of AODV and directed diffusion was motivated by their potential to become representative routing and data dissemination protocols, respectively, in ad hoc networks and sensor networks. In this section, we give an overview of model-checking AODV in *J-Sim*. We ran all experiments on a Pentium 4 1.6 GHz machine with Microsoft Windows XP 2002 SP2 with 1 GB memory. We used Sun's Java 2 SDK 1.4.2 JVM with 512 MB allocated memory.

4.1. Overview of AODV

The implementation of AODV [22] in *J-Sim* is based on the AODV Draft (version 11) [23]. In AODV, each node n in the ad hoc network maintains a routing table. A routing table entry (RTE), at node n, to a destination node d contains, among other fields: $nexthop_{n,d}$ (the address of the node to which n forwards packets destined for d), $hops_{n,d}$ (the number of hops needed to reach d from n) and $seqno_{n,d}$ (a measure of the freshness of the route information). Each RTE is associated with a lifetime. Periodically, a route timeout event is triggered invalidating (but not deleting) all the RTEs 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 RTE involves incrementing $seqno_{n,d}$ and setting $hops_{n,d}$ to ∞.

When a node n requires a route to a destination d, it *broadcasts* a route request (RREQ) packet. When a node receives the RREQ, if it has a fresh enough route to d (or it is d itself), it satisfies the RREQ by *unicasting* a route reply (RREP) packet back to n; otherwise, it rebroadcasts the RREQ. The unicast RREP travels back to n. Each intermediate node along the path of RREP sets up a forward pointer

to the node from which the RREP came, thus establishing a forward route to d, and forwards the RREP packet to the next hop towards n. If node m offers node n a new route to d, n compares $seqno_{m,d}$ of the offered route to $seqno_{n,d}$, and accepts the route with the greater sequence number. If the sequence numbers are equal, the offered route is accepted only if $hops_{n,d} > hops_{m,d}$.

Each node maintains two monotonically increasing counters: $seqno_n$ and bid_n. When node n broadcasts a RREQ packet, it includes the current value of bid_n in the RREQ packet and then increments bid_n. Therefore, the pair $< n, bid_n >$ uniquely identifies a RREQ packet. Each node, receiving the RREQ packet from node n, 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, a broadcast ID timeout event is triggered causing the deletion of entries in the cache that have expired.

4.2. Model-checking AODV

Three steps constitute a generic methodology for model-checking a network protocol in *J-Sim*.

(1) Definitions of the global state, the initial state, state equality and safety property: We define *GlobalState* as a tuple that has two components; namely, the protocol state and the network cloud. The protocol state of a node n includes n's routing table, broadcast ID cache, $seqno_n$ and bid_n. The network cloud models the network as an unbounded set that contains AODV packets, 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 for one of the neighbors (i.e., the nodes that are within the transmission range) of s.

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

Two states, s_1 and s_2, are considered equal if they have the same (unordered) set of AODV packets, the same neighborhood information, and for each node n, s_1 and s_2 have equal corresponding values for $seqno_n$, bid_n, and node n's routing table and broadcast ID cache (each viewed as an unordered set of entries).

An important safety property in a routing protocol such as AODV is the *loop-free* property; i.e., data packets are not routed through loops. 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 as follows [18]:

$$((seqno_{n,d} < seqno_{m,d}) \vee (seqno_{n,d} == seqno_{m,d} \wedge hops_{n,d} > hops_{m,d}))$$

(2) Specifying the set of events, when each event is enabled, and how each event is handled. We specify six events as follows:

- T_0 Initiation of a route request to a destination d: This event is enabled if the node does not have a valid RTE to the destination d. The event is handled by broadcasting a RREQ.
- T_1 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 (or the next hop towards the destination) of the packet and n is one of the neighbors of the source of the packet. The event is handled by removing this packet from the network and forwarding it to node n.
- T_2 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.
- T_3 Loss of an AODV packet destined for node n: This event is enabled if the network contains at least one AODV packet that is destined for node n. The event is handled by removing this packet from the network.
- T_4 Broadcast ID timeout 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 this entry.
- T_5 Timeout of the route to destination d at node n: This event is enabled if n has a valid RTE to d. The event is handled by invalidating this RTE.

(3) Use of protocol-specific properties to facilitate a BeFS strategy: A suitable BeFS strategy for exploring the state space of AODV can be obtained by inspecting the loop-free property. A node, which does not have a valid RTE to the destination d, does not affect the truth value of the loop-free property. Therefore, a suitable BeFS strategy (which we call AODV-BeFS) is to consider a state s_1 better than a state s_2 if the number of *valid* RTEs to any node in s_1 is greater than that in s_2. We devised other BeFS strategies in [26].

4.3. Errors discovered

We consider an initial state of an ad hoc network consisting of 3 nodes: n_0, n_1 and n_2 (the only destination node) arranged in a chain topology where each node is a neighbor of both the node to its left and the node to its right (if any exists). Although this initial state is simple, it ensures that n_0 requires a multihop route to reach n_2; i.e., AODV multihop routing is needed. (We studied larger network topologies

in [26].) In the course of model checking, we have discovered an error (which we call Counterexample 1) in the *J-Sim* implementation of AODV caused by not following part of the AODV specification. Conceptually, if $nexthop_{0,2} = 1$ and the AODV process at n_1 restarts, the net effect is that all the RTEs stored at n_1 will be deleted. As a result, n_1 may later accept a route that was offered by n_2 with a lower sequence number than that of n_0 (i.e., $seqno_{0,2} > seqno_{1,2}$), hence violating the loop-free property. We also manually injected two errors (which we call Counterexamples 2 and 3 respectively): in Counterexample 2, $seqno_{n,d}$ is not incremented when a RTE is invalidated and in Counterexample 3, a RTE is deleted (instead of invalidated) when its lifetime expires. The model checking framework was able to find these two errors too. (For Counterexamples 2 and 3, we require that the counterexample contain at least one state that is generated due to the route timeout event, T_5.) A routing loop may occur due to either of these two errors because if $nexthop_{0,2} = 1$ and a route timeout event takes place at n_1, in either Counterexample 2 or 3, if n_1 is later offered a route to n_2 by n_0, this route will be accepted (because in Counterexample 2, $hops_{1,2} = \infty$; hence, $hops_{1,2} > hops_{0,2}$; whereas in Counterexample 3, $seqno_{0,2} > seqno_{1,2}$). The interested reader is referred to [25] for a detailed account (along with the traces) of the three counterexamples.

Table 1 gives the performance evaluation criteria: (i) time, (ii) space, and (iii) number of transitions explored for finding the three counterexamples using several search strategies, including breadth-first (BFS) and depth-first (DFS). As shown in Table 1, AODV-BeFS achieves an order of magnitude reduction with respect to the performance criteria when compared to BFS. In [26], we devised other BeFS strategies, studied their performance and discovered that the choice of the BeFS strategy has a significant impact on the performance. Based on our results. we provided recommendations for *good* search heuristics to model-check network protocols similar to AODV.

5. Related Work

Our work is inspired by previous work on model-checking the implementation code directly for C and C++ (e.g., CMC [18, 17] and VeriSoft [8]). Although CMC has been applied to model-check Linux implementations of networking code (e.g., AODV and TCP), the major distinction between our approach and CMC is that CMC uses *protocol-independent* properties in guiding the best-first search. It does so by attempting to focus on states that are the most different from previously explored states. However, our approach uses *protocol-dependent* properties, which exploit properties inherent to the network protocol and the safety property being checked, to guide the best-first search strategy. Likewise, VeriSoft uses *protocol-independent* techniques, namely partial-order reduction (POR) using the persistent/sleep sets [8].

In contrast to model-checking the implementation code directly, conventional model checkers (e.g., SPIN [11], SMV [16], Murphi [6]) require that the system be first specified using a high-level modeling language. This may not be desirable, as the process of describing the system in a high-level modeling language is time-consuming, painstaking, and error-prone. To deal with this problem, there has been recent work (e.g., [21, 9, 5, 7]) on translating programming languages (e.g., Java) into the input modeling languages of several conventional model checkers. However, this may not be always feasible because some features of C or Java (e.g., bit operations) do not have corresponding ones in the destination modeling language. Therefore, our approach of model-checking the simulation code, which has to be written by a network protocol designer anyway for the purpose of performance evaluation, directly reduces the network protocol designer's effort and avoids the limitations of the input languages of conventional model checkers. This also provides an important advantage when compared to previous work on testing and verification of network protocols (e.g., [15, 19]), which requires building *another* model for verification purposes.

Java PathFinder (JPF) [29] performs model checking at the bytecode level. This involved building a new Java Virtual Machine that is called from the model checker to interpret Java bytecode. In contrast, our approach does not require any modifications to the Java Virtual Machine. Our approach, however, requires the user to provide the code for state manipulation. JPF provides automatic manipulation of the *entire* Java states (including stack and heap).

As far as formal analysis of network simulation is concerned, Verisim [2] is a tool that was developed based on a collection of pre-existing tools; namely, ns-2 [20] and the MaC monitoring and checking framework [14]. 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.

6. Conclusions and Future Work

This paper documents our work on extending the *J-Sim* network simulator with the capability of finding bugs in network protocols using on-the-fly model checking. We demonstrated the effectiveness of the model checking framework in *J-Sim* to model-check AODV, a widely used and fairly complex network protocol. Experimental results show that the model checking framework in *J-Sim* is able to find violations of a safety property within acceptable time

Table 1. Time (in seconds) and space (in number of states explored) requirements and the number of transitions explored for finding the three counterexamples in a 3-node chain ad-hoc network using different search strategies. MAX_DEPTH = 10.

	Counterexample 1			Counterexample 2			Counterexample 3		
	Time	Space	Transitions	Time	Space	Transitions	Time	Space	Transitions
BFS	4262.039	19886	40445	4231.124	20072	40781	4094.928	19056	39489
DFS	940.672	1844	21135	962.935	1833	20979	893.896	1817	20814
AODV-BeFS	139.310	1156	7493	137.168	1151	7440	127.053	1150	7431

and space requirements, thus making *J-Sim* an integrated environment for both simulation and model checking of network protocols.

In our future work, we intend to make use of JPF to model-check the network protocols in *J-Sim*, and compare the model checking framework in *J-Sim* with that of JPF. The purpose of this comparative study is to assess the pros and cons of building a model checker in *J-Sim* instead of using an existing model checker such as JPF.

References

[1] T. Ball and S. K. Rajamani. The SLAM toolkit. In *Proc. of CAV'01*.

[2] 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*, 28(2):129–145, February 2002.

[3] S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular verification of software components in C. In *Proc. of ICSE'03*.

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

[5] 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*.

[6] 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*.

[7] A. Farzan, F. Chen, J. Meseguer, and G. Rosu. Formal analysis of Java programs in JavaFAN. In *Proc. of CAV'04*.

[8] P. Godefroid. Model checking for programming languages using VeriSoft. In *Proc. of ACM POPL'97*.

[9] K. Havelund. Java Pathfinder, A translator from Java to Promela. In *Proc. of SPIN'99*.

[10] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of POPL'02*.

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

[12] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proc. of ACM MobiCom'00*.

[13] J-Sim. http://www.j-sim.org/.

[14] 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*.

[15] D. Lee, D. Chen, R. Hao, R. E. Miller, J. Wu, and X. Yin. A formal approach for passive testing of protocol data portions. In *Proc. of IEEE ICNP'02*.

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

[17] M. Musuvathi and D. R. Engler. Model checking large network protocol implementations. In *Proc. of NSDI'04*.

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

[19] G. N. Naumovich, L. A. Clarke, and L. J. Osterweil. Verification of communication protocols using data flow analysis. In *Proc. of ACM SIGSOFT'96*.

[20] Ns-2. http://www.isi.edu/nsnam/ns/.

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

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

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

[24] A. Sobeih, M. Viswanathan, and J. C. Hou. Check and Simulate: A case for incorporating model checking in network simulation. In *Proc. of ACM-IEEE MEMOCODE'04*.

[25] A. Sobeih, M. Viswanathan, and J. C. Hou. Incorporating bounded model checking in network simulation: Theory, implementation and evaluation. Technical Report UIUCDCS-R-2004-2466, Department of Computer Science, University of Illinois at Urbana-Champaign, July 2004.

[26] A. Sobeih, M. Viswanathan, D. Marinov, and J. C. Hou. Finding bugs in network protocols using simulation code and protocol-specific heuristics. In *Proc. ICFEM'05, Springer-Verlag* LNCS *3785*.

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

[28] H.-Y. Tyan., A. Sobeih, and J. C. Hou. Towards composable and extensible network simulation. In *Proc. of IPDPS'05, NSF Next Generation Software Program Workshop*.

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