Finding Bugs in Network Protocols Using Simulation Code and Protocol-Specific Heuristics

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

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

Abstract. Traditional network simulators perform well in evaluating the performance of network protocols but lack the capability of verifying the correctness of protocols. To address this problem, we have extended the J-Sim network simulator with a model checking capability that explores the state space of a network protocol to find an execution that violates a safety invariant. In this paper, we demonstrate the usefulness of this integrated tool for verification and performance evaluation by analyzing two widely used and important network protocols: AODV and directed diffusion. Our analysis discovered a previously unknown bug in the J-Sim implementation of AODV. More importantly, we also discovered a serious deficiency in directed diffusion. To enable the analysis of these fairly complex protocols, we needed to develop protocol-specific search heuristics that guide state-space exploration. We report our findings on discovering *good* search heuristics to analyze network protocols similar to AODV and directed diffusion.

1 Introduction

Network simulators have been used for decades to provide an environment for a protocol designer to build a prototype of a network protocol and evaluate its performance. One major deficiency of traditional network simulators, however, is that they only evaluate the performance of network protocols in scenarios provided by the 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. If the error cases do not appear (and hence cannot be investigated) in the scenarios studied, subtle errors in the protocol specification/implementation may not be identified in the simulation. These errors may then eventually manifest themselves after the protocol has been implemented and deployed. In the light of recent research [1] that creates a physical implementation of a protocol from the existing simulation code, without modification, this seems to be highly likely. Therefore, building an integrated tool that allows a network protocol designer to *both* verify a prototype *and* evaluate its performance is an important task.

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 not only ensure that verifying a protocol is easier for the designer but also ensure that the model being verified is consistent with the implementation. Second, using a model checker for C or Java (like [2–7]) to verify the protocol code *along* with the simulator code might likely be intractable due to the complexity of the general-purpose simulator code.

We have built a tool that extends J-Sim [8]—a component-based network simulator written entirely in Java—with the model checking [9] 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). We previously provided a proof-of-concept case study [10] in which we used our tool to model-check an automatic repeat request (ARQ) protocol. In this paper, we demonstrate the usefulness and effectiveness of our tool in analyzing much more complicated protocol code. We examine two widely used and fairly complex network protocols: the Ad-Hoc On-Demand Distance Vector (AODV) routing protocol [11, 12] for wireless ad hoc networks and the directed diffusion protocol [13] for wireless sensor networks. These are reasonably complex 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. We investigate whether these protocols satisfy the *loop-free* safety property, i.e., data packets are not routed through loops.

Our surprising discoveries illustrate the practical importance of our tool. First, we find a previously unknown bug in the J-Sim implementation of AODV. This shows that even if the protocol specification [12] is correct, the simulator code could have bugs that may eventually find their way to the deployed implementation. Second, we identify a serious deficiency in the directed diffusion protocol [13] not only in its J-Sim implementation. Specifically, our tool produces scenarios leading to corruption of data caches due to timeouts and/or node reboots in a sensor network. This deficiency would result in data packets being routed in a loop.

To analyze such large protocol implementations, we have developed search heuristics that better guide the model checker to discover bugs. Specifically, we develop best-first search (BeFS) strategies that exploit *properties inherent to the network protocol and the safety property being checked*. An interesting and important research question is how to determine a suitable BeFS strategy for a specific network protocol. In this paper, we make an attempt towards answering this question by studying the performance of several BeFS strategies for both AODV and directed diffusion. Unlike [3, 14–16], we found that the strategies need to explicitly make use of *both* protocol-specific characteristics *and* the property being verified in order to be successful. The results show that using good protocol-specific heuristics outperforms standard breadth-first search (BFS) and depth-first search (DFS) strategies.

In this paper, we make the following contributions. First, we demonstrate the ability of our tool to find bugs in complex network protocols with large simulation code. Second, we discover a previously unknown deficiency in directed diffusion. Third, we report our findings on discovering *good* protocol-specific heuristics to analyze network protocols similar to AODV and directed diffusion.

The rest of the paper is organized as follows. In Section 2, we give an overview of the model checking framework in J-Sim, and in Section 3, we present our performance results. In Section 4, we discuss related work. Finally, we conclude the paper in Section 5 with a list of future research work.

2 The Model Checking Framework

The model checker, that we incorporated into J-Sim, is an explicit-state model checker [9] 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, keeps track of 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 already 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). 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

```
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) ;
12.                nextState = currentState ;  /* Start with nextState equal to currentState */
13.                nextState.depth += 1 ;  /* Increment the depth of nextState */
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 } ; } } } } } }
```

Fig. 1. Stateful model checking procedure

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*()).

It should be mentioned that the model checking process is not fully automated. To model-check a network protocol, the protocol designer needs to do the following:

1. Provide an implementation of *GlobalState* (including writing the safety property as a Java method, the function *equals*, and a function that creates a copy of a state), and specify how to construct the initial state. To reduce the protocol designer's burden, we provide an implementation of a class, called *SystemState*, that includes the protocol-independent information (e.g., the depth of a state, which event generated the state). *GlobalState*, which can be implemented as a sub-class of *SystemState*, includes the protocol-specific information.
2. Specify (a) the set of events that exist in the network protocol, (b) when each event is enabled, and (c) how each event is handled (i.e., an event handler that makes a transition from one state to another). Note that the protocol designer has to write the event handlers anyway 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.
3. Provide implementations for *CopyFromModelToEntities()* and *CopyFromEntitiesToModel()*. To facilitate programming, we make use of *ports* (a feature provided by J-Sim) to provide a seamless interface between components; in this case, between the model checker and the protocol entities [17].
4. (Required only in the case of using a BeFS strategy) Write a Java method that assigns to each state a metric that represents how "good" this state is. The model checking procedure will explore the "best" state first.

3 Evaluation and Results

We applied the model checking framework to the J-Sim implementations of the AODV (Section 3.1) and directed diffusion (Section 3.2) protocols. For each protocol, we give an overview of the key functionality, describe the protocol actions and property being checked, present several BeFS heuristics, discuss detected errors, and show performance results for model checking. 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.

3.1 AODV Routing in Multihop Wireless Ad Hoc Networks

Overview of AODV. The implementation of AODV [11] in J-Sim is based on the AODV Draft (version 11) [12]. 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.

Model checking AODV. We next present the steps that we follow to model-check AODV. These 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. 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 [3, 18]:

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

(2) Events: Next, we specify the set of events, when each event is enabled, and how each event is handled. The events can be listed 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-1) 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. Another

BeFS strategy (which we call AODV-BeFS-2) can also be obtained by inspecting the loop-free property, which can be rewritten as follows:

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

Therefore, the greater $(seqno_{n,d} - seqno_{m,d})$ and/or $(hops_{m,d} - hops_{n,d})$ in a state s, the more likely s is close to an error. Hence, AODV-BeFS-2 considers a state s_1 better than a state s_2 if the following summation

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

in s_1 is greater than that in s_2, where $nexthop_{n,d} = m$. The summation S includes only the nodes n and m that have valid RTEs to the destination d. If none of the nodes have a valid RTE to d, S is set to $-\infty$. In addition to AODV-BeFS-1 and AODV-BeFS-2, we also study the performance of the following BeFS strategies:

1. AODV-BeFS-3: This strategy considers a state s_1 better than a state s_2 if the number of valid RTEs *to the destination* d in s_1 is greater than that in s_2. However, if s_1 and s_2 are equally good, s_1 is considered better than s_2 if the number of valid RTEs *to any node* in s_1 is greater than that in s_2.
2. AODV-BeFS-4: Since a valid RTE is established upon receiving a RREP packet, AODV-BeFS-4 considers a state s_1 better than a state s_2 if the number of RREP packets in s_1 is greater than that in s_2.
3. AODV-BeFS-5: AODV-BeFS-5 is the same as AODV-BeFS-4, except that if s_1 and s_2 are equally good under the condition specified in AODV-BeFS-4, s_1 is considered better than s_2 if the number of valid RTEs to any node in s_1 is greater than that in s_2.

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 will study larger network topologies later in this section. 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.[1] 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 [17] for a detailed account (along with the traces) of the three counterexamples.

[1] 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.

Table 1. AODV case study: 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-1	139.310	1156	7493	137.168	1151	7440	127.053	1150	7431
AODV-BeFS-2	833.719	1753	19617	810.035	1750	19581	775.766	1739	19468
AODV-BeFS-3	14.882	535	2118	14.120	535	2079	14.400	534	2070
AODV-BeFS-4	367.038	1626	14151	3905.015	4901	44851	365.215	1617	14051
AODV-BeFS-5	347.529	1923	13577	3076.274	4649	38853	323.515	1889	13101

Table 2. AODV case study: Time (in seconds) and space (in number of states explored) requirements and the number of transitions explored for finding Counterexample 3 in a N-node chain ad-hoc network using AODV-BeFS-1

N	MAX_DEPTH	Time	Space	Transitions
3	15	0.200	93	134
4	20	12.609	575	1971
5	25	944.769	3256	19803
6	30	1393.955	2640	25052
7	35	3784.462	3339	46532

Performance of the search strategies. 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-1 achieves an order of magnitude reduction with respect to the performance criteria when compared to BFS. Also, the choice of the BeFS strategy has an impact on the performance. For instance, as shown in Table 1, AODV-BeFS-2 performs worse than AODV-BeFS-1 for the three counterexamples. This is because AODV-BeFS-2 requires a node (and its next hop towards the destination) to have valid RTEs to the destination. This may not be true in the first few stages (i.e., lower depths) of the search space. Therefore, in the first few stages of the search, the nonvisited states may look equally good and thus, AODV-BeFS-2 may not be able to explore the states that are most likely to lead to the error first. AODV-BeFS-3 tackles this problem by further differentiating equally good states by using a two-level best-first search approach. As shown in Table 1, AODV-BeFS-1 and AODV-BeFS-3 outperform the other BeFS strategies because they are more able to guide the BeFS towards the error even at the lower depths of the search space.

Next, we study the effect of the size of the network on the performance of the model checking framework in J-Sim. As shown in Table 2, the model checking framework was able to find a counterexample in larger network topologies within reasonable time and space requirements.

3.2 Directed Diffusion in Wireless Sensor Networks (WSNs)

Overview of directed diffusion. Directed diffusion [13] is a *data-centric* information dissemination paradigm for wireless sensor networks (WSNs). In directed diffusion, a *sink node* periodically broadcasts an INTEREST packet, containing the

description of a sensing task that it is interested in (e.g., detecting a chemical weapon in a specific area). INTEREST packets are diffused throughout the network (e.g., via flooding), and are used to set up *exploratory gradients*. A gradient is the direction state created in each node that receives an INTEREST, where the gradient direction is set toward the neighboring node from which the INTEREST is received. Each node maintains an interest cache. Each interest entry in this cache corresponds to a distinct interest and stores information about the gradients that a node has (up to one gradient per neighbor) for that interest. Each gradient in an interest entry has a lifetime that is determined by the sink node. When a gradient expires, it is removed from its interest entry. When all gradients in an interest entry have expired, the interest entry itself is removed from the interest cache.

When an INTEREST packet arrives at a *sensor node* that senses data which matches the interest (this sensor node is called a *source node*), the source node prepares DATA packets (each of which describes the sensed data) and sends them to neighbors for whom it has a gradient once every *exploratory interval*. Each node also maintains a data cache that keeps track of recently seen DATA packets. When a node receives a DATA packet, if the DATA packet has a matching data cache entry, the DATA packet is discarded; otherwise, the node adds the received DATA packet to the data cache and forwards the DATA packet to each neighbor for whom it has a gradient. As a result, DATA packets are forwarded toward the sink node(s) along (possibly) multiple gradient paths.

Upon receipt of a DATA packet, a sink node *reinforces* its preferred neighbor that is determined based on a data-driven local rule. For instance, the sink node may reinforce any neighbor from which it received previously unseen data (i.e., the neighbor from which it first received the latest data matching the interest). The data cache is used to determine that preferred neighbor. In order to reinforce a neighbor, the sink node sends a *positive reinforcement* packet to that neighbor to inform it of sending data at a smaller interval (i.e., higher rate) than the exploratory interval, thereby establishing a *reinforced gradient* (also called *data gradient*) towards the sink node. The reinforced neighbor will in turn reinforce its preferred neighbor. This process repeats all the way back to the data source, resulting in a reinforced path (i.e., a chain of reinforced gradients) between the source and the sink nodes.

Model checking directed diffusion. In order to illustrate the applicability of the model checking framework, we follow the same steps given in Section 3.1.

(1) Definitions of the global state, the initial state, state equality and safety property: To model-check directed diffusion, we use the same definitions of *GlobalState* and network cloud that were introduced in Section 3.1. On the other hand, since the protocol state is protocol-specific, the protocol state in directed diffusion includes each node's interest cache and data cache. In the initial global state, the network does not contain any packets and the directed diffusion process at each node starts with an empty interest cache and an empty data cache.

Two states, s_1 and s_2, are considered equal if they have the same (unordered) set of packets, the same neighborhood information, and for each node n, s_1 and s_2 have correspondingly equal node n's interest cache and data cache (each viewed as an unordered set of entries).

An important safety property in the directed diffusion protocol is the *loop-free* property of the reinforced path. Consider two nodes n and m where $RPath(n, m)$ is true if and only if there is a reinforced path from n to m. The loop-free property can be expressed as follows:

$$\neg \; (\; RPath(n, m) \; \wedge \; RPath(m, n) \;)$$

(2) Events: The events can be listed as follows:

T_0 Initiation of a sensing task by node n: This event is enabled if n is a sink node. The event is handled by broadcasting an INTEREST packet.

T_1 Delivering a packet to node n: This event is enabled if the network contains at least one packet that is destined for node n such that node 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 directed diffusion 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 directed diffusion process at node n.

T_3 Loss of a packet destined for node n: This event is enabled if the network contains at least one packet that is destined for node n. The event is handled by removing this packet from the network.

T_4 Gradient timeout at node n: This event is enabled if the interest cache of node n contains at least one interest entry that has at least one gradient. The event is handled by deleting this gradient.

T_5 Data cache timeout[2] at node n: This event is enabled if there is at least one entry in the data cache of node n. The event is handled by deleting this entry.

(3) Use of protocol-specific properties to facilitate a BeFS strategy: In the course of model-checking AODV, AODV-BeFS-1 and AODV-BeFS-3 provided comparatively good performance results. We use these two BeFS strategies to devise two corresponding BeFS strategies for directed diffusion. In particular, as the loop-free property involves only valid RTEs to a destination d in AODV; by analogy, the loop-free property involves only reinforced gradients in directed diffusion. Similarly, forwarding of data packets in AODV is based on the next hop information stored in the valid RTEs; by analogy, forwarding of data packets in directed diffusion is based on the gradients established at the nodes. Therefore, two good BeFS strategies for exploring the state space of directed diffusion are:

1. DD-BeFS-1: This strategy considers a state s_1 better than a state s_2 if the total number of *both exploratory and reinforced gradients* in s_1 is greater than that in s_2.
2. DD-BeFS-2: This strategy considers a state s_1 better than a state s_2 if the number of *reinforced* gradients in s_1 is greater than that in s_2. However, if s_1 and s_2 are equally good, s_1 is considered better than s_2 if the total number of *both exploratory and reinforced* gradients in s_1 is greater than that in s_2.

[2] For practical reasons, previously received DATA packets can not be kept in the data cache for an indefinitely long time; otherwise, the size of the data cache can increase arbitrarily. In the implementation of directed diffusion in J-Sim, each DATA packet in the data cache is associated with a lifetime. Periodically, a data cache timeout event is triggered causing the deletion of entries in the cache that have expired.

Along a similar line of arguments, we also devise the following BeFS strategies:

1. DD-BeFS-3: Since a reinforced gradient is established upon receiving a positive reinforcement packet, DD-BeFS-3 considers a state s_1 better than a state s_2 if the number of positive reinforcement packets in s_1 is greater than that in s_2.
2. DD-BeFS-4: DD-BeFS-4 is the same as DD-BeFS-3, except that if s_1 and s_2 are equally good under the condition specified in DD-BeFS-3, s_1 is considered better than s_2 if the total number of *both exploratory and reinforced* gradients in s_1 is greater than that in s_2.
3. DD-BeFS-5: This strategy considers a state s_1 better than a state s_2 if the total number of data cache entries at all nodes in s_1 is greater than that in s_2.
4. DD-BeFS-6: DD-BeFS-6 is the same as DD-BeFS-5, except that if s_1 and s_2 are equally good under the condition specified in DD-BeFS-5, s_1 is considered better than s_2 if the total number of *both exploratory and reinforced* gradients in s_1 is greater than that in s_2.

Errors discovered. Next, we give two previously unknown errors that the model checking framework in J-Sim was able to discover in directed diffusion (which we call Counterexamples 1 and 2 respectively). We consider an initial state that consists of a chain topology of 4 nodes: n_0 (the only sink node), n_1, n_2 and n_3 (the only source node). The errors take place because in directed diffusion, the interest and gradient setup mechanisms themselves do *not* guarantee loop-free reinforced paths between the source and the sink nodes. In order to prevent loops from taking place, the data cache is used to suppress previously seen DATA packets. However, we discover that, in case of (a) a node reboot (which effectively deletes all the entries in the data and interest caches) and/or (b) the deletion of a DATA packet from the data cache, a loop may be created. For instance, in the 4-node chain topology, if n_1 accepts a DATA packet sent by n_2, n_2 becomes n_1's preferred neighbor. Now, if n_2 deletes the DATA packet from its data cache due to a data cache timeout (Counterexample 1) or a node reboot (Counterexample 2), it may later accept the DATA packet sent by n_1 (because it will be previously unseen data) causing n_1 to become n_2's preferred neighbor. Therefore, n_1 and n_2 may positively reinforce each other causing a loop in the reinforced path. In fact, positive reinforcement packets may not eventually reach the source node causing a disruption in the reinforced path (i.e., the reinforced path may include a loop that does not include the source node).[3] The interested reader is referred to [19] for a detailed account, and traces, of the two counterexamples.

Performance of the search strategies. Table 3 gives the performance of the various search strategies in finding the two counterexamples. As shown in Table 3, DD-BeFS-1 provides comparatively good results in terms of time and space requirements and the number of transitions explored for finding a violation of a safety property. Furthermore, DD-BeFS-4 outperforms DD-BeFS-3, and DD-BeFS-6 outperforms DD-BeFS-5. This is because both DD-BeFS-4 and DD-BeFS-6 are two-level

[3] For Counterexample 2, we require that the counterexample contain at least one state that is generated due to a node reboot event, T_2. Furthermore, in order to show that the error may still take place even if the data cache timeout event, T_5, does not happen (i.e., the data cache size is infinite), we disabled T_5.

Table 3. Directed diffusion case study: Time (in seconds) and space (in number of states explored) requirements and the number of transitions explored for finding the two counterexamples in a 4-node chain sensor network using different search strategies. N/A indicates that the model checker was not able to find a counterexample in 8 hours

	Counterexample 1, $MAX\_DEPTH = 15$			Counterexample 2, $MAX\_DEPTH = 20$		
	Time	Space	Transitions	Time	Space	Transitions
BFS	22287.938	21224	84530	N/A	N/A	N/A
DFS	23876.914	4736	95706	N/A	N/A	N/A
DD-BeFS-1	3.475	200	1051	3900.118	6026	41132
DD-BeFS-2	4.026	200	1168	12189.227	6640	57124
DD-BeFS-3	19536.362	4630	93924	N/A	N/A	N/A
DD-BeFS-4	0.981	124	469	726.024	1870	17656
DD-BeFS-5	N/A	N/A	N/A	N/A	N/A	N/A
DD-BeFS-6	24743.349	12920	72911	N/A	N/A	N/A

Table 4. Directed diffusion case study: Time (in seconds) and space (in number of states explored) requirements and the number of transitions explored for finding Counterexample 1 in a N-node chain sensor network using DD-BeFS-4

N	MAX_DEPTH	Time	Space	Transitions
4	15	0.981	124	469
5	20	335.833	925	11816
6	25	857.303	1346	19245
7	30	1538.152	1985	27640
8	35	7244.277	3679	59093

BeFS strategies that use DD-BeFS-1 if the non-visited states are equally good and are thus more able to guide the BeFS in the lower depths of the search space than DD-BeFS-3 and DD-BeFS-5 respectively.

Table 4 gives the time and space requirements and the number of transitions explored for finding Counterexample 1 in a chain topology consisting of N nodes using DD-BeFS-4. For sensor networks consisting of more than four nodes, both BFS and DFS failed to find counterexamples.

3.3 Lessons Learned

In this subsection, we summarize the lessons that we learned. First, the ability of the model checking framework to model-check large and complex network protocols such as AODV and directed diffusion demonstrates that the model checking framework is general enough and not tied to a particular network protocol. Specifically, for model-checking another network protocol, one needs to follow the steps that we followed in sections 3.1 and 3.2.

Second, we demonstrate that the use of BeFS strategies (that leverage *protocol-specific* properties) reduces the time and space requirements by several orders of magnitude. Based on the results obtained for the BeFS strategies that we studied, we recommend deriving the BeFS strategy from properties inherent to the network protocol and the safety property being checked. This is justified by the fact that AODV-BeFS-1 (and DD-BeFS-1) provided good performance results in terms of time and space requirements and number of transitions explored for finding a violation of a safety property. Furthermore, using a two-level BeFS strategy, in which a BeFS strategy such as AODV-BeFS-1 (or DD-BeFS-1) is used if the nonvisited states are

equally good, also improved the performance. This is justified by the fact that AODV-BeFS-5 outperforms AODV-BeFS-4, DD-BeFS-4 outperforms DD-BeFS-3, and DD-BeFS-6 outperforms DD-BeFS-5.

4 Related Work

Our work is inspired by previous work on model-checking the implementation code directly for C and C++ (e.g., CMC [3, 14] and VeriSoft [20]). 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 [20]. Traditional POR was static, but recent work shows how to perform dynamic POR [21]. POR can be combined with BeFS strategies; while POR determines what transitions to explore, BeFS determines the order in which to explore them [22].

In contrast to model-checking the implementation code directly, conventional model checkers (e.g., SPIN [23], SMV [24], Murphi [25]) 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., [2,7,26,27]) 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., [28, 29]), which requires building *another* model for verification purposes.

Java PathFinder [30] 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 (Section 2). Java PathFinder provides automatic manipulation of the *entire* Java states (including stack and heap); to use Java PathFinder for a tractable checking of protocol simulation code in J-Sim, the user would still need to manually provide the code that manipulates state by abstracting the stack and parts of the heap.

The idea of using best-first search strategies and/or heuristics to expedite the model checking process has been explored in previous work (e.g., [15, 22, 31–33]). However, what distinguishes our work is that we study the use of protocol-specific heuristics in model-checking the simulation code directly and we focus on a specific

domain; namely, routing protocols for wireless ad hoc and sensor networks, and attempt to discover effective protocol-specific heuristics that enable a best-first search strategy to find counterexamples with less time and space requirements than classic breadth-first and depth-first search strategies.

5 Conclusions and Future Work

This paper presents our research on extending the J-Sim network simulator with the capability of verifying network protocols using on-the-fly model checking. We demonstrate the effectiveness of the model checker to model-check two widely used and fairly complex network protocols: AODV and directed diffusion. To the best of our knowledge, the deficiency identified in directed diffusion has not been discovered before. Experimental results show that the model checker is able to find violations of a safety property within acceptable time and space requirements. Furthermore, we study several best-first search strategies for both AODV and directed diffusion, and provide recommendations based on our results.

We have identified several research avenues for future work. First, we intend to extend the model checker to check general temporal properties. Second, the experiments reported in this paper require considerable manual effort; in future research, we will consider how to reduce such manual effort. An important research question is how to (semi-)automatically derive the heuristics from the simulation code and the safety property. Another interesting research avenue lies in studying to what extent symbolic model checking can expedite model-checking the simulation code.

References

1. A. K. Saha, K. To, S. PalChaudhuri, S. Du, and D. B. Johnson, "Physical implementation of ad hoc network routing protocols using unmodified ns-2 models," ACM MobiCom'04, Poster.
2. K. Havelund, "Java Pathfinder, a translator from Java to Promela," in *Proc. of SPIN'99*.
3. 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*.
4. T. Ball, and S. K. Rajamani, "The SLAM Toolkit," in *Proc. of CAV'01*.
5. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, "Modular Verification of Software Components in C," in *Proc. of ICSE'03*.
6. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy Abstraction," in *Proc. of POPL'02*.
7. A. Farzan, F. Chen, J. Meseguer, and G. Rosu, "Formal analysis of Java programs in JavaFAN," in *Proc. of CAV'04*.
8. J-Sim, "http://www.j-sim.org/"
9. E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 1999.
10. 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*.
11. C. E. Perkins and E. M. Royer, "Ad-hoc on-demand distance vector routing," in *Proc. of IEEE WMCSA'99*.
12. C. E. Perkins, E. M. Royer, and S. Das, "Ad hoc on demand distance vector (aodv) routing," IETF Draft, January 2002.
13. C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," in *Proc. of ACM MobiCom'00*.

14. M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *Proc. of NSDI'04*.
15. S. Edelkamp, S. Leue and A. Lluch-Lafuente, "Directed Explicit-State Model Checking in the Validation of Communication Protocols," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 5, no. 2-3, pp. 247–267, March 2004.
16. P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for heuristic determination of minimum path cost," *IEEE Transactions on Systems Science and Cybernetics*, vol. 4, pp. 100–107, 1968.
17. A. Sobeih, M. Viswanathan and J. C. Hou, "Incorporating Bounded Model Checking in Network Simulation: Theory, Implementation and Evaluation," Tech. Rep. UIUCDCS-R-2004-2466, Department of Computer Science, University of Illinois at Urbana-Champaign, July 2004.
18. K. Bhargavan, D. Obradovic, and C. A. Gunter, "Formal verification of standards for distance vector routing protocols," *Journal of the ACM*, vol. 49, no. 4, pp. 538–576, July 2002.
19. A. Sobeih, M. Viswanathan and J. C. Hou, "Bounded Model Checking of Network Protocols in Network Simulators by Exploiting Protocol-Specific Heuristics," Tech. Rep. UIUCDCS-R-2005-2547, Department of Computer Science, University of Illinois at Urbana-Champaign, April 2005.
20. P. Godefroid, "Model checking for programming languages using VeriSoft," in *Proc. of ACM POPL'97*.
21. C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of ACM POPL'05*.
22. P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *Proc. of TACAS'02*.
23. G. J. Holzmann, "The model checker SPIN," *IEEE Trans. on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997.
24. K. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.
25. 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*.
26. D. Y. Park, U. Stern, J. U. Skakkebæk, and D. L. Dill. Java model checking. In *Proc. of IEEE ASE'00*.
27. 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*.
28. 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*.
29. G. N. Naumovich, L. A. Clarke, and L. J. Osterweil, "Verification of communication protocols using data flow analysis," in *Proc. of ACM SIGSOFT'96*.
30. W. Visser, K. Havelund, G. Brat, and S.Park, "Model checking programs," in *Proc. of IEEE ASE'00*.
31. J. Tan, G. S. Avrunin, L. A. Clarke, S. Zilberstein, and S. Leue. Heuristic-guided counterexample search in FLAVERS. In *Proc. of ACM SIGSOFT'04/FSE-12*.
32. C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Proc. of ACM/IEE DAC'98*.
33. A. Groce and W. Visser "Heuristics for Model Checking Java Programs," *International Journal on Software Tools for Technology Transfer (STTT)*, vol. 6, no. 4, pp. 260–276, August 2004.