Bounded Model Checking of Network Protocols in Network Simulators by Exploiting Protocol-Specific Heuristics

Ahmed Sobeih, Mahesh Viswanathan, and Jennifer C. Hou

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

Abstract. Network simulators perform well in evaluating the performance of network protocols, but lack the capability of verifying their correctness. In order to address this lacuna, we have extended the J-Sim network simulator with a model checking capability to explore the state space of a network protocol to find either an execution where a safety invariant is violated or an exceution where the satisfaction of an eventuality property is witnessed. 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: Ad-Hoc On-Demand Distance Vector (AODV) routing protocol and directed diffucion protocol. Our analysis discovered a previously unknown bug in the J-Sim implementation of AODV, and a design flaw in directed diffusion, illustrating that our tool can uncover both bugs in the implementation and design of a protocol. To enable the analysis of these fairly complex protocols, we needed to develop search heuristics to explore the state space. We report our preliminary findings on discovering good search heuristics to analyze routing protocols.

1 Introduction

Prior to deploying a network protocol, it is critical for a designer to be able to build a prototype of the protocol, integrate it with other interacting protocols in the network protocol stack, and evaluate its performance with respect to certain pre-selected networking metrics (e.g., system throughput, packet delivery ratio, and end-to-end delay) with the goal of re-designing the protocol or its parameter fine tuning if needed. Several existing network simulators (e.g., ns-2 [1] and J-Sim [2]) provide a protocol designer with just such an environment to build protoypes and analyze them. One major deficiency of existing network simulators, however, is that they only evaluate the performance of network protocols in scenarios provided by the user, and 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 identified in the simulation. The errors may then eventually manifest themselves after the protocol has been implemented and deployed; especially, in the light of recent research [3] that creates a physical implementation of a protocol from the existing simulation code, without modification, this seems to be highly likely. The presence of errors in a deployed protocol could lead to serious, and sometimes disastrous, consequences, like having a routing protocol that routes packets along network loops.

Therefore, building an integrated tool that allows a network protocol designer to both verify a prototype and evaluate its performance is an important task. Having 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, we not only ensure that verifying a protocol is easier for the designer (and hence something he/she is more likely to perform) but also ensure that the model being verified is consistent with the implementation. Second, using a model checker for C or Java (like [4–9]) to verify the protocol code along with the simulator code is unlikely to be feasible. Moreover, in many cases the model checker for the network simulator can conveniently ignore the call stack and heap when performing the state space search, and this can lead to a more tractable analysis.

Motivated thus, we have extended J-Sim [2] — an open-source network simulator that is written entirely in Java — with the model checking [10] capability to explore the state space created by a network protocol to find either violations of a desirable safety property or witnesses for a desirable liveness property if any exists. In particular, we have incorporated bounded model checking (BMC) [11, 12] into J-Sim [13] to explore the state-space up to a (configurable) maximum depth in order to discover an execution that violates a safety invariant or one in which an eventuality requirement is satisfied.

In this paper, we demonstrate the usefulness and effectiveness of such a tool in analyzing protocol code. We examine two widely used and fairly complex network protocols; the Ad-Hoc On-Demand Distance Vector (AODV) routing protocol [14, 15] for wireless ad hoc networks and the directed diffusion protocol [16] for wireless sensor networks[1]. Our choice of AODV and directed diffucion was motivated by their potential to become representative routing and data dissemination protocols, respectively, in ad hoc networks and sensor networks. We investigated whether these protocols satisfy the *loop-free* safety property, i.e., we checked if the protocols ensure that data packets are not routed through network loops. Our surprising discoveries reported here, illustrate the practical importance of our tool.

First, we discovered a previously unknown bug in the J-Sim implementation of AODV. This demonstrates that even if the protocol specification [15] is correct, the simulator code could have bugs which may eventually find its way to the deployed protocol. Second, we found a design flaw in directed diffusion. In particular, our tool produced scenarios leading to the corruption of the data

[1] These are reasonably complex protocols whose J-Sim implementation (not including the J-Sim library) has about 1200 to 1400 lines of code.

cache due to timeouts and/or node reboots in a sensor network, that would result in data packets being routed in a loop and never reaching the destination. This shows that the tool is also capable of discovering design flaws involving real-time features such as timeouts.

In order to analyze such large protocol implementations, we discovered that we needed to develop search heuristics that would enable us to discover bugs. We developed 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 using time and space overheads that are several orders of magnitude less than those incurred by standard search strategies like breadth first and depth first. An interesting and important question is how to determine a suitable best-first search strategy for a specific network protocol. In this paper, we make an attempt towards answering this question by studying the performance of several best-first search strategies for both AODV and directed diffusion. Unlike [5, 17–19], 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 rest of the paper is organized as follows. In Section 2, we discuss related work and explain what distinguishes our paper from previous work. In Section 3, we describe the implementation of BMC in J-Sim. Following that, we present our performance results in Section 4. Finally, we conclude the paper in Section 5 with a list of future research work.

2 Related Work

Conventional model checkers (e.g., SPIN [20], SMV [21], Murphi [22]) have been used in formal reasoning of distributed systems. In [23], 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. This may not be desirable, as the process of writing a model of a system is known to be a time-consuming, painstaking, and error-prone process. In [24], the process of writing the model is automated by using an extensible compiler, xg++, that can extract a model (described in the Murphi input language) from the original implementation code written in C. However, the technique is not fully automatic because the user still needs to specify the state variables and subroutines that should be extracted into the model and also needs to specify rules for translating the extracted code into a Murphi model. Furthermore, the Murphi input language is more minimal than C; hence, it does not provide some of the features that C does (e.g., bit operations). Compared to [24], our goal is not to extract a model, but instead to model check the J-Sim protocol implementation code itself, which has to be written by a network protocol designer anyway for the purpose of performance evaluation. Therefore, our approach 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., [25, 26]), which requires building *another* model for verification purposes.

The BMC framework in J-Sim 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 previous work on model checking the implementation code directly (e.g., CMC [5, 17] and Verisoft [27]) for C and C++. Although CMC has been applied to model check implementations of networking code (namely, 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 are derived from the safety or liveness property being checked, to guide the best-first search strategy. Likewise, Verisoft uses *protocol-independent* properties (e.g., partial order reduction). Another notable research is HSF-SPIN [18], which extends SPIN by using protocol-independent heuristics in order to guide the A* Algorithm [19] towards safety property violations. However, a model of the protocol being checked needs to be written in Promela, the input language of SPIN. Furthermore, results have shown that SPIN outperformed HSF-SPIN in terms of time and space overheads [18].

A recent version of Java PathFinder [28] 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. In contrast, our approach does not require any modifications to the Java Virtual Machine.

The Maude LTL model checker [29] has been used in the verification of the AER/NCA network protocol [30] (which also has to be first specified in Maude by means of object-oriented modules). In [13], 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.

3 Implementation of Bounded Model Checking in J-Sim

In this section, we discuss how we implement the BMC framework in J-Sim. The 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 up to a maximum depth MAX_DEPTH. Specifically, the protocol designer specifies the safety and/or liveness properties that need to be checked and then the model checker starts from an initial state and recursively generates successor system states by executing the transitions of the system until either a counterexample disproving a safety property or a witness proving a liveness property is located, or the state space is explored up to MAX_DEPTH, whichever occurs first[2].

[2] In this paper, we consider only universal safety properties and existential liveness properties.

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, we have defined a class, *GlobalState*, which includes the set of system variables and the depth of the state as data members. 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*. The protocol designer is also responsible for identifying the transitions and the event handlers for them in the protocol code that are relevant for model checking.

The model checking procedure is then a stateful exploration of the state space, where the search strategy can be breadth-first, depth-first or based on search heuristics. Since the model checker steps through the state space of the protocol by executing the protocol's simulation code, there are a couple of fundamental challenges that must be overcome. First, we need to ensure that the simulator can be guided to take steps on a need basis, without affecting the setting and handling of timeout events that are used by the simulator to address the real-time constraints of the protocol. Second, since event handlers invoked by the model checker are executed inside the protocol entities (i.e., the J-Sim classes that implement the entities comprising the network protocol being model-checked) themselves, the model checking procedure must first restore the state of the protocol entities to the state from which the model checker would like to take a step before the execution of the event handler. Details of how we overcome these problems and seamlessly integrate a model checker into a network simulator can be found in [13, 31].

4 Evaluation and Results

In this section, we demonstrate how we use the BMC framework to model-check the implementation of the AODV routing protocol (Section 4.1) and the directed diffusion protocol (Section 4.2) in J-Sim. 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.

4.1 AODV Routing in Wireless Ad Hoc Networks

Overview of AODV. The implementation of AODV [14] in J-Sim is based on the AODV Draft (version 11) [15]. Due to the page limit, we will not give the complete description of AODV, but that which pertains to the work reported in this paper. The interested reader is referred to [31] for a detailed account of AODV. In AODV, 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: $nexthop_{n,d}$ (the address of the node to which packets destined for destination d are forwarded), $hops_{n,d}$ (the number of hops needed to reach the destination d) and $seqno_{n,d}$ (a measure of the freshness of the route information). Each routing table entry is associated with a lifetime and a route timeout event is triggered invalidating (but not deleting) 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 ∞.

When a node n requires a route to a destination d, it *broadcasts* a route request (RREQ) packet. When a node receives a RREQ, 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 if it does not have a fresh enough route to the destination (nor is it the destination itself). The unicast RREP travels back to the requesting node. 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 the destination and forwards the RREP packet to the next hop towards the requesting node. If a new route to destination d is offered to node n by node m, node 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 the same, the offered route is accepted only if $hops_{n,d} > hops_{m,d}$.

Model checking the AODV protocol. In this section, we present the steps that we follow in order to model check AODV. These steps constitute a generic methodology for model checking any network protocol in J-Sim:

1. Defining a global state and specifying the initial state(s).
2. Specifying the state transitions, when each transition occurs, and how each transition is handled.
3. Specifying the safety and/or liveness properties.
4. Using protocol-specific heuristics and properties to mitigate the state space explosion problem and devise a best-first search strategy.

Steps 1 and 2 are explained in Appendix A. For Step 3, 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 [32, 5]:

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

A desirable liveness property in a routing protocol, such as AODV, is that there exists a seqeuence of events such that nodes that require a route to a destination d will eventually have a valid routing table entry to d.

In the course of model checking AODV, we made use of three search strategies in exploring the state space: breadth-first (BFS), depth-first (DFS) and best-first (BeFS). In Step 4, we devise best-first search strategies making use of protocol-specific properties as follows. A suitable best-first search strategy for exploring the state space of AODV can be obtained by inspecting the loop-free property. 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 call AODV-BeFS-1) is to consider a state s_1 better than a state s_2 if the number of *valid* routing table entries to any node in s_1 is greater than that in s_2. In order to study the effect of the choice of the best-first search strategy, we investigate another best-first search strategy (which we call AODV-BeFS-2) that is also obtained by inspecting the loop-free property. The loop-free property 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 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 AODV-BeFS-2, we consider 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 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 $-\infty$. In addition, we also studied the performance of the following best-first search strategies:

1. AODV-BeFS-3: This strategy considers a state s_1 better than a state s_2 if the number of valid routing table entries *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 routing table entries *to any node* in s_1 is greater than that in s_2
2. AODV-BeFS-4: Since the valid routing table entries are established upon the reception of RREP packets, another best-first search strategy is to consider 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 on top of all the conditions, if s_1 and s_2 are equally good, s_1 is considered better than s_2 if the number of valid routing table entries to any node in s_1 is greater than that in s_2.

Errors discovered. Next, we explain the errors that the BMC framework in J-Sim discovered. We consider an initial state of an ad hoc network consisting of 3 nodes: $n0$, $n1$ and $n2$ (the only destination node) arranged in a chain topology where each node is within the transmission range of (i.e., can reach in one hop) both the node to its left and the node to its right if any exists. 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. We will study larger network topologies too. MAX_DEPTH was set to 15. We discovered an error in the J-Sim implementation of AODV caused by not following part of the AODV specification (details can be found in [31]). We will call that error Counterexample 1. Conceptually, if $nexthop_{0,2} = 1$ and the AODV process at $n1$ restarts (e.g., due to a node reboot), the net effect is that all the routing table entries stored at $n1$ will be deleted. As a result, $n1$ may later accept a route that was offered by $n2$ with a lower sequence number than that of $n0$ (i.e., $seqno_{0,2} > seqno_{1,2}$), hence violating the loop-free property. The trace and explanation of Counterexample 1 (shown in Figure 1) are given in Appendix A. We also manually injected two errors (which we will call Counterexamples 2 and 3 respectively). Specifically, Counterexample 2 is not to increment $seqno_{n,d}$ when a routing table entry is invalidated and Counterexample 3 is to delete (instead of invalidating) a routing table entry when its lifetime expires. The BMC framework was able to find these two errors too. 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 $n1$. In either Counterexample 2 or 3, if $n1$ is later offered a route to $n2$ by $n0$, this route will be accepted (because in Counterexample 2, $hops_{1,2} = \infty$; hence, $hops_{1,2} > hops_{0,2}$ whereas in Counterexample 3, $seqno_{0,2} > seqno_{1,2}$). A more detailed explanation (along with the traces) of counterexamples 2 and 3 can be found in [31].

Effect of the search strategy. Next, 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 counterexamples and witnesses. Table 1 gives the time and space overheads needed to find three counterexamples and one witness using several search strategies[3]. As shown in Table 1, AODV-BeFS-1 can reduce both the time and space overheads by several orders of magnitude when compared to BFS and DFS. Also, the choice of the best-first search strategy affects both the time and space overheads. For instance, as shown in Table 1 for Counterexample 1, AODV-BeFS-2 incurs two orders of magnitude increase in the time overhead and one order of magnitude increase in the space overhead, when compared to AODV-BeFS-1. The reason why AODV-BeFS-2 performs worse than AODV-BeFS-1 is that AODV-BeFS-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 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. In order to distinguish between equally good states, we studied the performance of the two-level best-first search strategy (which we called AODV-BeFS-3) that is inspired by both AODV-BeFS-1 and AODV-BeFS-2. As shown in Table 1, AODV-BeFS-1 and AODV-BeFS-3 significantly outperform the other best-first search strategies because they are more able to guide the best-first search towards the counterexample/witness even at the lower depths of the search space.

Performance evaluation of the BMC framework under more complex network scenarios. Next, we study the effect of the size and/or the topology of the network on the performance of the BMC framework with respect to the time and space overheads required for finding a counterexample and a witness. Table 2 gives the time and space overheads needed to find Counterexample 3 in a chain topology consisting of N nodes and a witness in a grid topology consisting of $M \times M$ nodes using AODV-BeFS-1. As shown in Table 2, the BMC framework in J-Sim was able to find counterexamples and witnesses for larger network topologies with reasonable time and space overheads.

[3] In this paper, N/A indicates that the model checker was not able to find a counterexample or a witness in 24 hours.

Table 1. Model checking AODV: Time (sec.) and space (number of states in the stateful search) overheads of finding a counterexample or a witness in a chain topology consisting of 3 nodes using different search strategies.

	Counterexample 1		Counterexample 2		Counterexample 3		Witness	
	Time	Space	Time	Space	Time	Space	Time	Space
BFS	3947.493	26013	5685.178	29083	3558.614	24966	0.61	552
DFS	5513.528	4758	49603.651	9068	5046.795	4637	4.047	240
AODV-BeFS-1	1.656	384	0.132	93	0.137	93	0.098	58
AODV-BeFS-2	234.374	1792	232.978	1786	231.961	1783	1.26	154
AODV-BeFS-3	0.916	299	0.156	97	0.159	97	0.107	58
AODV-BeFS-4	314.691	3299	N/A	N/A	311.067	3283	4.103	479
AODV-BeFS-5	171.646	2841	1389.929	5591	160.709	2774	3.372	499

Table 2. Model checking AODV: Time (sec.) and space (number of states in the stateful search) overheads of finding a counterexample and a witness for larger network topologies using AODV-BeFS-1.

Finding Counterexample 3 in a chain topology consisting of N nodes		
N	Time	Space
3	0.137	93
4	3.946	575
5	427.404	3817
6	582.150	3013
7	1803.807	3846

Finding a witness in a grid topology consisting of $M \times M$ nodes		
$M \times M$	Time	Space
2 × 2	0.334	140
3 × 3	12.375	2014
4 × 4	209.196	10112

4.2 Directed Diffusion in Wireless Sensor Networks (WSNs)

Overview of directed diffusion. Directed diffusion [16] is a *data-centric* information dissemination paradigm for WSNs. In directed diffusion, a *sink node* periodically broadcasts an INTEREST packet, containing the description of a sensing task that it is interested in knowing (e.g., detecting a vehicle in a specific area), to its neighbors. INTEREST messages are diffused throughout the network (e.g., via flooding), and are used to set up *exploratory gradients* within the network. A gradient is the direction state created in each node that receives an INTEREST packet where the gradient direction is set toward the neighboring node from which the INTEREST packet is received. Each node maintains an interest cache. Each interest entry in the interest 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 corresponding sink node. When a gradient expires, it is removed from its interest entry. When all gradients for 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 receives previously unseen data (i.e., the neighbor from whom a node 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* message 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 the directed diffusion protocol. In this section, we present the steps that we follow in order to model check the directed diffusion protocol. In order to illustrate how generic the BMC framework is, we follow the generic methodology given in Section 4.1.

Steps 1 and 2 are given in Appendix A. For Step 3, 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:

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

A desirable liveness property in directed diffusion is that there exists a sequence of events such that a sink node, n, that initiates a sensing task t, by broacasting an INTEREST packet containing the description of t, will eventually set up a reinforced path from a sensor node, $source(t)$, that can sense data which matches the interest, to n.

In model checking AODV, AODV-BeFS-1 and AODV-BeFS-3 provided comparatively better performance results in terms of both time and space. In Step 4, we use these two best-first search strategies to devise two corresponding best-first search strategies for directed diffusion. In particular, as the loop-free property involves only valid routing table entries 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 routing table entries, by analogy, forwarding of data packets in directed diffusion is based on the gradients established at the nodes. Therefore, two good best-first search 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.

Along a similar line of arguments, we also devise the following best-first search strategies:

1. DD-BeFS-3: Since the reinforced gradients are established upon the reception of positive reinforcement messages, another best-first search strategy is to consider a state s_1 better than a state s_2 if the number of positive reinforcement messages 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 on top of all the conditions, 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.
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 on top of all the conditions, 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.

Errors discovered. Next, we explain two previously unknown errors that the BMC framework in J-Sim was able to discover in directed diffusion (we will call these errors Counterexamples 1 and 2 respectively). We consider an initial state that consists of a chain topology of 4 nodes: $n0$ (the only sink node), $n1$, $n2$ and $n3$ (the only source node). The traces of Counterexample 1 (shown in Figure 2) and Counterexample 2 (shown in Figure 3) and their explanations are given in Appendix A. 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 have discovered 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 can be created. For instance, in the 4-node chain topology, if $n1$ accepts a DATA packet sent by $n2$, $n2$ becomes $n1$'s preferred neighbor. Now, if $n2$ 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 $n1$ (because it will be previously unseen data) causing $n1$ to become $n2$'s preferred neighbor. Therefore, $n1$ and $n2$ 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).

Table 3. Model checking Directed Diffusion: Time (sec.) and space (number of states in the stateful search) overheads of finding a counterexample or a witness in a chain topology consisting of 4 nodes using different search strategies.

	Counterexample 1		Counterexample 2		Witness	
	Time	Space	Time	Space	Time	Space
BFS	12900.183	21224	N/A	N/A	111.723	3441
DFS	7969.714	4736	N/A	N/A	N/A	N/A
DD-BeFS-1	1.355	200	1461.189	6026	5.411	446
DD-BeFS-2	1.574	200	3310.452	6640	14.669	493
DD-BeFS-3	7485.786	4630	N/A	N/A	N/A	N/A
DD-BeFS-4	0.489	124	216.995	1870	27.93	458
DD-BeFS-5	60605.76	16004	N/A	N/A	61.953	2187
DD-BeFS-6	15881.463	12920	N/A	N/A	0.238	100

Effect of the search strategy. Next, we study the performance of the various best-first search strategies with respect to the time and space overheads required for finding counterexamples and witnesses. Table 3 gives the time and space overheads needed to find two counterexamples and one witness with the use of several search strategies. As shown in Table 3, DD-BeFS-1 provides comparatively good results in terms of both time and space overheads for both disproving safety and proving liveness. 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 best-first search strategies that use DD-BeFS-1 if the non-visited states are equally good and are thus more able to guide the best-first search in the early stages of the search space than DD-BeFS-3 and DD-BeFS-5 respectively.

Performance evaluation of the BMC framework under more complex network scenarios. Table 4 gives the time and space overheads needed to find Counterexample 1 in a chain topology consisting of N nodes using DD-BeFS-4 and to find a witness in a grid topology consisting of $M \times M$ nodes using DD-BeFS-6. For sensor networks consisting of more than four nodes, both BFS and DFS failed to find counterexamples/witnesses. This shows that a best-first search strategy, which exploits protocol-specific heuristics, can find a counterexample/witness where other search strategies fail.

5 Conclusions and Future Work

This paper documents our research on extending the J-Sim network simulator with the capability of verifying network protocols using on the fly bounded model checking (BMC). We demonstrate the effectiveness of the model checker to model-check two widely used and fairly complex network protocols: AODV and directed diffusion. This shows that the BMC framework is generic enough and

Table 4. Model checking Directed Diffusion: Time (sec.) and space (number of states in the stateful search) overheads of finding a counterexample and a witness for larger network topologies.

Finding Counterexample 1 in a chain topology consisting of N nodes using DD-BeFS-4		
N	Time	Space
4	0.489	124
5	0.869	176
6	121.25	1017
7	7177.39	5476
8	43645.207	12314

Finding a witness in a grid topology consisting of $M \times M$ nodes using DD-BeFS-6		
$M \times M$	Time	Space
2 × 2	0.106	29
3 × 3	3.032	199
4 × 4	35145.625	5350

not tied to a particular network protocol. For model checking a network protocol, one needs to follow the generic methodology that we give in Sections 4.1-4.2.

Experimental results have shown that the model checker is able to find, within acceptable time, counterexamples of a safety property and witnesses for a liveness property. Note that the counterexample identified in directed diffusion is first discovered in this paper and previously unknown. Furthermore, we demonstrate that the use of best-first search strategies (that leverage *protocol-specific* properties) reduces the time and space overheads by several orders of magnitude. We study several best-first search strategies for both AODV and directed diffusion. Based on the results presented in this paper, we recommend deriving the best-first search strategy from the *safety or liveness property* being checked itself for routing protocols (such as AODV and directed diffusion). This is justified by the fact that AODV-BeFS-1 (and DD-BeFS-1) provided good performance results in terms of both time and space overheads for both disproving safety and proving liveness. Other best-first search strategies can still provide good performance results for either disproving safety or proving liveness. However, we recommend using a two-level best-first search strategy in which a best-first search strategy such as AODV-BeFS-1 (or DD-BeFS-1) is used if the nonvisited states are equally good. 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.

We believe we have made a case for incorporating model checking into simulation, so as to provide an integrated performance evaluation and verification environment. We have also identified several research avenues for future work. First, we are currently extending the J-Sim model checker to check general temporal properties. Second, we intend to compare the performance of exploiting protocol-specific heuristics against that of using protocol-independent properties in exploring the state space.

References

1. NS-2, "http://www.isi.edu/nsnam/ns/," .

2. J-Sim, "http://www.j-sim.org/," .
3. 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.
4. K. Havelund, "Java Pathfinder, a translator from Java to Promela," in *Proc. of SPIN'99*.
5. 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*.
6. T. Ball, and S. K. Rajamani, "The SLAM Toolkit," in *Proc. of CAV'01*.
7. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith, "Modular Verification of Software Components in C," in *Proc. of ICSE'03*.
8. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre, "Lazy Abstraction," in *Proc. of POPL'02*.
9. A. Farzan, F. Chen, J. Meseguer, and G. Rosu, "Formal analysis of Java programs in JavaFAN," in *Proc. of CAV'04*.
10. E. M. Clarke, O. Grumberg, and D. Peled, *Model Checking*, MIT Press, 1999.
11. A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," in *Proc. of TACAS'99*.
12. T. Schuele and K. Schneider, "Bounded model checking of infinite state systems: Exploiting the automata hierarchy," in *Proc. of ACM-IEEE MEMOCODE'04*.
13. 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*.
14. C. E. Perkins and E. M. Royer, "Ad-hoc on-demand distance vector routing," in *Proc. of IEEE WMCSA'99*.
15. C. E. Perkins, E. M. Royer, and S. Das, "Ad hoc on demand distance vector (aodv) routing," IETF Draft, January 2002.
16. C. Intanagonwiwat, R. Govindan, and D. Estrin, "Directed diffusion: A scalable and robust communication paradigm for sensor networks," in *Proc. of ACM MobiCom'00*.
17. M. Musuvathi and D. R. Engler, "Model checking large network protocol implementations," in *Proc. of NSDI'04*.
18. 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 (STTT)*, vol. 5, no. 2-3, pp. 247–267, March 2004.
19. 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.
20. G. J. Holzmann, "The model checker SPIN," *IEEE Trans. on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997.
21. K. McMillan, *Symbolic Model Checking*, Kluwer Academic Publishers, 1993.
22. 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*.
23. J. M. Wing and M. Vaziri-Farahani, "Model checking software systems: a case study," in *Proc. of ACM SIGSOFT'95*.
24. D. Lie, A. Chou, D. Engler, and D. Dill, "A simple method for extracting models from protocol code," in *Proc. of ISCA'01*.
25. 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*.
26. G. N. Naumovich, L. A. Clarke, and L. J. Osterweil, "Verification of communication protocols using data flow analysis," in *Proc. of ACM SIGSOFT'96*.

27. P. Godefroid, “Model checking for programming languages using VeriSoft,” in *Proc. of ACM POPL’97*.
28. W. Visser, K. Havelund, G. Brat, and S.Park, “Model checking programs,” in *Proc. of IEEE ASE’00*.
29. S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In *Proc. of WRLA’02*.
30. P. Olveczky, M. Keaton, J. Meseguer, C. Talcott, and S. Zabele, “Specification and analysis of the AER/NCA active network protocol suite in Real-Time Maude,” in *Proc. of FASE’01*.
31. 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.
32. 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.

Appendix A: Model Checking Details

In the appendix, we provide technical details of using the BMC framework to model-check the implementations of AODV (Section A.1) and directed diffusion (Section A.2). This includes a detailed description of Steps 1 and 2 of the generic model checking methodology and traces and explanations of selected counterexamples.

A.1 Details of Model Checking AODV

Step 1: Defining a global state and specifying the initial state(s). To model check 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 routing table, the broadcast ID cache that keeps the information of previously received RREQ packets, sequence number $seqno_n$ and broadcast ID bid_n. Each of $seqno_n$ and bid_n is a monotonically increasing counter. 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. If a destination node d sends a RREP packet (in response to a RREQ packet requesting a route to d), it first increments $seqno_d$ and then sends a RREP packet containing the new value of $seqno_d$. Therefore, $seqno_d$ is used to maintain the freshness of the route to d. 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 for 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 $bid_n = 1$.

Step 2: Specifying the state 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 one of the entries in the broadcast ID cache. The broadcast ID timeout event may generate multiple successor states depending on which entry in the cache is deleted.
   (d) Timeout of the 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.
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. This is an example of an event that may generate multiple successor states. This is because if the network contains two packets m_1 and m_2 whose destination is node n (or n is the next hop towards the destination) and n is one of the neighbors of the source of the packet, two successor states can be generated depending on the sequence in which m_1 and m_2 are received by node n.
   (b) Loss of an AODV packet destined to 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 one of these AODV packets from the network. Similar to the event of receiving an AODV packet, this event may generate multiple successor states depending on which packet in the network is lost.

Trace and explanation of Counterexample 1. The counterexample, shown in Figure 1, can be explained as follows. State 1 is the initial state. In state 2, $n0$ initiates a route request to destination $n2$ by broadcasting a RREQ packet. Similarly, in state 3, $n1$ initiates a route request to 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 for $n0$ 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 ∞ 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). 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 1, $nexthop_{0,2} = 1$ but $seqno_{0,2} > seqno_{1,2}$; i.e., a routing loop is created.

A.2 Details of Model Checking Directed Diffusion

Step 1: Defining a global state and specifying the initial state(s). To model check directed diffusion, we use the same definitions of *GlobalState* and network cloud that were introduced in Section A.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 each of the nodes starts with an empty interest cache and an empty data cache.

Step 2: Specifying the state transitions, when each transition occurs, and how each transition is handled. Similar to what we did in Section A.1, we will classify the state transitions into two categories: node transitions and network transitions. The transitions in each category are listed below:

1. Node Transitions
   (a) Initiation of a sensing task by node n: This event is enabled if the node n is a sink node. The event is handled by broadcasting an INTEREST packet to node n's neighbors.
   (b) 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. Specifically, the directed diffusion process at node n restarts with an empty interest cache and an empty data cache.
   (c) Gradient timeout event 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, and is handled by deleting one of the gradients. The gradient timeout event may generate multiple successor states depending on which gradient is deleted.
   (d) Data cache timeout event[4] at node n: This event is enabled if there is at least one entry in the data cache of node n, and is handled by deleting the entries in the data cache that have expired. The data cache timeout event may generate multiple successor states depending on which entries in the data cache are deleted.
2. Network Transitions
   (a) Delivering a packet to node n: This event is enabled if the network contains at least one packet that is destined for n such that 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. As mentioned above, this is an example of an event that may generate multiple successor states.
   (b) Loss of a packet destined to node n: This event is enabled if the network contains at least one packet that is destined for n. The event is handled by removing one of these packets from the network. Similar to the event of receiving a packet, this event may generate multiple successor states depending on which packet in the network is lost.

Trace and explanation of Counterexample 1. The counterexample, shown in Figure 2, can be explained as follows. State 1 is the initial state. In state 2, $n0$ initiaties a sensing task by broadcasting an INTEREST packet. In state 3, $n1$ receives the INTEREST packet sent by $n0$, sets up an exploratory gradient whose direction is set toward $n0$ (indicated by $previousHop = 0$ $datarate = 60.0$, where 60.0 seconds is the datarate of the exploratory gradient) and rebroadcasts the INTEREST packet. Specifically, after receiving the INTEREST packet, $n1$ decides to resend the INTEREST packet to its neighbors (i.e., $n0$ and $n2$). In state 4, $n2$ receives the INTEREST packet sent by $n1$, sets up an exploratory gradient whose direction is set toward $n1$ (because this is the node from which the INTEREST packet is received) and rebroadcasts the INTEREST packet. In state 5, $n1$ receives the INTEREST packet sent by $n2$ in state 4 and sets up an exploratory gradient whose direction is set toward $n2$. However, as mentioned in [16], a node may suppress a received INTEREST packet if it recently resent a matching INTEREST packet; therefore, $n1$ did not rebroadcast the INTEREST

[4] 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, the data cache is purged to delete stale entries.

packet because it recently resent a matching packet (in state 3). It should be noted that the mechanism of interest propagation and gradient establishment results in neighboring nodes establishing a gradient toward each other. This is because when a node receives an INTEREST packet from its neighbor, it has no way of knowing whether that INTEREST packet was in response to one that it sent out earlier or is an identical INTEREST packet from another sink on the other side of that neighbor [16].

In state 6, $n3$ receives the INTEREST packet sent by $n2$, sets up an exploratory gradient whose direction is set toward $n2$ and rebroadcasts the INTEREST packet. In addition, since $n3$ is located in the area specified by the INTEREST packet, it tasks its local sensors to begin collecting samples and sends a DATA packet to each neighbor for whom it has a gradient (in this case, $n3$ has a gradient to $n2$ only). In state 7, $n2$ receives the DATA packet sent by $n3$ in state 6 and, since the received DATA packet does not have a matching data cache entry, $n2$ adds the received DATA packet to the data cache (indicated by $source = 3$ in state 7) and resends the DATA packet to each neighbor for whom it has a gradient (in this case, $n2$ has a gradient to $n1$ only). In state 8, $n1$ receives the DATA packet sent by $n2$ in state 7 and, since the received DATA packet does not have a matching data cache entry, $n1$ adds the received DATA packet to the data cache (indicated by $source = 2$ in state 8) and resends the DATA packet to each neighbor for whom it has a gradient (in this case, $n1$ has a gradient to both $n0$ and $n2$). In state 9, $n0$ receives the DATA packet from $n1$. Since $n0$ has received this previously unseen event from $n1$, it positively reinforces $n1$ by sending a positive reinforcement packet (denoted as $POSREINFORCE$) to $n1$. In state 10, $n1$ receives the positive reinforcement packet sent by $n0$, and hence establishes a reinforced gradient whose direction is set toward $n0$ (indicated by $previousHop = 0$ $datarate = 5.0$ where 5.0 seconds is the datarate of the reinforced gradient). In turn, $n1$ should positively reinforce its preferred neighbor. Assuming that all nodes use the same data-driven local rule for determining the preferred neighbor, $n1$ will use its data cache to determine the neighbor from whom it first received the latest event matching the interest. In this case, $n1$'s preferred neighbor is $n2$; hence, $n1$ sends a positive reinforcement packet to $n2$.

In state 11, the data cache timeout event takes place at $n2$ causing the data packet received in state 7 to be deleted. In state 12, $n2$ receives the DATA packet sent by $n1$ in state 8 and, since the received DATA packet does not have a matching data cache entry, $n2$ adds the received DATA packet to the data cache (indicated by $source = 1$ in state 12) and resends the DATA packet to each neighbor for whom it has a gradient. In state 13, $n2$ receives the positive reinforcement packet sent by $n1$ in state 10, and hence establishes a reinforced gradient whose direction is set toward $n1$ (indicated by $previousHop = 1$ $datarate = 5.0$). In turn, $n2$ should positively reinforce its preferred neighbor. Using the same data-driven local rule for determining the preferred neighbor, $n2$ will use its data cache to determine the neighbor from whom it first received the latest event matching the interest. In this case, $n2$'s preferred neighbor is $n1$; hence, $n2$

sends a positive reinforcement packet to $n1$. In state 14, $n1$ receives the positive reinforcement packet sent by $n2$ and establishes a reinforced gradient whose direction is set toward $n2$ (indicated by $previousHop = 2$ $datarate = 5.0$). Note that, a loop is created in the reinforced path between $n1$ and $n2$; i.e., $(RPath(1,2) \wedge RPath(2,1))$ violating the loop-free safety property. Furthermore, the positive reinforcement packet did not reach the data source causing a disruption in the reinforced path (i.e., the reinforced path does not include the data source).

Trace and explanation of Counterexample 2. Using the same initial state used in Counterexample 1, we show that a node reboot may also create a loop in the reinforced path. In order not to obtain the same counterexample given in Figure 2, we require that the counterexample should contain at least one state that is generated due to a node reboot event. Furthermore, in order to show that the error may still take place even if the data cache timeout event does not occur (i.e., the data cache size is infinite), we disable the data cache timeout event. The counterexample, shown in Figure 3, can be explained as follows. State 1 is the initial state. In state 2, $n0$ initiaties a sensing task by broadcasting an INTEREST packet. In state 3, $n1$ receives the INTEREST packet sent by $n0$, sets up an exploratory gradient whose direction is set toward $n0$ and rebroadcasts the INTEREST packet. In state 4, $n2$ receives the INTEREST packet sent by $n1$, sets up an exploratory gradient whose direction is set toward $n1$ and rebroadcasts the INTEREST packet. In state 5, $n3$ receives the INTEREST packet sent by $n2$, sets up an exploratory gradient whose direction is set toward $n2$ and rebroadcasts the INTEREST packet. In addition, since $n3$ is located in the area specified by the INTEREST packet, it instruments its local sensors to begin collecting samples and sends a DATA packet to each neighbor for whom it has a gradient (in this case, $n3$ has a gradient to $n2$ only).

In state 6, $n2$ receives the INTEREST packet sent by $n3$ in state 5 and sets up an exploratory gradient whose direction is set toward $n3$. However, $n2$ did not rebroadcast the INTEREST packet because it recently resent a matching packet (in state 4). In state 7, $n0$ rebroadcasts the INTEREST packet. In state 8, $n1$ reboots causing all the entries in the interest cache to be deleted. In state 9, $n1$ receives the INTEREST packet sent by $n0$, sets up an exploratory gradient whose direction is set toward $n0$ and rebroadcasts the INTEREST packet. In state 10, $n1$ receives the INTEREST packet sent by $n2$ in state 4 and sets up an exploratory gradient whose direction is set toward $n2$. In state 11, $n2$ receives the DATA packet sent by $n3$ in state 5 and, since the received DATA packet does not have a matching data cache entry, $n2$ adds the received DATA packet to the data cache and resends the DATA packet to each neighbor for whom it has a gradient (in this case, $n2$ has a gradient to both $n1$ and $n3$). In state 12, $n0$ rebroadcasts the INTEREST packet.

In state 13, $n2$ reboots causing all the entries in the data and interest caches to be deleted. In state 14, $n2$ receives the INTEREST packet sent by $n1$ in state 9, sets up an exploratory gradient whose direction is set toward $n1$ and rebroadcasts the INTEREST packet. In state 15, $n1$ receives the DATA packet sent by $n2$ in state 11 and, since the received DATA packet does not have a matching data cache entry, $n1$ adds the received DATA packet to the data cache and resends the DATA packet to each neighbor for whom it has a gradient (in this case, $n1$ has a gradient to both $n0$ and $n2$). In state 16, $n2$ receives the DATA packet sent by $n1$ and, since the received DATA packet does not have a matching data cache entry, $n1$ adds the received DATA packet to the data cache and resends the DATA packet to each neighbor for whom it has a gradient (in this case, $n2$ has a gradient to $n1$ only). In state 17, $n0$ receives the DATA packet from $n1$. Since $n0$ has received this previously unseen event from $n1$, it positively reinforces $n1$ by sending a positive reinforcement packet to $n1$. In state 18, $n1$ receives the positive reinforcement packet sent by $n0$, and hence establishes a reinforced gradient whose direction is set toward $n0$. In turn, $n1$ should positively reinforce its preferred neighbor. Assuming that all nodes use the same data-driven local rule for determining the preferred neighbor, $n1$ will use its data cache to determine the neighbor from whom it first received the latest event matching the interest. In this case, $n1$'s preferred neighbor is $n2$; hence, $n1$ sends a positive reinforcement packet to $n2$. In state 19, $n2$ receives the positive reinforcement packet sent by $n1$, and hence establishes a reinforced gradient whose direction is set toward $n1$. In turn, $n2$ should positively reinforce its preferred neighbor. Using the same data-driven local rule for determining the preferred neighbor, $n2$ will use its data cache to determine the neighbor from whom it first received the latest event matching the interest. In this case, $n2$'s preferred neighbor is $n1$; hence, $n2$ sends a positive reinforcement packet to $n1$. In state 20, $n1$ receives the positive reinforcement packet sent by $n2$ and establishes a reinforced gradient whose direction is set toward $n2$. Note that, similar to counterexample 1, a loop is created in the reinforced path between $n1$ and $n2$; i.e., ($RPath(1,2) \wedge RPath(2,1)$). In addition, the positive reinforcement packet did not reach the data source causing a disruption in the reinforced path (i.e., the reinforced path does not include the data source).

```
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. 1. Model checking AODV: Trace of counterexample 1 discovered by the BMC framework in J-Sim using breadth-first search.

```
COUNTEREXAMPLE
State 1 Depth = 0
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: Empty; Data Cache: Empty
        Node 2 Interest Cache: Empty; Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network Empty
State 2 Depth = 1
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: Empty; Data Cache: Empty
        Node 2 Interest Cache: Empty; Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network INTEREST(src:0--dest:1)
State 3 Depth = 2
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0); Data Cache: Empty
        Node 2 Interest Cache: Empty; Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:2)
State 4 Depth = 3
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0); Data Cache: Empty
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:1) INTEREST(src:2--dest:3)
State 5 Depth = 4
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0) (previousHop=2 datarate=60.0);
               Data Cache: Empty
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:3)
State 6 Depth = 5
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0) (previousHop=2 datarate=60.0);
               Data Cache: Empty
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: Empty
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:3--dest:2)
State 7 Depth = 6
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0) (previousHop=2 datarate=60.0);
               Data Cache: Empty
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: source=3
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:2--dest:1)
State 8 Depth = 7
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0) (previousHop=2 datarate=60.0);
               Data Cache: source=2
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: source=3
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:1--dest:0)
                DATA(src:1--dest:2)
```

Fig. 2. Model checking Directed Diffusion: Trace of counterexample 1 discovered by the BMC framework in J-Sim using breadth-first search. MAX_DEPTH was set to 15.

```
State 9 Depth = 8
        Node 0 Interest Cache: Empty; Data Cache: source=1
        Node 1 Interest Cache: (previousHop=0 datarate=60.0) (previousHop=2 datarate=60.0);
               Data Cache: source=2
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: source=3
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:1--dest:2)
                POSREINFORCE(src:0--dest:1)
State 10 Depth = 9
        Node 0 Interest Cache: Empty; Data Cache: source=1
        Node 1 Interest Cache: (previousHop=0 datarate=5.0) (previousHop=2 datarate=60.0);
               Data Cache: source=2
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: source=3
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:1--dest:2)
                POSREINFORCE(src:1--dest:2)
State 11 Depth = 10
        Node 0 Interest Cache: Empty; Data Cache: source=1
        Node 1 Interest Cache: (previousHop=0 datarate=5.0) (previousHop=2 datarate=60.0);
               Data Cache: source=2
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: Empty
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:1--dest:2)
                POSREINFORCE(src:1--dest:2)
State 12 Depth = 11
        Node 0 Interest Cache: Empty; Data Cache: source=1
        Node 1 Interest Cache: (previousHop=0 datarate=5.0) (previousHop=2 datarate=60.0);
               Data Cache: source=2
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: source=1
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) POSREINFORCE(src:1--dest:2)
                DATA(src:2--dest:1)
State 13 Depth = 12
        Node 0 Interest Cache: Empty; Data Cache: source=1
        Node 1 Interest Cache: (previousHop=0 datarate=5.0) (previousHop=2 datarate=60.0);
               Data Cache: source=2
        Node 2 Interest Cache: (previousHop=1 datarate=5.0); Data Cache: source=1
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:2--dest:1)
                POSREINFORCE(src:2--dest:1)
State 14 Depth = 13
        Node 0 Interest Cache: Empty; Data Cache: source=1
        Node 1 Interest Cache: (previousHop=0 datarate=5.0) (previousHop=2 datarate=5.0);
               Data Cache: source=2
        Node 2 Interest Cache: (previousHop=1 datarate=5.0); Data Cache: source=1
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:3--dest:2) DATA(src:2--dest:1)
                POSREINFORCE(src:1--dest:2)
Total Time = 1.2900183E7 msecs.
```

Fig. 2. (cont'd) Model checking Directed Diffusion: Trace of counterexample 1 discovered by the BMC framework in J-Sim using breadth-first search. MAX_DEPTH was set to 15.

```
COUNTEREXAMPLE
State 1 Depth = 0
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: Empty; Data Cache: Empty
        Node 2 Interest Cache: Empty; Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network Empty
State 2 Depth = 1
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: Empty; Data Cache: Empty
        Node 2 Interest Cache: Empty; Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network INTEREST(src:0--dest:1)
State 3 Depth = 2
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0); Data Cache: Empty
        Node 2 Interest Cache: Empty; Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:2)
State 4 Depth = 3
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0); Data Cache: Empty
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: Empty
        Node 3 Interest Cache: Empty; Data Cache: Empty
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:1) INTEREST(src:2--dest:3)
State 5 Depth = 4
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0); Data Cache: Empty
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: Empty
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:1) INTEREST(src:3--dest:2) DATA(src:3--dest:2)
State 6 Depth = 5
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0); Data Cache: Empty
        Node 2 Interest Cache: (previousHop=1 datarate=60.0) (previousHop=3 datarate=60.0);
               Data Cache: Empty
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:1) DATA(src:3--dest:2)
State 7 Depth = 6
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0); Data Cache: Empty
        Node 2 Interest Cache: (previousHop=1 datarate=60.0) (previousHop=3 datarate=60.0);
               Data Cache: Empty
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:1) DATA(src:3--dest:2) INTEREST(src:0--dest:1)
State 8 Depth = 7
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: Empty; Data Cache: Empty
        Node 2 Interest Cache: (previousHop=1 datarate=60.0) (previousHop=3 datarate=60.0);
               Data Cache: Empty
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:1) DATA(src:3--dest:2) INTEREST(src:0--dest:1)
State 9 Depth = 8
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0); Data Cache: Empty
        Node 2 Interest Cache: (previousHop=1 datarate=60.0) (previousHop=3 datarate=60.0);
               Data Cache: Empty
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:2--dest:1) DATA(src:3--dest:2)
                INTEREST(src:1--dest:0) INTEREST(src:1--dest:2)
State 10 Depth = 9
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0) (previousHop=2 datarate=60.0);
               Data Cache: Empty
        Node 2 Interest Cache: (previousHop=1 datarate=60.0) (previousHop=3 datarate=60.0);
               Data Cache: Empty
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) DATA(src:3--dest:2) INTEREST(src:1--dest:0) INTEREST(src:1--dest:2)
State 11 Depth = 10
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0) (previousHop=2 datarate=60.0);
               Data Cache: Empty
        Node 2 Interest Cache: (previousHop=1 datarate=60.0) (previousHop=3 datarate=60.0);
               Data Cache: source=3
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) INTEREST(src:1--dest:2)
                DATA(src:2--dest:1) DATA(src:2--dest:3)
```

Fig. 3. Model checking Directed Diffusion: Trace of counterexample 2 discovered by the BMC framework in J-Sim using DD-BeFS-1. MAX_DEPTH was set to 20.

```
State 12 Depth = 11
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0) (previousHop=2 datarate=60.0);
               Data Cache: Empty
        Node 2 Interest Cache: (previousHop=1 datarate=60.0) (previousHop=3 datarate=60.0);
               Data Cache: source=3
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) INTEREST(src:1--dest:2)
                DATA(src:2--dest:1) DATA(src:2--dest:3) INTEREST(src:0--dest:1)
State 13 Depth = 12
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0) (previousHop=2 datarate=60.0);
               Data Cache: Empty
        Node 2 Interest Cache: Empty; Data Cache: Empty
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) INTEREST(src:1--dest:2)
                DATA(src:2--dest:1) DATA(src:2--dest:3) INTEREST(src:0--dest:1)
State 14 Depth = 13
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0) (previousHop=2 datarate=60.0);
               Data Cache: Empty
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: Empty
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) DATA(src:2--dest:1)
                DATA(src:2--dest:3) INTEREST(src:0--dest:1) INTEREST(src:2--dest:1) INTEREST(src:2--dest:3)
State 15 Depth = 14
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0) (previousHop=2 datarate=60.0);
               Data Cache: source=2
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: Empty
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) DATA(src:2--dest:3) INTEREST(src:0--dest:1)
                INTEREST(src:2--dest:1) INTEREST(src:2--dest:3) DATA(src:1--dest:0) DATA(src:1--dest:2)
State 16 Depth = 15
        Node 0 Interest Cache: Empty; Data Cache: Empty
        Node 1 Interest Cache: (previousHop=0 datarate=60.0) (previousHop=2 datarate=60.0);
               Data Cache: source=2
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: source=1
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) DATA(src:2--dest:3) INTEREST(src:0--dest:1)
                INTEREST(src:2--dest:1) INTEREST(src:2--dest:3) DATA(src:1--dest:0) DATA(src:2--dest:1)
State 17 Depth = 16
        Node 0 Interest Cache: Empty; Data Cache: source=1
        Node 1 Interest Cache: (previousHop=0 datarate=60.0) (previousHop=2 datarate=60.0);
               Data Cache: source=2
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: source=1
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) DATA(src:2--dest:3) INTEREST(src:0--dest:1)
                INTEREST(src:2--dest:1) INTEREST(src:2--dest:3) DATA(src:2--dest:1) POSREINFORCE(src:0--dest:1)
State 18 Depth = 17
        Node 0 Interest Cache: Empty; Data Cache: source=1
        Node 1 Interest Cache: (previousHop=0 datarate=5.0) (previousHop=2 datarate=60.0);
               Data Cache: source=2
        Node 2 Interest Cache: (previousHop=1 datarate=60.0); Data Cache: source=1
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) DATA(src:2--dest:3) INTEREST(src:0--dest:1)
                INTEREST(src:2--dest:1) INTEREST(src:2--dest:3) DATA(src:2--dest:1) POSREINFORCE(src:1--dest:2)
State 19 Depth = 18
        Node 0 Interest Cache: Empty; Data Cache: source=1
        Node 1 Interest Cache: (previousHop=0 datarate=5.0) (previousHop=2 datarate=60.0);
               Data Cache: source=2
        Node 2 Interest Cache: (previousHop=1 datarate=5.0); Data Cache: source=1
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) DATA(src:2--dest:3) INTEREST(src:0--dest:1)
                INTEREST(src:2--dest:1) INTEREST(src:2--dest:3) DATA(src:2--dest:1) POSREINFORCE(src:2--dest:1)
State 20 Depth = 19
        Node 0 Interest Cache: Empty; Data Cache: source=1
        Node 1 Interest Cache: (previousHop=0 datarate=5.0) (previousHop=2 datarate=5.0);
               Data Cache: source=2
        Node 2 Interest Cache: (previousHop=1 datarate=5.0); Data Cache: source=1
        Node 3 Interest Cache: (previousHop=2 datarate=60.0); Data Cache: source=3
        Network INTEREST(src:1--dest:0) INTEREST(src:1--dest:0) DATA(src:2--dest:3) INTEREST(src:0--dest:1)
                INTEREST(src:2--dest:1) INTEREST(src:2--dest:3) DATA(src:2--dest:1) POSREINFORCE(src:1--dest:2)
Total Time = 1461189.0 msecs.
```

Fig. 3. (cont'd) Model checking Directed Diffusion: Trace of counterexample 2 discovered by the BMC framework in J-Sim using DD-BeFS-1. MAX_DEPTH was set to 20.