Verification of Simulation Models of Network Protocols Using State Space Exploration and Protocol-Specific Properties

Ahmed Sobeih, Marcelo d'Amorim, Mahesh Viswanathan, Darko Marinov and Jennifer C. Hou

Department of Computer Science
University of Illinois at Urbana Champaign
201 North Goodwin Avenue
Urbana, IL 61801
{*sobeih, damorim, vmahesh, marinov, jhou*}*@cs.uiuc.edu*

Abstract

Verification and Validation (V&V) is a critically important phase in the development life cycle of a simulation model. In the context of network simulation, traditional network simulators perform well in using a simulation model for evaluating/predicting the performance of a network protocol but lack the capability of verifying the "correctness" of the simulation model being used. To address this problem, we have extended J-Sim — an open-source component-based network simulator written entirely in Java — with a state space exploration capability that explores the (entire) state space created by a network simulation model in order to find an execution (if any) that violates an assertion; i.e., a safety property.

In this paper, we elaborate on the state space exploration framework in J-Sim and demonstrate its usefulness and effectiveness in verifying complicated simulation models. Specifically, we verify the simulation models of two widely used and fairly complex network protocols: the *Ad-Hoc On-Demand Distance Vector (AODV)* routing protocol for wireless ad hoc networks and the *directed diffusion* data dissemination protocol for wireless sensor networks. To enable the verification of these fairly complex network simulation models, we make use of structural properties in the underlying state space along two orthogonal dimensions; the first uses a non-trivial *simulation relation* to prune the states to be searched, and the second is *state ranking* that determines whether a state is "better than" another in order to enable the implementation of a best-first search (BeFS). We also develop protocol-specific search heuristics to guide state space exploration towards finding assertion violations in less time. In particular, we report findings on how to devise *good* search heuristics for routing/data dissemination protocols similar to AODV and directed diffusion. We also show that the time needed to find an assertion violation by our state space exploration framework in J-Sim is comparable to that of Java PathFinder (JPF), a state-of-the-art model checker for Java programs.[1]

1 Introduction

Verification and Validation (V&V) is an important and integral part of the development of simulation models. A model is, by definition, an abstract representation of a real (existing or proposed) system built for the purpose of studying the system according to certain objectives. In particular, the *conceptual model* is a mathematical/logical/verbal representation of the system. In order to conduct simulation experiments on a model of a real-world system, the model is usually implemented in a simu-

[1]Preliminary versions of this paper appeared in [59, 62]. This paper contains more detailed explanations, more search techniques, more experiments, and the comparison with JPF.

lation software on a computer; i.e., turning the conceptual model into a *computerized model*. Validation addresses the question of "Are we building the right model?" and verification addresses the question of "Are we building the model right?". The process of building, verifying and validating a simulation model is iterative and may have to be repeated several times until a certain level of credibility in the model (and the information derived from it) is achieved.

Some of the V&V methods are informal subjective comparisons between the output of the (computerized) model and the output of the real system; some are formal objective procedures based on statistical techniques such as hypothesis testing or confidence intervals. Examples of V&V techniques are: (1) face validity, (2) checking the model assumptions (both structural assumptions of how the system operates and data assumptions of the input data), (3) sensitivity analysis, (4) driving the simulation model with historical data or samples from distributions and comparing the model input-output transformations to corresponding input-output transformations for the real system, (5) Turing tests, (6) using a simulation trace, (7) fault/failure insertion testing, (8) extreme input testing, and (9) static and dynamic testing of the computerized model. See [2–6, 39] for several verification and validation techniques.

As indicated by Sargent in [55], there is no set of specific tests that can be easily applied to determine the "correctness" of a simulation model and that no algorithm exists to determine what techniques or procedures to use. In this paper, we elaborate on a technique that is complementary to existing techniques and can be used in combination with them. Specifically, we present the design, implementation and evaluation of a framework that aids in the verification of a network simulation model by directly executing the model along several execution paths and checking whether the model satisfies certain assertions (i.e., safety properties) that the real system satisfies. As pointed out by Balci et al. in [6], computer-aided support for verification, validation, and accreditation (VV&A) is one of the strategic directions to achieve VV&A. Our framework facilitates this computer-aided support by providing the assertion checking V&V technique [5] *along several execution paths in the state space*.

The motivation of our work is that network simulators have been used for decades to build a model of a network protocol and evaluate its performance under different scenarios. One major deficiency of traditional network simulators, however, is that they only evaluate/predict the performance of network protocols in the scenarios provided by the user (e.g., a network protocol designer or a simulation modeler) but can *not* exhaustively analyze all possible scenarios for correctness. For example, a network simulator can use a simulation model of a routing protocol to conduct experiments and evaluate the performance of this protocol, but cannot check whether the simulation model being used may suffer from routing loops. If the assertion violations in the simulation model do not appear (and hence cannot be investigated) in the scenarios studied, they may not be identified as early as possible. If an assertion violation exists in the simulation model, the results obtained from the simulation experiments may be incorrect and may lead to wrong decision making. Furthermore, if the simulation model models a nonexisting system, undiscovered assertion violations in the simulation model may eventually manifest themselves only after the system has been implemented and deployed. In the light of recent research [54] that creates a physical implementation of a network protocol from existing simulation code without much modification, the consequence of leaving assertion violations undiscovered in the simulation code is especially severe. Therefore, building an integrated environment that allows the user to *both* verify a simulation model *and* use it to evaluate/predict performance is an important task.

Motivated thus, we have extended J-Sim [34]—an open-source component-based network simulation environment written entirely in Java—with the state space exploration capability that explores the (entire) state space created by a network simulation model, up to a configurable maximum depth in order to find violations of a safety property (e.g., the absence of routing loops) if any exists. Specifically, we have implemented a state space explorer (written in Java for seamless integration with J-Sim), and incorporated it into J-Sim. The basic idea is to execute the simulation model along several execution paths in order to pinpoint assertion violations. However, the challenge is how to enable the state space explorer to take control of the network simulation model to explore the (entire) state space, rather than simply exploring one single execution path (i.e., sequence of events) as J-Sim traditionally does. Moreover, this has to be done *without* requiring the core design and implementation of J-Sim to be altered and *without* degrading the execution time of the J-Sim simulation model if the user is only interested in using the model for performance evaluation purposes. In Section 3, we explain how we have overcome that challenge.

After the state space exploration framework is laid, we demonstrate the usefulness, effectiveness, and generality of our framework in verifying the simulation models of two widely used and fairly complex network protocols: the *Ad-Hoc On-Demand Distance Vector (AODV)* routing protocol [50, 51] for wireless ad hoc networks and the *directed diffusion* data dissemination protocol [33] for wireless sensor networks. These are reasonably complex network protocols whose J-Sim simulation models (not including the J-Sim library) have about 1200 and 1400 lines of code, respectively. (As a proof-of-concept case study, we verified the simulation model of an *automatic repeat request (ARQ)* protocol, also known as the *alternating bit* protocol, in [59].) Our choice of AODV and directed diffusion in this paper is motivated by their potential to become representative routing and data dissemination protocols, respectively, in ad hoc networks and sensor networks. We investigate whether these simulation models satisfy the *loop-free* safety property, i.e., data packets are not routed through loops.

To analyze such large simulation models of real network protocols, we make use of algorithms and heuristics that exploit structural properties salient in the state space of message passing systems in general, and some network protocols in particular. The first technique exploits the existence of a non-trivial *simulation relation* (in the process algebraic sense) between states to reduce the search space. The idea is as follows. A traditional state space exploration tool performs a classical search algorithm (like depth-first or breadth-first search) on the directed graph of states and transitions defined by the protocol, choosing to terminate certain branches of the search when it visits a state previously encountered. We observe that this basic algorithm is sound even when branches of the search are terminated whenever a state s is visited that can be *simulated by* another state s' that has been explored before, not just when the *same* state is re-encountered, i.e., whenever there is some previously visited state s' that can exhibit every behavior of s; the formal definition of simulation is deferred to technical sections. This can drastically reduce the search space, if the simulation relation is good. Next, we observe that, when the communication between network entities is unreliable (i.e., when message delivery is neither guaranteed nor ordered), which is typically the case for protocols for wireless networks, there is a very simple and natural simulation relation. Note that the state space of such protocols will consist of states comprising of a protocol state (encoding information needed in modeling the specific protocol) and an unordered collection of messages that have been sent but not delivered. For states s and s' with identical protocol states, and with the additional property that every undelivered message in s is also undelivered in s' (s' could have more undelivered messages), we observe that s' simulates s. We exploit this simulation relation in the examples.

Our second technique, called *state ranking*, is used to direct the state exploration more effectively towards finding assertion violations quickly. State ranking, as the name suggests, orders states based on which state is more "likely" to have an incorrect execution starting from it. The state exploration tool then uses a *best-first search (BeFS)* algorithm that biases the search towards states with higher ranks; more specifically, the algorithm after visiting a state, always picks successor states to visit in the order of their ranks. To obtain a ranking of states, we exploit *properties inherent to the network protocol and the safety property being checked.* However, unlike the simulation relations that we exploit, the state ranking that we present for the examples is not provably correct. In other words, we cannot statically prove that states with higher ranks in our schemes have more incorrect executions. Best-first search, though a heuristic, does not affect the soundness of our search engine (since it only changes the order of the search) and can find assertion violations significantly faster, as our experiments demonstrate. One interesting and important research question is how to determine a suitable BeFS heuristic for a specific network protocol. We make an attempt towards answering this question by studying the performance of several BeFS heuristics for both AODV and directed diffusion, and providing design guidelines based on our results.

Finally, we evaluate the efficiency of our state space exploration framework by comparing its performance to that of a state-of-the-art model checker for Java programs, namely Java PathFinder (JPF) [35, 72]. The results of the comparison show that our framework is comparable to JPF in terms of the time needed to find an assertion violation.

The rest of the paper is organized as follows. In Section 2, we give a brief overview of network simulation in J-Sim. In Section 3, we elaborate on the state space exploration framework that we implemented in J-Sim. Section 4 presents the performance results. In Section 5, we discuss related work. Finally, we conclude the paper in Section 6.

2 Network Simulation in J-Sim

As indicated by Balci et al. in [6], developing modeling and simulation applications using the component-based technology [15] is one of the strategic directions to achieve VV&A. J-Sim [34, 70] is an open-source network simulation and emulation environment that is developed entirely in Java on top of a component-based software architecture, called the *autonomous component architecture (ACA)* [70], that closely mimics the integrated circuit (IC) design.

The basic entities in the ACA are *components*, which communicate with one another via sending/receiving data at their *ports*. When data arrives at a port of a component, the component processes the data immediately in an independent execution context (e.g., *thread* in Java). Following the same line of design principles of IC chips, how components in the ACA behave (in terms of how a component handles and responds to data that arrives at a port) is specified at the system design time in *contracts*, but component binding does not take place until the system integration time when the system is being "composed." A contract specifies how an initiator (caller) and a reactor (callee) fulfill a certain function; i.e., the causality of information exchange between components but *not* the components that may participate in information exchange. Two components, acting respectively as the initiator and the reactor, are bound at the system integration time to fulfill the contract. In some sense, the ACA realizes the notion of *software IC* [9] where an IC corresponds to a component, pins correspond to ports, signals correspond to data that arrives at a port of a component, and an IC specification corresponds to a contract. Similar to composing an electronic system (e.g., ALU) by interconnecting ICs via pins, building a simulation model of a network protocol in J-Sim requires designing and implementing a set of components (e.g., senders, receivers, routers, links, and protocols that run within each router/host) and interconnecting them via ports. On top of the ACA, a generalized packet-switched internetworking framework (called *INET*) has been laid based on common features extracted from the various layers in the network protocol stack. Both the ACA and the INET have been implemented in Java, and the resulting code, along with its scripting framework and GUI interfaces, is called J-Sim.

J-Sim possesses several other desirable features (e.g., composability and extensibility [71]). The fact that J-Sim is implemented in Java, along with its ACA, makes J-Sim a truly platform-independent simulation environment. J-Sim provides a script interface that allows its integration with different script languages such as Perl, Tcl, or Python. In particular, the latest release of J-Sim (version 1.3) has been fully integrated with a Java implementation of Tcl interpreter, called Jacl, with the Tcl/Java extension. Therefore, similar to ns-2 (ns version 2) [67], J-Sim is a dual-language simulation environment in which classes are written in Java (for ns-2, in C++) and "wired" together using Tcl/Java[2].

For all the reasons mentioned above, we believe that J-Sim is a promising candidate to be extended with the state space exploration capability. However, we believe that our technique can be implemented in other network simulators too. Furthermore, although we propose our technique in the context of simulation of computer networks, the idea itself is generic enough and can be applied to other application domains of simulation.

3 State Space Exploration Framework in J-Sim

In this section, we elaborate on the state space exploration framework that we implemented in J-Sim.

First, we review some basic concepts of discrete-event system simulation [8] and explicit-state model checking [16] by state space exploration. A *system* is a group of objects that are joined together in some regular interaction or interdependence toward the accomplishment of some purpose (e.g., a wireless network). A *simulation model* is a representation of the system for the purpose of studying the behavior of the system with respect to certain performance evaluation criteria (e.g., network throughput). An *entity* is an object of interest in the system (e.g., a wireless node). An entity is described or characterized by its *attributes* (e.g., routing table, MAC address, etc.). Each attribute has a type, which may be either simple (e.g., boolean, integer, etc.) or composite (e.g., an array of integers, an object, an array of objects, etc.). The type of an attribute indicates the domain over which the attribute ranges. The *state* of the system is a complete description of the system and includes values of all attributes of entities that are relevant to the objective of the study (e.g., the routing table entries of all the wireless nodes and the number/contents of the messages being sent over the wireless channel). A state s can be regarded as a function that assigns to each attribute a value over its domain. The *state space*, denoted by S, is the set of all possible system states. It should be noted that although the set of attributes is finite, S may be infinite because the domains over which the attributes range may be infinite; this phenomenon is known as the *state space explosion problem*. An *event* is an instantaneous occurrence that might change the state of the system (e.g., message sending and message receiving); i.e., assign new values to (some of) the attributes. An event may be either conditional or unconditional. An *unconditional event* is an event that can always occur (e.g., an unpredictable crash of an active wireless node). A *conditional event* is an event that can occur only if a certain *enabling condition* is true (e.g., a wireless node can receive a message only if another node has sent a message to it). A *safety property*, also called an *assertion*, is a property that must always hold true in all states (e.g., the absence of routing loops in a routing table). The meaning of the safety property depends on the network protocol itself. For example, if the protocol is a reliable unicast/multicast protocol, the safety property may be that the receiver(s) receive all the packets that the sender believes to have been received. In the case of a security protocol, the safety property may be that unauthorized users do not get access to the system. In our framework, the user specifies the safety property as a Java method whose output is true/false.

While building the state space exploration framework in J-Sim, we had two major design goals in mind:

[2]The term "wire" is the term that J-Sim uses when displaying a connection between two ports.

1. The core design and implementation of J-Sim must not be modified.

2. Any modifications to the J-Sim simulation model of the network protocol must be minimal and must not degrade the execution time of the J-Sim simulation model if a user is only interested in using the model for performance evaluation purposes.

We believe that these two design goals should be also kept in mind if one desires to build a state space exploration framework in any other (network) simulator.

Towards realizing the above design goals, we implement an explicit-state stateful state space explorer in Java as a component in the ACA of J-Sim. The state space explorer executes the J-Sim simulation model of the network protocol *directly* and explores the state space on-the-fly. Specifically, the state space explorer starts from an initial state and generates successor states by executing the events of the simulation model. This process continues until either a counterexample disproving the safety property is found or the state space is explored up to a configurable maximum depth (MAX_DEPTH). In the former case, the state space explorer outputs a counterexample; i.e., the sequence of events that leads to violating the safety property. In the latter case, the state space explorer reports a message stating that "No assertion violation was found"; this does not mean that the simulation model does not violate the assertion and further testing is not required. A violation of the safety property may exist at depths larger than MAX_DEPTH. Furthermore, the state space explorer may run out of memory (or take an excessively long amount of time) before exploring the state space up to MAX_DEPTH. Hence, the goal of our framework is *not* to prove that the simulation model satisfies the safety properties. Instead, the goal is to find violations of those safety properties if any exists given a certain budget of time and memory constraints.

3.1 The verification model

For the purpose of performance evaluation, a simulation model contains only components of the real system that are relevant to measuring performance. Similarly, for the purpose of verification using state space exploration, a *verification model* contains only components of the simulation model that are relevant to state space exploration. In other words, the simulation model is an abstraction of the real system whereas the verification model is an abstraction of the simulation model. In particular, the attributes that are not relevant to checking safety properties should not be part of the verification model. Hence, the definition of the verification model may change if the safety property that is to be checked changes. This is similar to changing the simulation model if the objectives of a performance evaluation study change. Note that the verification model is *not* needed for measuring performance; it is only needed for checking whether or not the simulation model satisfies the safety properties. In contrast, the simulation model *is* needed for state space exploration because the events are executed *inside* the simulation model. This will be explained in more detail in Section 3.2.

Figure 1 illustrates the overall framework of incorporating state space exploration into J-Sim. As shown in Figure 1, the state space explorer interacts via ports with the protocol entities P_1, P_2, ... , P_n, which are instances of the Java classes that implement the simulation model of the network protocol. In order to explore the state space created by the simulation model, the notion of the "state" has to be adequately defined in the verification model. To this end, the state space explorer makes use of a class called *GlobalState*. The *GlobalState* class includes (some of) the attributes of the entities as data members. In the verification model, a state is an instance of the *GlobalState* class that assigns to each data member (i.e., attribute) a value. The implementation of *GlobalState* differs from one network protocol to another; hence, it is the responsibility of the user to provide an implementation of *GlobalState*. In

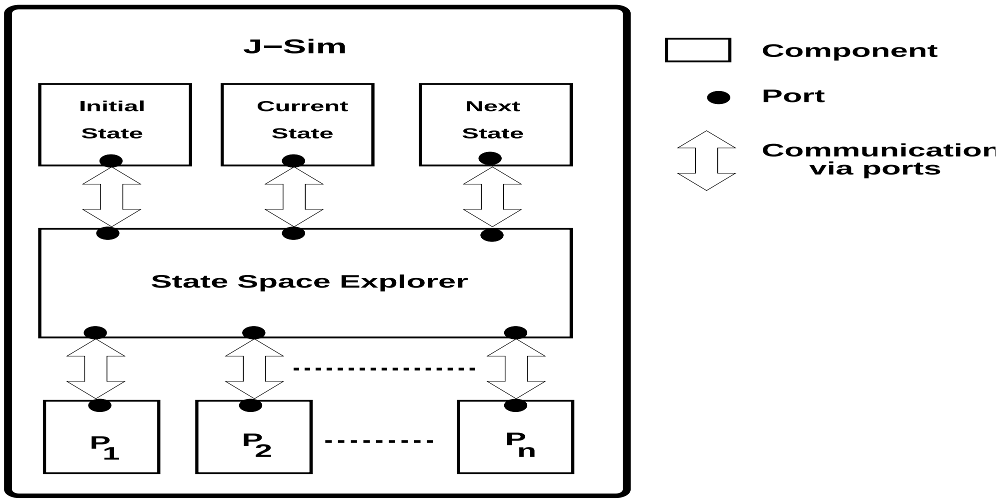


Figure 1: Overall framework of state space exploration in J-Sim. The protocol entities P_1, P_2, ... , P_n constitute the simulation model whereas the initial state, current state and next state constitute the verification model.

addition, the user should also construct an initial state in order for state space exploration to get started. As shown in Figure 1, the state space explorer interacts via ports with three instances of *GlobalState*, namely *initialState* (the initial state), *currentState* (the current state being explored) and *nextState* (one of the possible successors of the current state). The contract needed to define the information exchange between the state space explorer and the three instances of *GlobalState* is implemented in the *ModelCheckingGlobalStateContract* class, which is a subclass of *Contract*, a key J-Sim class that defines a contract. As mentioned in Section 2, a contract specifies the causality of information exchange between components but not the components that may participate in the information exchange. At the system design time, neither the initiator nor the reactor knows the identity of the other. The connection between the initiator and the reactor (which is shown using double-arrows in Figure 1) takes place only at the system integration time when the two components are bound to fulfill the contract. This ensures a loosely-coupled component architecture. Similarly, the *ModelCheckingProtocolEntityContract* class implements the contract needed to define the information exchange between the state space explorer and the protocol entities. Making the state space explorer take control of the simulation model in order to explore the (entire) state space, rather than just exploring one single execution path as J-Sim traditionally does, is achieved by having the state space explorer be the initiator of the *ModelCheckingProtocolEntityContract* (*ModelCheckingGlobalStateContract*) contract, and having a protocol entity (global state) be the reactor respectively.

In each state in the state space, some events may or may not be enabled. Examples of events that are common among various network protocols are: packet reception, packet loss and timeout. Enabled events are parameterized; hence, an enabled event may generate multiple successor states depending on the values of its parameters. For instance, a packet reception event may generate multiple successor states because if the network contains K packets $m_1, m_2, \ldots, m_K$ whose destination is node n and the network does not guarantee ordered packet delivery, K successor states can be generated depending on which of the K packets is to be received by node n. On the other hand, a node reboot event may generate only one successor state (namely the state of the node after reboot). In the current implementation, the parameters of the events are limited to integer values. For example, in the packet reception event described above, the index i ($0 \leq i \leq K-1$) of a packet in the network is used as a parameter to determine which packet is to be received. It is important to note that the number of possible successor states that an enabled event can generate is state-dependent. In particular, an event may be enabled in one state but disabled in another. It is the responsibility of the user to specify (a) the set of events that can occur, (b) a function that dynamically determines the state-dependent number of possible successor states that each event generates (zero if the event is disabled), and (c) how each event is handled (i.e., an event handler function that makes a state change). Note that the event handlers are part of the simulation model; i.e., the user has to write the event handlers anyway in order to have a working simulation model of the network protocol in J-Sim, even if he/she does not intend to make use of the state space exploration framework for the purpose of verification.

3.2 The state space exploration process

Figure 2 shows the pseudo-code of the state space exploration procedure SSExploreAddCurrent(), which is one of the main procedures in the state space explorer (Figure 1). The two major data structures in SSExploreAddCurrent() are *NonVisitedStates* (which stores the states that have been encountered but not yet explored) and *AlreadyVisitedStates* (which stores the states that have already been explored). Now a traditional explicit-state model checker would avoid (re-)exploring a state s_1 if the *same, identical* state were previously explored, i.e., were in the data structure *AlreadyVisitedStates.* Figure 2, on the other hand, presents a modified explicit-state stateful search that avoids visiting a state s_1 if another state s_2 has already been visited before, which (provably) *simulates* it. Informally, s_1 is simulated by s_2 (or s_2 simulates s_1) if the states that can be explored from s_1 are also simulated by those that can be explored from s_2; hence, there is no need to explore states from s_1.[3] *AlreadyVisitedStates* stores concrete states, and a state s_1 is said to be simulated by another state s_2 if $s_1.isSimulatedBy(s_2)$ returns true. When the communication between network entities is unreliable (i.e., when message delivery is neither guaranteed nor ordered), which is typically the case for protocols for wireless networks, there is a very simple and natural simulation relation. Note that the state space of such protocols will consist of states comprising of a protocol state (encoding information needed in modeling the specific protocol) and an unordered collection of messages that have been sent but not delivered. For states s and s' with identical protocol states, and with the additional property that every undelivered message in s is also undelivered in s' (s' could have more undelivered messages), we observe that s' simulates s. For the examples in this paper we exploit this simulation relation. However, in general, a user could potentially exploit specific details of the protocol to reveal a stronger simulation relation (and hence a faster exploration of a larger state space). Hence, we leave the implementation of *isSimulatedBy*() to the user.

SSExploreAddCurrent() starts with an empty *AlreadyVisitedStates* (Figure 2, line 2). *NonVisitedStates* initially contains the initial state only (Figure 2, line 3). As long as *NonVisitedStates* is not empty (Figure 2, line 4), SSExploreAddCurrent() removes a state from *NonVisitedStates* and sets *currentState* to it (Figure 2, line 5). This operation is an example of information exchange that makes use of the *ModelCheckingGlobalStateContract* contract. Specifically, the state space explorer is the initiator and *currentState* is the reactor. The state space explorer sends *currentState* the state s that has been

[3]Formally, a simulation relation is a binary relation R over the set of states S (i.e., $R \subseteq S \times S$) such that for every pair of states $s_1, s_2 \in S$, if $(s_1, s_2) \in R$ then for all events e that are enabled in s_1, all parameters i of these events, and all states $s'_1 \in S$, $s_1 \xrightarrow{e(i)} s'_1$ implies that there exists an event e' that is enabled in s_2, a parameter i' of this event, and a state $s'_2 \in S$ such that $s_2 \xrightarrow{e'(i')} s'_2$, and $(s'_1, s'_2) \in R$. In this case, we say that s_2 simulates s_1 (or s_1 is simulated by s_2). Note that the similarity relation is a preorder; hence, it is reflexive and transitive.

removed from *NonVisitedStates*. *currentState* responds by setting its attributes to the values assigned to them in the state s. SSExploreAddCurrent() then explores *currentState* only if a state simulating it has not been visited before (Figure 2, line 6). For each state being explored (*currentState*), SSExploreAddCurrent() first adds it to *AlreadyVisitedStates* (Figure 2, line 7) and then determines the events that are enabled in *currentState* by invoking *GenerateEnabledEvents()* (Figure 2, line 8). In *GenerateEnabledEvents()*, the enabling function (Figure 2, line 24) returns the number of possible successor states for each event (zero if the event is disabled). This operation is another example of an information exchange that makes use of the *ModelCheckingGlobalStateContract* contract because calculating the number of possible successor states is done inside the verification model. *GenerateEnabledEvents()* returns *EnabledEvents*, which is a list of enabled events (Figure 2, line 27). Each entry in *EnabledEvents* stores the corresponding event information *EventInfo* (Figure 2, line 26). Specifically, let each protocol entity have a unique ID p (Figure 2, line 22), each event have a unique ID e (Figure 2, line 23), and each enabled event has a set of integer-valued parameters i where $0 \leq i \leq NumberOfNextStates - 1$ (Figure 2, line 25), then each instance of *EventInfo* stores p, e and i.

SSExploreAddCurrent() then generates the successor states (*nextState*) by calling the *GenerateNextState()* function (Figure 2, line 11) for each enabled event, which in turn invokes the corresponding event handler (Figure 2, line 31). Note that an event handler is only invoked from the state space explorer but actually executed *inside* the simulation model; namely the protocol entities themselves. Therefore, SSExploreAddCurrent() must first set the state of the protocol entities to the state reflected in *currentState* before the execution of the event handler. This is achieved by the *CopyFromVModelToSModel()* function call (Figure 2, line 28). *CopyFromVModelToSModel()* is an example of an operation that makes use of both the *ModelCheckingGlobalStateContract* and the *ModelCheckingProtocolEntityContract* contracts. Specifically, the state space explorer uses *ModelCheckingGlobalStateContract* to query *currentState*, and *currentState* responds with the state of the protocol entities reflected in it. Following that, the state space explorer uses *ModelCheckingProtocolEntityContract* to instruct the protocol entities to set their state to the state reflected in *currentState*. Executing an event handler (Figure 2, line 31) is an example of an operation that makes use of the *ModelCheckingProtocolEntityContract* contract, where the state space explorer instructs a protocol entity to execute an event. After the execution of the event handler, the *CopyFromSModelToVModel()* function is called (Figure 2, line 32) to perform the reverse operation; i.e., extract the new state information from the protocol entities and copy them to *nextState*. *CopyFromSModelToVModel()* is another example of an operation that makes use of both the *ModelCheckingProtocolEntityContract* and the *ModelCheckingGlobalStateContract* contracts.

SSExploreAddCurrent() then checks whether *nextState* violates a safety property (Figure 2, line 12). Our state space exploration framework in J-Sim also allows the user to specify that a counterexample has to contain at least one state that is generated due to a particular event. This requirement is checked by calling the *DoesCounterexampleContainEvent()* function (Figure 2, line 13). (We have made use of this feature in some of our experiments in Section 4.) If the user does not want to make use of this feature, *DoesCounterexampleContainEvent()* always returns true. A counterexample is printed by calling the *printCounterexample()* function (Figure 2, line 15), which is a recursive function that traces the state space backwards from *nextState* until *initialState* is reached. If *nextState* does not violate a safety property, *nextState* is added to *NonVisitedStates* (Figure 2, line 20) in order to be explored later if its depth is strictly less than MAX_DEPTH (Figure 2, line 17). Adding a state to *NonVisitedStates* (Figure 2, line 20) or *AlreadyVisitedStates* (Figure 2, line 7) requires a function that creates a copy of a state (e.g., *clone*()).

Depending on the order in which states are added to, and removed from, *NonVisitedStates*, SSExploreAddCurrent() can employ breadth-first (BFS), depth-first (DFS) and best-first (BeFS) search strate-

```
procedure SSExploreAddCurrent()
 1. initialState.depth = 0 ;
 2. AlreadyVisitedStates = { } ;
 3. NonVisitedStates = { initialState } ;
 4. while ( | NonVisitedStates | > 0 ) {
 5.    currentState = NonVisitedStates.remove() ;
 6.    if ( currentState is not simulated by any state in AlreadyVisitedStates ) { /* use protocol-specific properties */
 7.       AlreadyVisitedStates.add(currentState) ;
 8.       EnabledEvents = GenerateEnabledEvents(currentState) ;
 9.       for ( int i = 0 ; i < EnabledEvents.size() ; i++ ) {
10.          EventInfo E = EnabledEvents.get(i) ;
11.          nextState = GenerateNextState(currentState, E) ;
12.          checkProperty = nextState.verifySafety() ;
13.          if ( (checkProperty == false) AND (DoesCounterexampleContainEvent(nextState)) ) {
14.             Print("Counterexample ") ;
15.             printCounterexample(nextState) ;
16.             exit ;
             } /* end if */
17.          else if ( (checkProperty == true) AND (nextState.depth < MAX_DEPTH) ) {
18.             if ( search strategy is BeFS ) {
19.                nextState.computeBeFSTuple() ;  /* use protocol-specific properties */
                }
20.             NonVisitedStates.add(nextState) ;
             } /* end if */
          }  /* end for */
       }  /* end if */
    }  /* end while */

EventInfoList GenerateEnabledEvents(GlobalState currentState)
21. EnabledEvents = { } ;
22. for ( all protocol entities p ) { /* for all protocol entities */
23.    for ( all possible events e ) { /* for all events */
24.       NumberOfNextStates = EnablingFunction(currentState, p, e) ;
25.       for ( int i = 0 ; i < NumberOfNextStates ; i++ ) { /* for all integer-valued parameters */
26.          EnabledEvents.add(new EventInfo(p, e, i)) ;
          }  /* end for */
       }  /* end for */
    }  /* end for */
27. return EnabledEvents ;

GlobalState GenerateNextState(GlobalState currentState, EventInfo E)
28. CopyFromVModelToSModel(currentState) ;
29. nextState = currentState ;  /* Start with nextState as a copy of currentState */
30. nextState.depth += 1 ;  /* Increment the depth of nextState */
31. ExecuteEvent(E) ;  /* Invoke E's event handler */
32. CopyFromSModelToVModel(nextState) ;
33. return nextState ;
```

Figure 2: An explicit-state stateful state space exploration procedure. This procedure adds a state being explored (i.e., *currentState*) to *AlreadyVisitedStates* (line 7) and a state being generated (i.e., *nextState*) to *NonVisitedStates* (line 20).

gies. Precisely, in BFS, *NonVisitedStates* is implemented as a first-in first-out (FIFO) queue; in DFS, *NonVisitedStates* is implemented as a last-in first-out (LIFO) stack; whereas in BeFS, *NonVisitedStates* is implemented as a priority queue. We call these three search strategies: BFS-AC, DFS-AC and BeFS-AC respectively[4]. A best-first search strategy is implemented by *state ranking*. Specifically, the user writes a function that assigns each state a tuple $< b_1, b_2 >$ (Figure 2, line 19) based on protocol-specific properties. The state space explorer then considers *NonVisitedStates* as a priority queue in which states are ranked

[4]AC stands for "Add Current".

```
procedure SSExploreAddNext()
 1. initialState.depth = 0 ;
 2. AlreadyVisitedStates = { initialState } ;
 3. NonVisitedStates = { initialState } ;
 4. while ( | NonVisitedStates | > 0 ) {
 5.    currentState = NonVisitedStates.remove() ;
 6.    EnabledEvents = GenerateEnabledEvents(currentState) ;  /* See Figure 2 for GenerateEnabledEvents() */
 7.    for ( int i = 0 ; i < EnabledEvents.size() ; i++ ) {
 8.       EventInfo E = EnabledEvents.get(i) ;
 9.       nextState = GenerateNextState(currentState, E) ;  /* See Figure 2 for GenerateNextState() */
10.       checkProperty = nextState.verifySafety() ;
11.       if ( (checkProperty == false) AND (DoesCounterexampleContainEvent(nextState)) ) {
12.          Print("Counterexample ") ;
13.          printCounterexample(nextState) ;
14.          exit ;
          } /* end if */
15.       else if ( (checkProperty == true) AND (nextState.depth < MAX_DEPTH) ) {
16.          if ( nextState is not simulated by any state in AlreadyVisitedStates ) { /* use protocol-specific properties */
17.             if ( search strategy is BeFS ) {
18.                nextState.computeBeFSTuple() ;  /* use protocol-specific properties */
                }
19.             AlreadyVisitedStates.add(nextState) ;
20.             NonVisitedStates.add(nextState) ;
             } /* end if */
          } /* end if */
       }  /* end for */
    }  /* end while */
```

Figure 3: An explicit-state stateful state space exploration procedure. This procedure adds a state being generated (i.e., *nextState*) to both *AlreadyVisitedStates* (line 19) and *NonVisitedStates* (line 20).

in decreasing lexicographical order of this tuple; i.e., a state s_1 is considered "better than" a state s_2 if s_1 has a higher lexicographical order of this tuple than s_2.

Note that SSExploreAddCurrent() adds a state being explored (i.e., *currentState*) to *AlreadyVisitedStates* (Figure 2, line 7). Another way of implementing the explicit-state stateful search is to add a state being generated (i.e., *nextState*) to *AlreadyVisitedStates* only if *nextState* is not simulated by a state that has been visited before. This stateful search, which we call SSExploreAddNext(), is shown in Figure 3. The difference between SSExploreAddNext() and SSExploreAddCurrent() is that the former eagerly detects and eliminates similar states while the latter lazily does so. The choice of eager versus lazy detection of similar states may affect the time and memory costs of the state space search. Since the number of similar states in a state space is generally unknown in advance and the time and memory budgets may differ from one case study to another, we provide both types of stateful searches in our framework. Similar to SSExploreAddCurrent(), SSExploreAddNext() can also employ BFS, DFS and BeFS search strategies depending on the order in which states are added to, and removed from, *NonVisitedStates*. We call these three search strategies: BFS-AN, DFS-AN and BeFS-AN respectively[5]. A third way of implementing the explicit-state stateful search is a recursive depth-first search, which does not make use of *NonVisitedStates*. We call this search strategy: DFS-R (shown in Figure 4)[6].

The performance of each of the seven search strategies mentioned above depends on the order in which enabled events are added to the list of enabled events *EnabledEvents* (Figure 2, line 26). SSExploreAddCurrent(), SSExploreAddNext() and SSExploreRecursiveDFS() assume a fixed search order; i.e., the order is the same each time the procedure executes. Specifically, the search order determined by the

[5]AN stands for "Add Next".

[6]R stands for "Recursive".

```
procedure SSExploreRecursiveDFS()
 1. initialState.depth = 0 ;
 2. AlreadyVisitedStates = { initialState } ;
 3. RecursiveDFS(initialState) ;

procedure RecursiveDFS(GlobalState s)
 4. currentState = s ;
 5. temp = currentState ;  /* save a copy of current state */
 6. EnabledEvents = GenerateEnabledEvents(currentState) ;  /* See Figure 2 for GenerateEnabledEvents() */
 7. for ( int i = 0 ; i < EnabledEvents.size() ; i++ ) {
 8.    EventInfo E = EnabledEvents.get(i) ;
 9.    nextState = GenerateNextState(currentState, E) ;  /* See Figure 2 for GenerateNextState() */
10.    checkProperty = nextState.verifySafety() ;
11.    if ( (checkProperty == false) AND (DoesCounterexampleContainEvent(nextState)) ) {
12.       Print("Counterexample ") ;
13.       printCounterexample(nextState) ;
14.       exit ;
       } /* end if */
15.    else if ( (checkProperty == true) AND (nextState.depth < MAX_DEPTH) ) {
16.       if ( nextState is not simulated by any state in AlreadyVisitedStates ) { /* use protocol-specific properties */
17.          AlreadyVisitedStates.add(nextState) ;
18.          RecursiveDFS(nextState) ;
19.          currentState = temp ;  /* restore current state */
          } /* end if */
       } /* end if */
    }  /* end for */
```

Figure 4: An explicit-state stateful state space exploration procedure. SSExploreRecursiveDFS() employs a recursive depth-first search that does not make use of *NonVisitedStates*. This procedure adds a state being generated (i.e., *nextState*) to *AlreadyVisitedStates* (line 17).

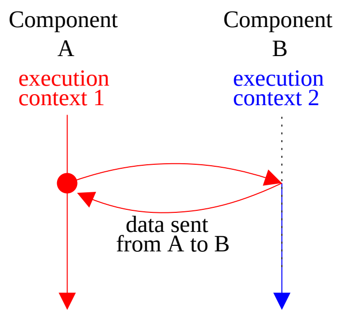
three for loops (Figure 2, lines 22-26) is: increasing order of protocol entity IDs p, increasing order of event IDs e and increasing order of event parameters i. However, it has been shown in [23] that variations in the search order can give rise to very large variations in state space exploration costs and assertion violation detection effectiveness. In order to allow for search order variations, we implement randomized versions of SSExploreAddCurrent(), SSExploreAddNext() and SSExploreRecursiveDFS(). Similar to [22], randomization is achieved by shuffling the set of enabled events at each state being explored using a Fisher-Yates shuffling algorithm [38]. Hence, the order of enabled events in *EnabledEvents* is randomized each time the function *GenerateEnabledEvents()* executes (Figure 2 (line 8) and Figures 3-4 (line 6)). Randomization in the shuffle follows a pseudo-random sequence whose seed is passed as a parameter to the state space exploration framework. We call the corresponding seven randomized search strategies: BFS-ACS, DFS-ACS, BeFS-ACS, BFS-ANS, DFS-ANS, BeFS-ANS and DFS-RS.[7]

3.3 Implementation problems and solutions

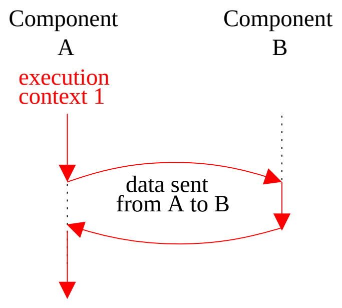
We have encountered two major implementation problems in the course of incorporating the state space explorer into J-Sim: one is related to how network protocol entities communicate with each other, with the state space explorer in between; and the other is related to the ACA timers. In this section, we describe both problems and how we solve them while keeping our two design goals met.

Without verification using state space exploration, the protocol entities communicate with each

[7]ACS, ANS and RS respectively stand for "Add Current with Shuffle", "Add Next with Shuffle" and "Recursive with Shuffle".



(a) Independence execution model



(b) Function-call execution model

Figure 5: The execution models supported by the ACA. (This figure is excerpted from [70].)

other via ports. However, when the simulation model is being verified and the state space explorer is used as shown in Figure 1, the protocol entities need to communicate with each other via the state space explorer. Initially, we simply connected the ports of each protocol entity to those of the state space explorer, but then found that protocol-specific data/control messages generated by the protocol entities during the execution of an event handler (Figure 2, line 31) may not be forwarded to the state space explorer at the required time. This is because the state space explorer does not wait until the protocol entities finish executing an event handler. This may cause the state space explorer to exclude some of the new state's information in *nextState* (Figure 2, line 32). We solve this problem by setting the ports that are involved in the interaction between the state space explorer and the protocol entities to the *function-call execution model* instead of the default *independence execution model* [70]. Figure 5 shows the two execution models supported by the ACA. In the function-call execution model, the state space explorer (Component A in Figure 5) *waits* until a protocol entity (Component B in Figure 5) finishes executing an event handler; therefore, this solution ensures that all the new state's information will be included in *nextState*. However, this solution requires modifying the J-Sim simulation model of the network protocol; hence, it might violate our second design goal. In order to keep our design goals met, we prefer making this modification in a subclass of the Java class that implements the simulation model of a network protocol entity, thus keeping the original parent Java class unmodified and ensuring that the J-Sim simulation model execution time will not degrade if the user is only interested in using the model for performance evaluation purposes. For a similar reason, we set the ports that are involved in the interaction between the state space explorer and the global states to the *function-call execution model*.

The second problem is related to the ACA timers, which are used to model timers (e.g., retransmission timers) in network simulation models in J-Sim. Without verification using state space exploration, a protocol entity that uses an ACA timer sets the timer to a pre-determined time interval. When the timer expires, a `timeout()` callback function is invoked to handle the timeout event if the timer is still active (i.e., has not already been cancelled). If the simulation model is to be verified, the state space explorer should explore all the possible transitions from a given state, and should not be limited to a single timeout value for each timer. Instead, the state space explorer should trigger the timeout event when that event may occur in the real world. For example, a typical retransmission timer in a reliable unicast protocol may expire at any time as long as there is a pending data message that has been sent but not yet acknowledged. (We assume that the setting of the interval of a timer may differ from one run of the simulation to another (which is typically the case, especially if a pseudo-random number sequence is used for generating the interval of a timer); otherwise, this approach may suffer from excessive false positives.) A possible solution to this problem is to modify the implementation of the *setTimeout()* method defined in class *Module*, a key J-Sim class that has a *timer port* used to set up (and cancel) timers. However, this solution requires modifying the core implementation of J-Sim, and hence violates our first design goal. In order to keep our design goals met, we choose to make this modification in a subclass of the Java class that implements the simulation model of a network protocol entity.

3.4 Role of the user

It should be noticed that the state space exploration process is not fully automated. We summarize below what the user needs to do, in order to verify the simulation model of a network protocol.

1. **States:** Provide an implementation of *GlobalState* (including writing the safety property as a Java method and a function that creates a copy of a state), and specify how to construct the initial state. To reduce the user'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 should be implemented as a sub-class of *SystemState*, includes the protocol-specific information.

2. **Events:** Specify (a) the set of events that exist in the network protocol, (b) the *EnablingFunction()* that returns the state-dependent number of possible successor states that an event generates; zero if the event is disabled, and (c) how each event is handled. (As mentioned above, the user has to write the event handlers anyway in order to have a working simulation model of the network protocol in J-Sim.)

3. **Simulation Relation:** Write the Java method $isSimulatedBy()$ that determines, based on protocol-specific properties, whether two states are similar. Observe that there is a general simulation relation that can be exploited when the communication is assumed to be unreliable (Section 3.2), which is what we use in our examples.

4. **State Ranking:** Write the Java method $computeBeFSTuple()$ (Figure 2 (line 19) and Figure 3 (line 18)) that assigns each state a tuple $< b_1, b_2 >$ based on protocol-specific properties such that a state s_1 is considered "better than" a state s_2 if s_1 has a higher lexicographical order of this tuple than s_2.

5. **Interaction with the State Space Explorer:** Provide implementations for the operations that involve information exchange with the state space explorer component (e.g., *CopyFromVModel-ToSModel()* and *CopyFromSModelToVModel()*). To facilitate programming, we made use of ports and contracts to provide a seamless interface between components; in this case, between the state space explorer on one side and either the protocol entities or the global states on the other side (see Figure 1). In addition, modify the Java class that implements the simulation model of a network protocol entity (or preferably its subclass), as explained in Section 3.3, to facilitate interaction with the state space explorer.

It is important to note that making use of protocol-specific properties for the simulation relation and state ranking is done in the verification model and is hence isolated from both the state space explorer and the simulation model. This ensures that the state space exploration framework is general enough and not tied to a particular network protocol or communication mechanism. In the following section, we elaborate further on this point.

4 Evaluation and Results

We apply the J-Sim state space exploration framework to the J-Sim simulation models of two network protocols: (a) Ad-Hoc On-Demand Distance Vector routing (Section 4.1) for wireless ad hoc networks and (b) Directed Diffusion (Section 4.2) for wireless sensor networks. For each protocol, we give an overview of its key functionality, describe the steps that we follow to verify its simulation model including how we exploit protocol-specific properties, and present the results of this verification. We conduct all the experiments on a dual-processor Intel Xeon 2.8 GHz machine running Linux version 2.6.15 with 2 GB memory. We use Sun's 1.5.0 04-b05 JVM, allocating 1.5 GB for the maximum heap size.

4.1 Case study 1: AODV routing in multihop wireless ad hoc networks

4.1.1 Overview of AODV

The Ad-Hoc On-Demand Distance Vector (AODV) routing protocol [51] is a well-known and widely used reactive routing protocol for multihop wireless ad-hoc networks. AODV is reactive in the sense that a route to a given destination is established via a route discovery process only when it is needed by a source node (i.e., traffic-driven). In this section, we describe the J-Sim simulation model of AODV, which is based on AODV Draft (version 11) [50].

In AODV, each node n in the ad hoc network maintains a routing table. For node n, a routing table entry (RTE) to a destination node d contains, among other fields: a next hop address $nexthop_{n,d}$ (the address of the node to which n forwards data packets destined for node d), a hop count $hops_{n,d}$ (the number of hops needed to reach node d from node n) and a destination sequence number $seqno_{n,d}$ (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 ∞.

Each node n also maintains two monotonically increasing counters: a node sequence number $seqno_n$ and a broadcast ID bid_n. When node n requires a route to a destination d, if it does not already have a valid RTE to node d, it first creates an invalid RTE to node d with $hops_{n,d}$ set to ∞. Following that, node n *broadcasts* a route request (RREQ) packet containing the following fields $< n, seqno_n, bid_n, d, seqno_{n,d}, hopCount_q >$ and then increments bid_n. The $hopCount_q$ field is initialized to 1. The pair $< n, bid_n >$ uniquely identifies a RREQ packet. Each node m, 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. If so, the incoming RREQ packet is discarded. If not, node m either satisfies the RREQ by *unicasting* a route reply (RREP) packet back to node n if it has a fresh enough route to node d (or it is node d itself), or rebroadcasts the RREQ to its own neighbors after incrementing the $hopCount_q$ field if it does not have a fresh enough route to node d (nor is itself node d). An intermediate node m determines whether it has a fresh enough route to node d by comparing the destination sequence number $seqno_{m,d}$ in its own RTE with the $seqno_{n,d}$ field in the RREQ packet. Each intermediate node also records a reverse route to the requesting node n;

this reverse route will be used to send/forward route replies to node n. The requesting node's sequence number $seqno_n$ is used to maintain the freshness of this reverse route. Each entry in the broadcast ID cache has a lifetime. Periodically, a broadcast ID timeout event is triggered causing the deletion of entries in the cache that have expired.

A RREP packet, which is sent by an intermediate node m, contains the following fields $< hopCount_p, d, seqno_{m,d}, n >$. The $hopCount_p$ field is initialized to $1 + hops_{m,d}$. If it is the destination d that sends the RREP packet, it first increments $seqno_d$ and then sends a RREP packet containing the following fields $< 1, d, seqno_d, n >$. The unicast RREP travels back to the requesting node n via the reverse route. Each intermediate node along the reverse route sets up a forward pointer to the node from which the RREP came, thus establishing a forward route to the destination d, increments the $hopCount_p$ field and forwards the RREP packet to the next hop towards n.

If node m offers node n a new route to node d, node n compares $seqno_{m,d}$ (the destination sequence number of the offered route) to $seqno_{n,d}$ (the destination sequence number of the current route), and accepts the route with the greater sequence number. If the sequence numbers are equal, the offered route is accepted only if it has a smaller hop count than the hop count in the RTE; i.e., $hops_{n,d} > hops_{m,d}$.

4.1.2 Verifying the simulation model of AODV

We next present the steps that we follow to verify the J-Sim simulation model of AODV. These steps constitute a generic methodology for verifying the simulation model of any network protocol in J-Sim.

Step 1. States: Definitions of the global state, the initial state, and the 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 node 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 J-Sim simulation model of J-Sim. Specifically, the AODV process starts with an empty routing table, empty broadcast ID cache, $seqno_n = 2$ and $bid_n = 1$.

An important safety property in a routing protocol such as AODV is the *loop-free* property. Intuitively, a node must not occur at two points on a path between two other nodes; therefore, at each hop along a path from a node n to a destination d, either the destination sequence number must increase or the hop count toward the destination must decrease. Formally, 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 [14, 45]:

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

Step 2. Events

Next, we specify the set of events, when each event is enabled and the corresponding *EnablingFunction()*, and how each event is handled. We classify the events into two categories: node events (i.e., events that are triggered inside a node) and network events (i.e., events that are triggered inside the network). The events in each category are listed as follows:

1. Node Events

   T_0 Initiation of a route request by node n to a destination $d \neq n$: This event is enabled if node n does not have a valid RTE to the destination d. When enabled, *EnablingFunction(currentState, n, T_0)* returns 1. The event is handled by broadcasting a RREQ.

   T_1 Restart of the AODV process at node n: This event may take place because of a node reboot. This event is always enabled; i.e., *EnablingFunction(currentState, n, T_1)* always returns 1. The event is handled by reinitializing the state of the AODV process at node n.

   T_2 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. When enabled, *EnablingFunction(currentState, n, T_2)* returns the number of entries in the broadcast ID cache of node n. The event is handled by deleting an entry from the broadcast ID cache of node n.

   T_3 Timeout of the route to destination d at node $n \neq d$: This event is enabled if node n has a valid RTE to node d. When enabled, *EnablingFunction(currentState, n, T_3)* returns 1. The event is handled by invalidating this RTE.

2. Network Events

   T_4 Delivering an AODV packet to node n: This event is enabled if the network contains at least one AODV packet such that node n is the destination (or the next hop towards the destination) of the packet and node n is one of the neighbors of the source of the packet. When enabled, *EnablingFunction(currentState, n, T_4)* returns the number of the AODV packets that satisfy these conditions. The event is handled by removing one of these AODV packets from the network and forwarding it to node n.

   T_5 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. When enabled, *EnablingFunction(currentState, n, T_5)* returns the number of the AODV packets that satisfy this condition. The event is handled by removing one of these AODV packets from the network.

Step 3. Simulation Relation: Exploiting the semantics of the communication medium

We use the general simulation relation outlined in the introduction and Section 3.2. For AODV specifically it reduces to the following definition. A state s_2 is said to simulate a state s_1 if (i) s_1 and s_2 have the same neighborhood information, (ii) for each AODV packet in s_1, there is a corresponding equivalent AODV packet in s_2, and (iii) 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).

Step 4. State Ranking: Exploiting protocol-specific properties

A suitable BeFS heuristic 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 any node, does not affect the truth value of the loop-free property. Therefore, a suitable BeFS heuristic (which we call AODV-1-BeFS) is to consider a state s_1 better than a state s_2 if the number of *valid* RTEs to any node in s_1 is greater than that in s_2. In other words, $< b_1, b_2 >$ is assigned to a state s such that b_1 is the number of valid RTEs to any node in s, and $b_2 = 0$.

Another BeFS heuristic (which we call AODV-2-BeFS) is 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 assertion violation. Hence, AODV-2-BeFS considers a state s_1 better than a state s_2 if the following summation

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

in s_1 is greater than that in s_2, where $nexthop_{n,d} = m$. The summation M_d 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 node d, M_d is set to $-\infty$ (i.e., worst or least interesting state).

In addition to AODV-1-BeFS and AODV-2-BeFS, we also consider the following BeFS heuristics:

1. AODV-3-BeFS: This heuristic 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 under this condition, 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. In other words, $< b_1, b_2 >$ is assigned to a state s such that b_1 is the number of valid RTEs to the destination d in s, and b_2 is the number of valid RTEs to any node in s.

2. AODV-4-BeFS: Since a valid RTE is established upon receiving a RREP packet, AODV-4-BeFS 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-5-BeFS: AODV-5-BeFS is the same as AODV-4-BeFS, except that if s_1 and s_2 are equally good under the condition specified in AODV-4-BeFS, 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.

4.1.3 Results of the verification

Clearly, the state space created by the J-Sim simulation model of AODV is infinite. Furthermore, there is an infinite number of possible initial states depending on the number of nodes and the network topology. As an attempt towards handling the state space explosion problem, we (1) consider an initial state of an ad hoc network consisting of N nodes: n_0, n_1, ..., n_{N-1} 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); i.e., all wireless links are assumed to be bidirectional, and (2) reduce the number of events and states by considering only one destination node n_{N-1}. Therefore, all RREQ packets request a route to node n_{N-1} and the route timeout event invalidates the RTE to node n_{N-1} only. Furthermore, the loop-free property checks the absence of routing loops to node n_{N-1} only. Although this scenario is simple, it ensures that nodes $n_0, n_1, \ldots, n_{N-3}$ require multihop routes to reach node n_{N-1}; i.e., AODV multihop routing is needed. In addition, if an assertion is violated in a chain network topology, it may also be violated in an arbitrary network topology.

By verifying the J-Sim simulation model of AODV in a chain network topology consisting of $N = 3$ nodes, we have discovered an assertion violation (which we call Counterexample 1) in the AODV simulation model caused by its failure to follow a part of the AODV specification [50] that determines certain actions that must be taken after a node reboot. Conceptually, if $nexthop_{0,2} = 1$ and the AODV process at node n_1 restarts due to a node reboot, the net effect is that all the RTEs stored at node n_1 will be deleted. As a result, node n_1 may later accept a route that was offered by node n_2 with a lower sequence number than that of node 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 state space exploration framework was able to find these two errors too.[8] A routing loop may occur due to either of these two errors. This is because in the case that $nexthop_{0,2} = 1$ and a route timeout event takes place at node n_1, in either Counterexample 2 or 3, if n_1 is later offered a route to node n_2 by node 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 [60] for detailed traces (along with the explanations) of the three counterexamples.

Table 1 gives the performance of the various randomized search strategies, for finding each of the three counterexamples, with respect to the following two types of performance evaluation criteria: (a) *platform-independent*; namely, the number of events executed and the number of states stored in memory (sum of the sizes of *AlreadyVisitedStates* and *NonVisitedStates*)[9], and (b) *platform-dependent*; namely, the time needed to find an assertion violation. We ran 100 experiments for each search strategy. Each experiment has a different seed, but the same set of 100 seeds were used for each of the seven search strategies. For each performance evaluation criterion, we report the minimum, the maximum and the average values. As shown in Table 1, AODV-1-BeFS-ACS (AODV-1-BeFS-ANS)[10] achieves significant reduction with respect to the evaluation criteria when compared to other standard search strategies such as BFS-ACS and DFS-ACS (BFS-ANS and DFS-ANS) respectively. Also, the choice of the BeFS heuristic has an impact on the performance. As shown in Table 1, AODV-2-BeFS-ACS (AODV-2-BeFS-ANS) performs worse than AODV-1-BeFS-ACS (AODV-1-BeFS-ANS) for the three counterexamples. This is because AODV-2-BeFS requires that a node (and its next hop towards the destination) 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 non-visited states may look equally good and thus, AODV-2-BeFS may not be able to explore the states that are most likely to lead to the assertion violation first. AODV-3-BeFS tackles this problem by further differentiating equally good states by using a two-level BeFS heuristic. Hence, as shown in Table 1, AODV-3-BeFS-ACS and AODV-5-BeFS-ACS (AODV-3-BeFS-ANS and AODV-5-BeFS-ANS) respectively outperform AODV-2-BeFS-ACS and AODV-4-BeFS-ACS (AODV-2-BeFS-ANS and AODV-4-BeFS-ANS) because they are able to better

[8]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_3. In order to achieve that, we made use of the *DoesCounterexampleContainEvent()* function provided by the state space exploration framework (Figures 2-4).

[9]Note that we do not report the total number of states generated because it is simply equal to the number of events executed plus 1 (to account for the initial state).

[10]We explain the notation as follows: AODV-1-BeFS denotes the BeFS heuristic itself whereas AODV-1-BeFS-ACS denotes the BeFS-ACS search strategy when making use of the AODV-1-BeFS heuristic.

Table 1: AODV case study: Time and space requirements (sum of the sizes of *AlreadyVisitedStates* and *NonVisitedStates*) and the number of events executed for finding the three counterexamples in a 3-node chain ad-hoc network using several randomized search strategies.

Counterexample 1	Time (s.)			Space (number of states)			Events		
$MAX\_DEPTH = 10$	Min	Max	Average	Min	Max	Average	Min	Max	Average
DFS-RS	0.116	14.72	3.826	20	1957	784.95	156	26215	10749.4
BFS-ACS	4.498	13.63	8.564	5995	11092	9458.18	9167	17098	14417.93
DFS-ACS	0.267	17.851	4.12	136	2084	851.12	722	27617	10958.45
AODV-1-BeFS-ACS	0.511	3.937	1.37	416	1393	709.69	1832	10965	4695.6
AODV-2-BeFS-ACS	0.214	16.734	3.431	148	2156	755.7	378	28885	8755.22
AODV-3-BeFS-ACS	0.377	4.107	1.0	278	1398	546.67	1106	11164	3170.88
AODV-4-BeFS-ACS	0.528	6.283	2.312	252	1998	937.46	1934	16313	7990.38
AODV-5-BeFS-ACS	0.535	3.966	1.718	320	1572	952.41	2045	10941	6308.36
BFS-ANS	16.007	54.494	33.365	2575	4603	3886.49	9167	17098	14417.93
DFS-ANS	0.24	16.173	3.69	100	2037	765.31	507	27753	9932.62
AODV-1-BeFS-ANS	0.514	4.477	1.548	433	1266	674.77	1571	10421	4397.15
AODV-2-BeFS-ANS	0.223	12.825	2.787	170	1839	695.18	364	22337	7343.08
AODV-3-BeFS-ANS	0.343	4.8	1.11	296	1263	538.67	954	10710	2976.71
AODV-4-BeFS-ANS	0.456	7.301	2.468	211	1736	811.82	1687	15102	7362.44
AODV-5-BeFS-ANS	0.541	5.569	2.236	233	1452	849.85	2004	10767	6139.34

Counterexample 2	Time (s.)			Space (number of states)			Events		
$MAX\_DEPTH = 10$	Min	Max	Average	Min	Max	Average	Min	Max	Average
DFS-RS	0.096	18.213	6.44	20	2230	1069.37	109	30454	14757.33
BFS-ACS	4.399	13.096	9.04	6496	12389	9748.7	9873	18744	14907.87
DFS-ACS	0.196	19.364	5.669	97	2363	1017.75	458	31077	13351.86
AODV-1-BeFS-ACS	0.105	10.203	2.934	88	1849	716.48	118	21999	6923.73
AODV-2-BeFS-ACS	0.22	20.346	4.152	148	2391	850.84	375	30871	9916.79
AODV-3-BeFS-ACS	0.115	10.492	2.447	96	1897	558.84	129	22185	5361.24
AODV-4-BeFS-ACS	0.383	8.427	4.901	195	2097	1502.81	1302	19424	13518.75
AODV-5-BeFS-ACS	0.453	6.098	2.348	344	1935	985.5	1597	15938	7048.29
BFS-ANS	15.159	56.095	35.232	2796	5012	3991.61	9873	18744	14907.87
DFS-ANS	0.2	18.831	4.955	96	2263	920.49	483	30646	12117.64
AODV-1-BeFS-ANS	0.124	11.499	3.31	97	1808	685.9	103	21937	6730.85
AODV-2-BeFS-ANS	0.212	18.387	3.606	168	2219	799.66	369	29352	8671.63
AODV-3-BeFS-ANS	0.13	11.412	2.69	108	1842	546.27	117	22085	5199.88
AODV-4-BeFS-ANS	0.28	9.454	5.641	121	1859	1323.48	761	19492	13001.43
AODV-5-BeFS-ANS	0.479	7.7	2.85	328	1632	829.49	1562	15394	6759.06

Counterexample 3	Time (s.)			Space (number of states)			Events		
$MAX\_DEPTH = 10$	Min	Max	Average	Min	Max	Average	Min	Max	Average
DFS-RS	0.21	12.308	3.167	62	1820	693.97	509	24509	9514.15
BFS-ACS	3.985	11.406	7.601	6074	10863	8734.19	9272	16941	13461.48
DFS-ACS	0.169	15.292	3.493	103	2147	770.35	238	28324	10058.98
AODV-1-BeFS-ACS	0.095	3.403	1.018	84	1346	512.0	99	10540	3429.25
AODV-2-BeFS-ACS	0.25	8.991	2.657	160	1512	679.49	441	19226	7844.9
AODV-3-BeFS-ACS	0.113	3.395	0.804	96	1352	413.36	128	10797	2533.99
AODV-4-BeFS-ACS	0.41	6.565	2.097	266	2085	868.41	1496	17130	7428.23
AODV-5-BeFS-ACS	0.446	1.497	0.742	303	791	490.2	1599	5518	2883.28
BFS-ANS	13.467	42.551	28.355	2587	4302	3516.31	9272	16941	13461.48
DFS-ANS	0.132	14.588	3.051	91	1912	693.38	144	25733	9199.9
AODV-1-BeFS-ANS	0.12	3.983	1.145	97	1190	468.08	99	10043	3218.25
AODV-2-BeFS-ANS	0.255	10.165	2.335	166	1608	641.02	394	20821	6984.61
AODV-3-BeFS-ANS	0.128	3.94	0.893	108	1153	383.9	112	10237	2363.52
AODV-4-BeFS-ANS	0.404	8.11	2.28	226	1857	767.34	1442	15929	6978.01
AODV-5-BeFS-ANS	0.47	1.745	0.827	237	703	434.18	1546	5361	2793.47

guide the best-first search towards the assertion violation even at the lower depths of the search space.

4.1.4 Comparison with the Java PathFinder (JPF) model checker

To further evaluate the state space exploration framework in J-Sim, we compare its performance to that of Java PathFinder (JPF) [35, 72], a state-of-the-art model checker for Java programs. In this section, we first give an overview of JPF (version 4.1). Following that, we elaborate on how we enable JPF to explore the state space of the J-Sim simulation model of AODV. Finally, we present the performance results.

Overview of JPF JPF is an explicit-state model checker for Java programs. JPF takes as input a Java program and an optional bound on the length of program execution. JPF explores all executions (up to the given bound) that the program can have due to different thread interleavings and nondeterministic choices. JPF can generate as output the traces of those executions that violate a given property; e.g., the loop-free safety property in AODV. JPF is implemented in Java as a special Java Virtual Machine (JVM) that runs on top of the host JVM.[11] The main difference between JPF and a regular JVM is that JPF implements a *backtrackable* JVM; i.e., JPF can (quickly) backtrack the program execution by restoring a state that was previously encountered during the execution. Backtracking allows exploration of different executions from the same state. To achieve fast backtracking, JPF uses a *special* representation of states and executes program bytecodes by modifying this representation. For instance, JPF represents the heap as follows: JPF uses integers to represent object identifiers and to encode all field values, be they primitive (int, boolean, float, etc.) or pointers to other objects (which can hold the special value null). (JPF determines the meaning of various integers based on the field types kept in the class information.) In fact, JPF represents each object as an integer array, and the entire heap as an array of integer arrays. This special representation enables JPF to quickly store and restore states; it is crucial for making the overall state space exploration efficient.

Model Java Interface (MJI) is a JPF mechanism that allows parts of JPF execution to be delegated from the JPF into the host JVM. MJI is analogous to the Java Native Interface (JNI) [66] that allows parts of JVM execution to be delegated from the JVM into the native code, written in say the C language. MJI, like JNI, splits executions at the method granularity; specifically, each method can be marked to be executed either in JPF or in the host JVM. (JPF uses special name mangling to mark methods for the host JVM execution.) MJI also provides API that allows the host JVM execution to manipulate the JPF special state representation, for example to read or write field values or to create new objects. One advantage of MJI is that it can be used to improve the performance of JPF [20].

Enabling JPF to explore the state space of the J-Sim simulation model of AODV JPF could *not* execute the code for the entire J-Sim simulator and the AODV protocol. Therefore, we had to create a simplified version of the J-Sim simulation model of AODV. This version does not have the full generality of the J-Sim simulator but provides the basic functionality needed to run AODV. Following that, we wrote a test driver for the (simplified) J-Sim simulation model of AODV. The driver produces an environment that checks which events are enabled, and initiates the execution of all sequences of enabled events up to a configurable maximum depth (MAX_DEPTH). Similar drivers were previously used in several studies on JPF [73–75].

Figure 6 gives the pseudo-code of the driver for state space exploration in JPF. This code is executed in JPF's backtrackable JVM. First, the *aodvNodes* array, whose elements correspond to the wireless nodes in the ad hoc network, is created and initialized (Figure 6, line 1). As long as the safety property has not been violated and MAX_DEPTH bound has not been reached, the while loop in Figure 6 (lines 4 to 15) is executed. Each iteration of the loop corresponds to a state being explored (i.e., *currentState*).

[11] In contrast to JPF, our state space exploration framework in J-Sim does not require a special JVM.

The events that are enabled in *currentState* are determined by invoking the *GenerateEnabledEvents()* function (Figure 6, line 5). These events are stored in the *EnabledEvents* list, which is returned by *GenerateEnabledEvents()* (Figure 6, line 22). Following that, the JPF's library method Verify.random(int maxBound) is invoked (Figure 6, line 6); this method nondeterministically returns a number between zero and $maxBound$. Hence, Figure 6, line 6, nondeterministically chooses an enabled event. This chosen event is then executed (Figure 6, line 7) and the new state's information is stored in the *aodvNodes* array. Following that, the safety property is checked (Figure 6, line 8). If the safety property is violated, a counterexample is printed (Figure 6, line 11); otherwise, if the depth of the next state is strictly less than MAX_DEPTH (Figure 6, line 13), the driver checks whether the next state is simulated by a state that has been visited before by invoking the *wasSimulated()* function using the MJI API (Figure 6, line 14). If *wasSimulated()* returns true, the JPF's library method Verify.ignoreIf() instructs JPF to backtrack the execution; hence, another enabled event will be nondeterministically chosen in Figure 6, line 6. If not, the state space exploration proceeds in the search order determined by the search strategy: breadth-first (BFS), depth-first (DFS) or best-first (BeFS). It should be noted that the driver (Figure 6) does not contain code for storing and restoring states because this is done by JPF. (In our state space exploration framework, storing and restoring states is implemented by adding states to, and removing them from, *NonVisitedStates*.)

In summary, the tasks that are executed inside the JPF's backtrackable JVM are: the overall state space exploration process, checking which events are enabled in each state being explored, executing the enabled events and checking the safety property. On the other hand, the tasks that are executed inside the host JVM are: implementing a stateful search (using the simulation relation that determines state similarity), and implementing a best-first search (using state ranking by assigning a tuple $< b_1, b_2 >$ to each state being generated). The reason why these tasks have to be done inside the host JVM instead of the JPF's backtrackable JVM is that these tasks depend on the *protocol-specific* properties (e.g., the routing table entries in AODV). Due to the JPF special state representation, these protocol-specific properties are represented using (arrays of) integers; hence, not directly accessible. Therefore, we make use of the MJI API to manipulate the JPF special state representation, extract the protocol-specific properties and construct an instance of the *GlobalState* class inside the regular host JVM. This is achieved by the constructGlobalState() function call (Figure 7, line 1). Following that, we can check state similarity (Figure 7, line 2) and enable state ranking by assigning a tuple $< b_1, b_2 >$ to *nextState* (Figure 7, line 4) in the same way as we did with instances of the *GlobalState* class in J-Sim.

Results of the comparison between J-Sim and JPF Table 2 gives the (i) time, (ii) space (size of *AlreadyVisitedStates*), and (iii) number of events executed for finding the three counterexamples using both J-Sim and JPF with several search strategies. Since we used the same definition of the simulation relation that determines state similarity and the same implementations of BeFS heuristics for state ranking in both tools, the number of events executed and the number of states in *AlreadyVisitedStates* are *exactly* the same for both J-Sim and JPF. (We have also verified that this is the case for lower values of MAX_DEPTH where the assertion violations do not occur. In fact, we have also verified that the *sequence* in which states are visited is exactly the same for both J-Sim and JPF.) Therefore, both J-Sim and JPF are having the same amount of "workload". Table 2 shows the time needed by both tools to finish that workload. The last column of Table 2 shows the ratio between the time needed by JPF and that needed by J-Sim to find the assertion violation. In the case that a small number of events is executed (e.g., the cases of AODV-2-BeFS-AN and AODV-3-BeFS-AN), JPF is much slower than J-Sim. In the case that a moderate number of events is executed (e.g., the cases of AODV-4-BeFS-AN and AODV-5-BeFS-AN in Counterexamples 1 and 3), JPF is slower than J-Sim. In the case that a large number of events is executed (e.g., the cases of DFS-R and BFS-AN), the time needed to find an assertion violation

```
procedure JPFDriver()
 1. initialize(aodvNodes) ;
 2. depth = 0 ;
 3. checkProperty = true ;
 4. while ( (checkProperty == true) AND (depth < MAX_DEPTH) ) {
 5.    EnabledEvents = GenerateEnabledEvents() ;
 6.    eventID = Verify.random(EnabledEvents.size() - 1) ;
 7.    ExecuteEvent(EnabledEvents.get(eventID)) ;
 8.    checkProperty = verifySafety() ;
 9.    if ( (checkProperty == false) AND (DoesCounterexampleContainEvent(nextState)) ) {
10.       Print("Counterexample ") ;
11.       printCounterexample() ;
12.       exit ;
       } /* end if */
13.    else if ( (checkProperty == true) AND ((depth + 1) < MAX_DEPTH) ) {
14.       Verify.ignoreIf(wasSimulated(aodvNodes)) ;  /* wasSimulated() is invoked using MJI API */
       } /* end if */
15.    depth = depth + 1 ;
    }  /* end while */

EventInfoList GenerateEnabledEvents()
16. EnabledEvents = { } ;
17. for ( all protocol entities p ) { /* for all protocol entities */
18.    for ( all possible events e ) { /* for all events */
19.       NumberOfNextStates = EnablingFunction(p, e) ;
20.       for ( int i = 0 ; i < NumberOfNextStates ; i++ ) { /* for all integer-valued parameters */
21.          EnabledEvents.add(new EventInfo(p, e, i)) ;
          }  /* end for */
       }  /* end for */
    }  /* end for */
22. return EnabledEvents ;
```

Figure 6: Driver for state space exploration in JPF. This code is executed in JPF's backtrackable JVM.

```
native boolean wasSimulated(MJIEnv env, int classRef, int objRef)
/* wasSimulated() is ``native''; hence, it is executed in the regular host JVM. */
 1. nextState = constructGlobalState(env, objRef) ;
 2. if ( nextState is not simulated by any state in AlreadyVisitedStates ) { /* use protocol-specific properties */
 3.    if ( search strategy is BeFS ) {
 4.       nextState.computeBeFSTuple() ;  /* use protocol-specific properties */
       }
 5.    AlreadyVisitedStates.add(nextState) ;
 6.    return false ;
    }
 7. else
 8.    return true ;
    }
```

Figure 7: Code executed in the regular host JVM.

by JPF is close to that of our state space exploration framework in J-Sim, with JPF outperforming J-Sim in only one case of BFS-AN.

In order to further compare the performance of the state space exploration framework in J-Sim with that in JPF in the cases that a large number of events is executed, we implemented the shuffling algorithm, which was mentioned in Section 3.2, in JPF and ran 100 experiments for each of the BFS-ANS and DFS-RS search strategies and the three counterexamples. Each experiment has a different seed, but the same set of 100 seeds were used for each of J-Sim and JPF. In all experiments, we have verified that

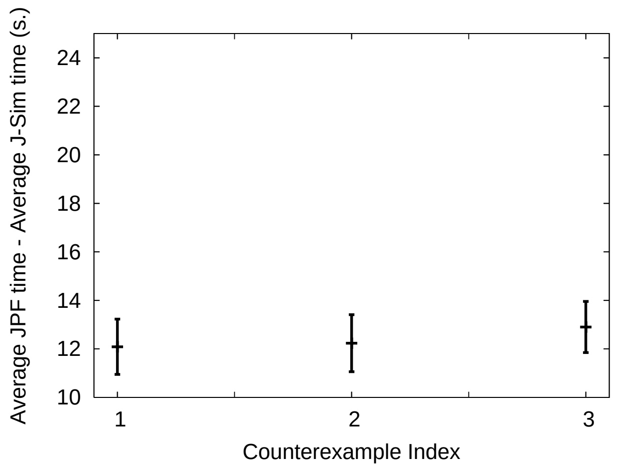
Table 2: AODV case study: Time and space requirements (size of *AlreadyVisitedStates*) and the number of events executed for finding the three counterexamples in a 3-node chain ad-hoc network using both J-Sim and JPF with several search strategies.

Counterexample 1 $MAX\_DEPTH = 10$	Space (number of states)	Events	JPF Time (s.)	J-Sim Time (s.)	JPF/J-Sim Time Ratio
BFS-AN	2828	16604	50.955	49.31	1.033
DFS-R	1086	15939	29.007	7.34	3.952
AODV-1-BeFS-AN	411	3515	7.603	1.128	6.74
AODV-2-BeFS-AN	311	1236	4.422	0.535	8.265
AODV-3-BeFS-AN	206	1004	3.652	0.433	8.434
AODV-4-BeFS-AN	582	6376	12.921	2.048	6.309
AODV-5-BeFS-AN	523	5532	11.265	1.839	6.126
Counterexample 2 $MAX\_DEPTH = 10$	Space (number of states)	Events	JPF Time (s.)	J-Sim Time (s.)	JPF/J-Sim Time Ratio
BFS-AN	2886	16928	51.729	51.915	0.996
DFS-R	1128	16533	29.896	8.215	3.639
AODV-1-BeFS-AN	407	3462	7.61	1.125	6.764
AODV-2-BeFS-AN	306	1192	4.329	0.54	8.017
AODV-3-BeFS-AN	206	965	3.598	0.421	8.546
AODV-4-BeFS-AN	1563	18536	34.023	10.017	3.397
AODV-5-BeFS-AN	1330	15314	28.448	8.063	3.528
Counterexample 3 $MAX\_DEPTH = 10$	Space (number of states)	Events	JPF Time (s.)	J-Sim Time (s.)	JPF/J-Sim Time Ratio
BFS-AN	2644	16002	56.858	44.213	1.286
DFS-R	1052	15497	33.836	7.011	4.826
AODV-1-BeFS-AN	406	3453	9.227	1.075	8.583
AODV-2-BeFS-AN	303	1182	5.246	0.508	10.327
AODV-3-BeFS-AN	205	956	4.343	0.408	10.645
AODV-4-BeFS-AN	574	6286	15.693	2.001	7.843
AODV-5-BeFS-AN	515	5300	13.49	1.741	7.748

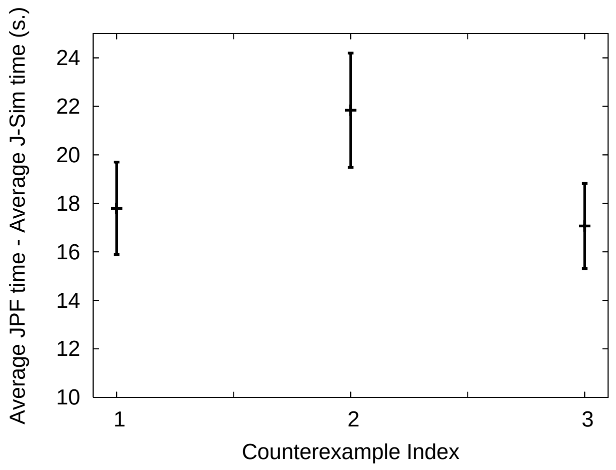
the sequence in which states are visited is exactly the same for both J-Sim and JPF; i.e., the common random numbers are synchronized. Figure 8 shows the difference between the average JPF time and the average J-Sim time for finding the three counterexamples in a 3-node chain ad-hoc network topology. As shown in Figure 8, the time needed to find an assertion violation by our state space exploration framework in J-Sim is several seconds less than that of JPF.

Next, we compare the scalability of our state space exploration framework in J-Sim with that of JPF. Specifically, we study the effect of increasing the number of nodes, N, in the network on the performance of both tools with respect to the time needed to find an assertion violation. Table 3 gives the (i) time, (ii) space (size of *AlreadyVisitedStates*), and (iii) number of events executed for finding Counterexample 3 in a N-node chain ad-hoc network using both J-Sim and JPF with the AODV-1-BeFS-AN search strategy. As shown in Table 3, our framework is able to find a counterexample in larger network topologies within reasonable time and space requirements. Furthermore, in the case that a large number of events is executed (e.g., the cases of $N = 11$ and $N = 12$), the time needed to find an assertion violation by our state space exploration framework in J-Sim is comparable to that of JPF.

The overall conclusion drawn from Tables 2-3 and Figure 8 is that our state space exploration framework in J-Sim is comparable to JPF in terms of the time needed to find an assertion violation. This justifies the need for building a framework in J-Sim for the verification of network simulation models by state space exploration instead of using a general-purpose model checking tool since we believe that wireless network protocol designers and simulation modelers will feel more comfortable using J-Sim as a single integrated environment for both building a simulation model and verifying its correctness than using J-Sim for building a simulation model and using another tool (JPF) for verifying its correctness.



(a) BFS-ANS



(b) DFS-RS

Figure 8: AODV case study: The difference between the average JPF time and the average J-Sim time for finding the three counterexamples in a 3-node chain ad-hoc network topology using the BFS-ANS and DFS-RS search strategies and $MAX\_DEPTH = 10$. We report the difference in means obtained from the 100 experiments and the 99% confidence interval.

4.2 Case study 2: Directed diffusion in wireless sensor networks (WSNs)

4.2.1 Overview of directed diffusion

A major objective of a wireless sensor network (WSN) is to monitor and sense events of interests (e.g., changes in the acoustic sound, seismic, or temperature) in a specific environment. Events of interest are generated by *target nodes*. For instance, a moving tank in a battlefield may generate ground vibrations that can be detected by seismic sensors. Upon detecting an event of interest, *sensor nodes* send reports to *sink (user) nodes* either periodically or on demand. From the perspective of network simulation, a WSN typically consists of these three types of nodes: sensor nodes (that sense and detect the events of interest), target nodes (that generate events of interest), and sink nodes (that utilize and consume the sensor information). The implementation details of the simulation and emulation frameworks for WSNs in J-Sim can be found in [56–58]. In this section, we describe the J-Sim simulation model of directed diffusion.

Directed diffusion [33] is a *data-centric* information dissemination paradigm for WSNs. Conceptually, *data* in WSNs is the collected (or processed) information of a physical phenomenon. 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 explosion in a specific area). INTEREST packets are diffused throughout the network; e.g., via flooding. After receiving an INTEREST packet, a node may decide to re-send the INTEREST packet to its neighbors, or suppress a received INTEREST if it has recently resent a matching INTEREST. INTEREST packets 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. It should be noted that this mechanism results in neighboring nodes establishing a gradient toward each other.

Table 3: AODV case study: Time and space requirements (size of *AlreadyVisitedStates*) and the number of events executed for finding Counterexample 3 in a N-node chain ad-hoc network using both J-Sim and JPF with the AODV-1-BeFS-AN search strategy.

N	MAX_DEPTH	Space (number of states)	Events	JPF Time (s.)	J-Sim Time (s.)	JPF/J-Sim Time Ratio
3	15	73	155	2.665	0.174	15.316
4	20	408	2138	9.728	1.002	9.709
5	25	1884	23834	98.206	26.248	3.741
6	30	517	2939	19.472	1.802	10.806
7	35	596	1862	22.629	1.954	11.581
8	40	828	2851	40.693	3.778	10.771
9	45	3344	87052	554.823	180.637	3.071
10	50	3761	108106	749.47	232.425	3.225
11	55	4000	121728	923.757	248.968	3.71
12	60	4069	127635	1201.156	274.348	4.378

This is because when a node receives an INTEREST from its neighbor, it has no way of knowing whether that INTEREST was in response to one it sent out earlier or is an identical INTEREST from another sink on the other side of that neighbor.

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 been removed, the interest entry itself is removed from the interest cache.

When an INTEREST packet arrives at a sensor node that can sense data which matches the interest, this sensor node becomes a *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, it is discarded; otherwise, the node adds the received DATA packet to the data cache and forwards it to each neighbor for whom the node 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 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. It should be noticed that this mechanism can result in more than one path being reinforced, thereby consuming more energy. Furthermore, one reinforced path may turn out to be consistently “better” than another path, which then needs to be negatively reinforced. Specifically, a *negative reinforcement* packet is used to inform a neighbor to send data at the lower rate determined by the exploratory interval. Similar to positive reinforcements, a data-driven local rule is used to decide whether to negatively reinforce a neighbor or not. One plausible rule is to negatively reinforce a neighbor from whom no new events have been received within a window of W events (i.e., the neighbor that consistently sends previously seen events).

4.2.2 Verifying the simulation model of directed diffusion

We follow the same steps in Section 4.1.2 to verify the simulation model of directed diffusion.

Step 1. States: Definitions of the global state, the initial state, and the safety property

We use the same definitions of *GlobalState* and network cloud that were introduced in Section 4.1.2. 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.

The safety property that we check 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:

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

Step 2. Events

Next, we specify the set of events, when each event is enabled and the corresponding *EnablingFunction()*, and how each event is handled. We classify the events into two categories: node events and network events. The events in each category are listed as follows:

1. Node Events

   T_0 Initiation of a sensing task by node n: This event is enabled if node n is a sink node. When enabled, *EnablingFunction(currentState, n,* T_0*)* returns 1. The event is handled by broadcasting an INTEREST packet.

   T_1 Restart of the directed diffusion process at node n: This event may take place because of a node reboot. This event is always enabled; i.e., *EnablingFunction(currentState, n,* T_1*)* always returns 1. The event is handled by reinitializing the state of the directed diffusion process at node n.

   T_2 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. When enabled, *EnablingFunction(currentState, n,* T_2*)* returns the total number of gradients in the interest cache of node n. The event is handled by removing one of the gradients in the interest cache of node n. If all gradients in an interest entry have been removed, the interest entry itself is removed from the interest cache.

   T_3 Data cache timeout[12] at node n: This event is enabled if there is at least one entry in the data cache of node n. When enabled, *EnablingFunction(currentState, n,* T_3*)* returns the number of entries in the data cache of node n. The event is handled by deleting an entry from the data cache of node n.

[12]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 J-Sim simulation model of directed diffusion, 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.

2. Network Events

   T_4 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. When enabled, *EnablingFunction(currentState, n, T_4)* returns the number of the packets that satisfy this condition. The event is handled by removing one of these packets from the network and forwarding it to node n.

   T_5 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. When enabled, *EnablingFunction(currentState, n, T_5)* returns the number of the packets that satisfy this condition. The event is handled by removing one of these packets from the network.

Step 3. Simulation Relation: Exploiting the semantics of the communication medium

Again, we use the general simulation relation outlined in the introduction and Section 3.2. For directed diffusion this reduces to the following definition. A state s_2 is said to simulate a state s_1 if (i) s_1 and s_2 have the same neighborhood information, (ii) for each packet in s_1, there is a corresponding equivalent packet in s_2, and (iii) 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).

Step 4. State Ranking: Exploiting protocol-specific properties

In the course of verifying the J-Sim simulation model of AODV, AODV-1-BeFS and AODV-3-BeFS provided comparatively better performance results. Hence we use these two BeFS heuristics to devise two corresponding BeFS heuristics for directed diffusion. In particular, the loop-free property for AODV involves only valid RTEs to a destination d, and by analogy, the loop-free property for directed diffusion involves only reinforced gradients. Similarly, data packets are forwarded in AODV based on the next hop information stored in the valid RTEs, and by analogy, data packets are forwarded in directed diffusion based on the gradients established at the nodes. Therefore, two potentially good BeFS heuristics for exploring the state space of directed diffusion are:

1. DD-1-BeFS: This heuristic 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. In other words, $< b_1, b_2 >$ is assigned to a state s such that b_1 is the total number of both exploratory and reinforced gradients in s, and $b_2 = 0$.

2. DD-2-BeFS: This heuristic 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 under this condition, 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. In other words, $< b_1, b_2 >$ is assigned to a state s such that b_1 is the number of reinforced gradients in s, and b_2 is the total number of both exploratory and reinforced gradients in s.

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

1. DD-3-BeFS: Since a reinforced gradient is established upon receiving a positive reinforcement packet, DD-3-BeFS 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-4-BeFS: DD-4-BeFS is the same as DD-3-BeFS, except that if s_1 and s_2 are equally good under the condition specified in DD-3-BeFS, 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-5-BeFS: This heuristic 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-6-BeFS: DD-6-BeFS is the same as DD-5-BeFS, except that if s_1 and s_2 are equally good under the condition specified in DD-5-BeFS, 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.

4.2.3 Results of the verification

We consider an initial state that consists of a chain topology of N nodes: n_0 (the only sink node), n_1, ..., n_{N-1} (the only source node). A loop in the reinforced path may take place because 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 the case of (a) the deletion of a DATA packet from the data cache and/or (b) a node reboot (which effectively deletes all the entries in the data and interest caches), a loop may be created. The loop that is created in the first case is referred to as Counterexample 1 while the loop that is created in the second case is referred to as Counterexample 2. For instance, consider a chain topology consisting of $N = 4$ nodes. 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. (Recall that neighboring nodes establish gradients toward each other.) 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).[13] The interested reader is referred to [61] for detailed traces (along with the explanations) of the two counterexamples. It has to be mentioned that although the reinforced path may have a loop, this loop will not continue to exist forever. It will be removed later either by the negative reinforcement mechanism or by the gradient timeout mechanism. Furthermore, forwarding a DATA packet over the loop will stop once all nodes on the loop have received the DATA packet.

Table 4 gives the (i) time, (ii) space (sum of the sizes of *AlreadyVisitedStates* and *NonVisitedStates*), and (iii) number of events executed for finding the two counterexamples using several search strategies. As shown in Table 4, DD-1-BeFS-AN provides orders of magnitude reduction with respect to the evaluation criteria when compared to other standard search strategies such as BFS-AN and DFS-R. Furthermore, DD-4-BeFS-AN outperforms DD-3-BeFS-AN, and DD-6-BeFS-AN outperforms DD-5-BeFS-AN. This is because both DD-4-BeFS and DD-6-BeFS are two-level BeFS heuristics that use DD-1-BeFS if the non-visited states are equally good and are thus able to better guide the best-first search strategy, in the lower depths of the search space, than DD-3-BeFS and DD-5-BeFS respectively.

[13]For Counterexample 2, we require that the counterexample contain at least one state that is generated due to a node reboot event, T_1. Furthermore, in order to show that a loop in the reinforced path may still take place even if the data cache timeout event, T_3, does not happen (i.e., the data cache size is infinite), we disabled T_3; i.e., we made *EnablingFunction(currentState, n, T_3)* always return zero.

Table 4: Directed diffusion case study: Time and space requirements (sum of the sizes of *AlreadyVisitedStates* and *NonVisitedStates*) and the number of events executed for finding the two counterexamples in a 4-node chain WSN using different search strategies. N/A indicates that the state space explorer is not able to find a counterexample in two hours.

	Counterexample 1, $MAX\_DEPTH = 15$		
	Time (s.)	Space (number of states)	Events
BFS-AN	915.115	16717	115698
DFS-R	384.863	8306	173823
DD-1-BeFS-AN	0.772	512	1019
DD-2-BeFS-AN	0.794	500	1256
DD-3-BeFS-AN	153.405	5638	113342
DD-4-BeFS-AN	0.396	284	474
DD-5-BeFS-AN	643.72	14284	175807
DD-6-BeFS-AN	402.375	12985	95036
	Counterexample 2, $MAX\_DEPTH = 20$		
	Time (s.)	Space (number of states)	Events
BFS-AN	6975.04	44328	425181
DFS-R	380.97	8027	179112
DD-1-BeFS-AN	344.434	18063	92820
DD-2-BeFS-AN	516.592	20435	132382
DD-3-BeFS-AN	5644.553	35175	835463
DD-4-BeFS-AN	51.816	5464	38304
DD-5-BeFS-AN	N/A	N/A	N/A
DD-6-BeFS-AN	5573.121	55164	505582

Table 5 gives the time and space requirements and the number of events executed for finding Counterexample 1 in a chain topology consisting of N nodes using DD-4-BeFS-AC. Again our framework is able to find a counterexample in larger network topologies within reasonable time and space requirements.

4.3 Lessons learned

In this section, we summarize the lessons learned in the two case studies. First, we show that the state space exploration framework is able to verify the simulation models of two fairly complex network protocols such as AODV and directed diffusion. This demonstrates that the framework is general enough and not tied to a particular network protocol. To verify the simulation model of another network protocol, one needs to follow the steps as outlined in Sections 4.1-4.2.

Second, we demonstrate that the use of BeFS heuristics reduces the time and space requirements by several orders of magnitude when compared to classic breadth-first and depth-first search strategies. Based on the results obtained for the BeFS heuristics that we devised, we recommend deriving the BeFS heuristic from properties inherent to the network protocol and the safety property being checked. This is justified by the observation that AODV-1-BeFS and DD-1-BeFS provide better performance results in terms of time and space requirements and number of events executed for finding a violation of a safety property as shown in Table 1 and Table 4 respectively. Furthermore, using a two-level BeFS heuristic, in which a BeFS heuristic such as AODV-1-BeFS or DD-1-BeFS is used if the non-visited states are equally good, also improved the performance. This is justified by the observation that AODV-5-BeFS outperforms AODV-4-BeFS (Table 1), DD-4-BeFS outperforms DD-3-BeFS, and DD-6-BeFS outperforms DD-5-BeFS (Table 4).

Table 5: Directed diffusion case study: Time and space requirements (sum of the sizes of *AlreadyVisitedStates* and *NonVisitedStates*) and the number of events executed for finding Counterexample 1 in a N-node chain WSN using DD-4-BeFS-AC.

N	MAX_DEPTH	Time (s.)	Space (number of states)	Events
4	15	0.296	247	474
5	20	19.683	2289	27826
6	25	51.19	3604	51341
7	30	48.137	4503	51649
8	35	147.889	8506	110540
9	40	291.87	13839	172912
10	45	931.59	24105	347204
11	50	1667.782	33422	495318
12	55	4543.767	51666	867921

5 Related Work

The amount of research on verification and validation (V&V) of simulation models is large and considerably focuses on statistical techniques that compare the output of the simulation model with the output of the real system (e.g., the method of simultaneous confidence intervals [7]). In this section, we highlight some previous work that is closely related to ours. For further details on verification, validation, and testing of simulation models, the interested reader is referred to [5].

Model-checking Network Protocol Implementation Code Our work on verifying the computerized simulation model of a network protocol using state space exploration and protocol-specific properties is inspired by the previous work on model-checking network protocol implementation code directly for C and C++ (e.g., CMC [44, 45] and VeriSoft [28]). 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. In contrast, our approach uses *protocol-dependent* properties, inherent to the network protocol and the safety property being checked, to guide a best-first search strategy towards finding an assertion violation faster.

Likewise, VeriSoft uses *protocol-independent* techniques, namely partial-order reduction (POR) using the persistent and sleep sets [28]. POR is an approach towards alleviating the state space explosion problem, and aims at reducing the size of the state space by exploiting the independence relation between events. Independent events can neither disable nor enable each other, and enabled independent events commute; i.e., result in the same state when executed in different orders. Traditional algorithms for computing persistent sets exploit information that is typically inferred by a *static* analysis of the code. However, as pointed out in [27], these algorithms suffer from a fundamental limitation, namely determining this information with acceptable precision, in the context of concurrent software systems executing arbitrary code, is often difficult or impossible. In an attempt towards avoiding this inherent imprecision of static analysis, *dynamic* POR [27] was proposed. The algorithm, which is called *dynamic partial-order reduction*, dynamically tracks interactions between processes and then exploits this information to identify backtracking points where alternative paths in the state space need to be explored. In principle, dynamic POR can be combined with both shuffling and best-first search strategies. Specifically, POR first determines which events to explore, a shuffling procedure then shuffles the sequence in which those events are executed, and a best-first search strategy finally determines which of the successor states being generated can potentially lead to an assertion violation faster.

The idea of using best-first search heuristics to expedite the model checking process has been explored in previous work (e.g., [24, 29, 30, 65, 76]). However, what distinguishes our work from the previous work is that we study the use of protocol-specific heuristics in verifying the simulation model. Moreover, we focus on a specific domain, namely routing and data dissemination 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 assertion violations in less time and space requirements than classic breadth-first and depth-first search strategies.

Conventional Explicit-State and Symbolic Model Checking In contrast to model-checking the implementation code (or the computerized simulation model) of a network protocol directly, conventional explicit-state and symbolic model checkers (e.g., SPIN [32], SMV [43], SAL [68], Murphi [21]) require that the system be first specified using a high-level modeling language. For example, [52] presents the simulation and verification of the priority-ceiling protocol (PCP) using SAL. In general, the process of describing the system in a high-level modeling language is time-consuming, painstaking, and error-prone [45]. To deal with this problem, there has been some work (e.g., [19, 26, 31, 49]) on translating programming languages (e.g., Java) to 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., memory allocation and bit operations) do not have corresponding ones in the destination modeling language. Therefore, our approach of directly verifying the simulation model, which has to be written by the user anyway for the purpose of performance evaluation, reduces the user's effort and avoids the limitations of the input languages of conventional model checkers.

Formal Analysis of Network Simulation As far as formal analysis of network simulation is concerned, Verisim [13] was developed based on a collection of pre-existing tools, namely ns-2 [67] and the MaC monitoring and checking framework [37]. Verisim replaces the monitor component of MaC by ns-2 and uses the checker component of MaC to verify user-defined properties on *traces* produced by ns-2. It should be noted, however, that not all assertion violations may manifest themselves in a trace because ns-2 does not explore all possible execution paths during a simulation run. Towards giving formal semantics to simulation models and hence enabling the exploration of several execution paths, a translation from the DEVS (Discrete Event System Specification) [77] modeling paradigm to the Z-DEVS formalism, which combines DEVS, the Z specification language [63] and first order logic, is proposed in [69]. (This is similar to translating programming languages to the input modeling languages of model checkers that was discussed above.) The static properties of the simulation model can then be analyzed with the Z/EVES theorem prover [53] while the temporal properties can be analyzed using a model checker.

Neural Network/Machine Learning Approaches for Validating Simulation Models In [40], a neural network approach to the validation of simulation models is presented. Specifically, a number of alternative simulation models train a neural network using multiple statistics (e.g., means, variances, autocovariances, etc.). Hence, the neural network learns to identify key features of these statistics to belong to a specific simulation model. Following that, an experiment with the real system is offered to the neural network. The network then outputs a probability vector, indicating for every simulation model the probability that the data comes from the model. This probability vector can be used to distinguish valid from invalid models. Another machine learning technique can be found in [41] where Martens et al. make use of concepts from machine learning and fuzzy set theory to define a resemblance relation for measuring the degree of similarity between the input-output transformation of a simulation model and the corresponding input-output transformation of the real system.

Integrated Environments for Verifying Models Examples of other software tools that provide an integrated environment, which allows the user to both verify a model and use it to evaluate/predict performance, are TwoTowers [11] and Maude [17, 42].

Using TwoTowers, the user models a concurrent system as an algebraic term in $EMPA_r$ [10], which is an extension of $EMPA$ (Extended Markovian Process Algebra) [12] allowing for the specification of performance measures based on rewards. TwoTowers has three kernels: (a) an integrated kernel can simulate an $EMPA_r$ specification and derive performance measures according to the well-known method of independent replications, (b) a functional kernel generates a labeled transition system (LTS) of the $EMPA_r$ specification, and (c) a performance kernel generates a Markov chain of the $EMPA_r$ specification. A version of the Concurrency Workbench of North Carolina (CWB-NC) [18] can then analyze the LTS using different types of formal verification (e.g., model checking to check temporal properties and state space exploration to check safety properties). A Markov Chain Analyzer (MarCA) [64] can conduct stationary and transient performance analysis on the Markov chain where the performance measures are derived using the rewards expressed in the $EMPA_r$ specification. TwoTowers has been used for analyzing several distributed algorithms and networking protocols such as the alternating bit protocol, CSMA/CD, ATM switches, and QoS protocols for Internet audio [1].

Maude [42] is a reflective language and system that supports both equational and rewriting logic specification and programming. Maude can be used to create *executable* specifications for a wide range of applications (e.g., other languages, theorem provers, concurrent systems). In fact, Maude can be used to build language extensions for Maude itself. Particularly, Full Maude is implemented in Maude as an extension of Core Maude[14] [17]. Concurrent object-oriented systems can be specified in Full Maude by means of object-oriented modules, which support objects, messages, classes and inheritance. Object-oriented modules can then be executed and also model-checked with an on-the-fly explicit-state Linear Temporal Logic (LTL) model checker [25]. Furthermore, Real-Time Maude [47] is a language and tool supporting the formal specification and analysis of real-time and hybrid systems. Real-Time Maude is implemented in Maude as an extension of Full Maude and offers a wide range of analysis techniques, including timed rewriting for simulation purposes, state space exploration to check safety properties, and time-bounded LTL model checking. The Real-Time Maude LTL model checker has been previously used in [46] for verifying the AER/NCA active network protocol suite [36]. In a more recent case study, Ölveczky and Thorvaldsen [48] use Real-Time Maude to formally model the *Optimally Geographical Density Control (OGDC)* algorithm [78] for wireless sensor networks, use the Real-Time Maude specification to perform all the simulation experiments done by the algorithm developers using ns-2 [67], and perform further formal analyses which are beyond the capabilities of (traditional) simulation tools. The major difference between our work and tools such as TwoTowers and Maude is that we verify a simulation model that is written in an imperative language (namely, Java) rather than a model that is written in a formal specification language.

6 Conclusions and Future Work

In this paper, we present the design, implementation and performance evaluation of a state space exploration framework that enriches the set of verification and validation tools available to network simulation modelers and protocol designers. Our framework uses state space exploration to explore the (entire) state space created by a network simulation model and check whether the model satisfies certain safety properties that the real network protocol satisfies. We make use of structural properties in

[14]Core Maude is the Maude interpreter implemented in C++ and provides all of Maude's basic functionality.

the underlying state space of the protocol along two orthogonal dimensions; the first uses a non-trivial *simulation relation* to prune the states to be searched, and the second is *state ranking* that determines whether a state is "better than" another in order to enable the implementation of a best-first search (BeFS).

We demonstrate the effectiveness of our framework by verifying the simulation models of two widely used and fairly complex network protocols: the Ad-Hoc On-Demand Distance Vector (AODV) routing protocol for wireless ad hoc networks and the directed diffusion data dissemination protocol for wireless sensor networks. Experimental results show that the state space explorer is able to find violations of a safety property within acceptable time and space requirements. Furthermore, BeFS heuristics that exploit *properties inherent to the network protocol and the safety property being checked* can expedite the process of finding those violations by several orders of magnitude. We study several BeFS heuristics for both AODV and directed diffusion, and provide guidelines for good heuristics based on our results. We also compare the performance of our framework to that of a state-of-the-art model checker for Java programs, namely Java PathFinder (JPF). The results of the comparison show that our framework is comparable to JPF in terms of the time needed to find an assertion violation.

In future work, we intend to devise efficient heuristics for each class of network protocols (e.g., routing protocols, MAC protocols, and transport protocols, etc.) so that if the simulation model of a protocol belonging to a certain class is to be verified, the user can use the appropriate heuristic for that class without having to start from scratch. In other words, the heuristics will be class-specific instead of protocol-specific. Another avenue of future research is state similarity. In this paper, we have used only one way of defining similar states. In future work, we intend to study the performance of different granularities of state similarity.

References

[1] A. Aldini, R. Gorrieri, M. Roccetti, and M. Bernardo. Comparing the QoS of Internet audio mechanisms via formal methods. *ACM Trans. on Modeling and Computer Simulation*, 11(1):1–42, January 2001.

[2] O. Balci. Quality assessment, verification, and validation of modeling and simulation applications. In *Proc. of the 2004 Winter Simulation Conference.*

[3] O. Balci. Verification, validation, and certification of modeling and simulation applications. In *Proc. of the 2003 Winter Simulation Conference.*

[4] O. Balci. Principles of simulation model validation, verification, and testing. *Transactions of the Society for Computer Simulation International*, 14(1):3–12, 1997.

[5] O. Balci. Verification, validation, and testing. In *The Handbook of Simulation*, pages 335–393. ed. J. Banks. New York, NY: John Wiley & Sons, 1998.

[6] O. Balci, R. E. Nance, J. D. Arthur, and W. F. Ormsby. Expanding our horizons in verification, validation, and accreditation research and practice. In *Proc. of the 2002 Winter Simulation Conference.*

[7] O. Balci and R. G. Sargent. Validation of simulation models via simultaneous confidence intervals. *American Journal of Mathematical and Management Sciences*, 4(3-4):375–406, 1984.

[8] J. Banks, J. S. Carson II, B. L. Nelson, and D. M. Nicol. *Discrete-event System Simulation.* Prentice Hall, Inc., 2005.

[9] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Trans. on Software Engineering and Methodology*, 1(4):355–398, 1992.

[10] M. Bernardo. An algebra-based method to associate rewards with EMPA terms. In *Proc. of ICALP'97.*

[11] M. Bernardo, W. R. Cleaveland, S. T. Sims, and W. J. Stewart. TwoTowers: A tool integrating functional and performance analysis of concurrent systems. In *Proc. of IFIP FORTE/PSTV'98.*

[12] M. Bernardo and R. Gorrieri. A tutorial on EMPA: A theory of concurrent processes with nondeterminism, priorities, probabilities and time. *Theoretical Computer Science*, 202:1–54, July 1998.

[13] K. Bhargavan, C. A. Gunter, M. Kim, I. Lee, D. Obradovic, O. Sokolsky, and M. Viswanathan. Verisim: Formal analysis of network simulations. *IEEE Trans. on Software Engineering*, 28(2):129–145, February 2002.

[14] K. Bhargavan, D. Obradovic, and C. A. Gunter. Formal verification of standards for distance vector routing protocols. *Journal of the ACM*, 49(4):538–576, July 2002.

[15] A. W. Brown, editor. *Component-Based Software Engineering.* IEEE Computer Society Press, Los Alamitos, CA, 1996.

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

[17] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. Maude 2.3 manual. January 2007.

[18] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In *Proc. of CAV'96.*

[19] 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.*

[20] M. d'Amorim, A. Sobeih, and D. Marinov. Optimized execution of deterministic blocks in Java PathFinder. In *Proc. ICFEM'06, Springer-Verlag* LNCS *4260.*

[21] 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.*

[22] M. B. Dwyer, S. Elbaum, S. Person, and R. Purandare. Parallel randomized state-space search. In *Proc. of ICSE'07.*

[23] M. B. Dwyer, S. Person, and S. Elbaum. Controlling factors in evaluating path-sensitive error detection techniques. In *Proc. of ACM FSE'06.*

[24] 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)*, 5(2-3):247–267, March 2004.

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

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

[27] C. Flanagan and P. Godefroid. Dynamic partial-order reduction for model checking software. In *Proc. of ACM POPL'05.*

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

[29] P. Godefroid and S. Khurshid. Exploring very large state spaces using genetic algorithms. In *Proc. of TACAS'02.*

[30] A. Groce and W. Visser. Heuristics for model checking java programs. *International Journal on Software Tools for Technology Transfer (STTT)*, 6(4):260–276, August 2004.

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

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

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

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

[35] Java PathFinder (JPF). http://javapathfinder.sourceforge.net/.

[36] S. K. Kasera, S. Bhattacharyya, M. Keaton, D. Kiwior, S. Zabele, J. Kurose, and D. Towsley. Scalable fair reliable multicast using active services. *IEEE Network Magazine*, 14(1):48–57, January-February 2000.

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

[38] D. E. Knuth. *The art of computer programming, volume 2: seminumerical algorithms.* Addison-Wesley, 1998.

[39] A. M. Law. How to build valid and credible simulation models. In *Proc. of the 2006 Winter Simulation Conference.*

[40] J. Martens, K. Pauwels, and F. Put. A neural network approach to the validation of simulation models. In *Proc. of the 2006 Winter Simulation Conference.*

[41] J. Martens, F. Put, and E. Kerre. A fuzzy set theoretic approach to validate simulation models. *ACM Transactions on Modeling and Computer Simulation*, 16(4):375–398, October 2006.

[42] Maude. http://maude.cs.uiuc.edu.

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

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

[45] 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.*

[46] P. Ölveczky, 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.*

[47] P. Ölveczky and J. Meseguer. Specification and analysis of real-time systems using Real-Time Maude. In *Proc. of FASE'04.*

[48] P. Ölveczky and S. Thorvaldsen. Formal modeling and analysis of wireless sensor network algorithms in Real-Time Maude. In *Proc. of International Workshop on Parallel and Distributed Real-Time Systems 2006, held in conjunction with IEEE IPDPS'06.*

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

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

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

[52] H. Rueß and L. de Moura. From simulation to verification (and back). In *Proc. of the 2003 Winter Simulation Conference.*

[53] M. Saaltink. The z/eves system. In *ZUM '97: Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation*, 1997.

[54] 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 (poster). In *ACM MobiCom'04.*

[55] R. G. Sargent. Verification and validation of simulation models. In *Proc. of the 2005 Winter Simulation Conference.*

[56] A. Sobeih, W.-P. Chen, J. C. Hou, L.-C. Kung, N. Li, H. Lim, H.-Y. Tyan, and H. Zhang. J-Sim: A simulation environment for wireless sensor networks. In *Proc. of the Annual Simulation Symposium (ANSS 2005), part of the 2005 Spring Simulation Multiconference (SpringSim 2005).*

[57] A. Sobeih, W.-P. Chen, J. C. Hou, L.-C. Kung, N. Li, H. Lim, H.-Y. Tyan, and H. Zhang. J-Sim: A simulation and emulation environment for wireless sensor networks. *IEEE Wireless Communications Magazine*, 13(4):104–119, August 2006.

[58] A. Sobeih and J. C. Hou. A simulation framework for sensor networks in J-Sim. Technical Report UIUCDCS-R-2003-2386, Department of Computer Science, University of Illinois at Urbana-Champaign, November 2003.

[59] 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.*

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

[61] A. Sobeih, M. Viswanathan, and J. C. Hou. Bounded model checking of network protocols in network simulators by exploiting protocol-specific heuristics. Technical Report UIUCDCS-R-2005-2547, Department of Computer Science, University of Illinois at Urbana-Champaign, April 2005.

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

[63] M. Spivey. *The Z Notation: A Reference Manual.* Prentice Hall International Series in Computer Science, 1992.

[64] W. J. Stewart. *Introduction to the Numerical Solution of Markov Chains.* Princeton University Press, 1994.

[65] 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.*

[66] Java Native Interface: Programmer's Guide and Specification. Online book. http://java.sun.com/docs/books/jni/.

[67] The Network Simulator ns 2. http://www.isi.edu/nsnam/ns/.

[68] Symbolic Analysis Laboratory. http://sal.csl.sri.com/.

[69] M. K. Traore. Analyzing static and temporal properties of simulation models. In *Proc. of the 2006 Winter Simulation Conference.*

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

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

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

[73] W. Visser, C. S. Pasareanu, and S. Khurshid. Test input generation with Java PathFinder. In *Proc. of ACM ISSTA'04.*

[74] W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for red-black trees using abstraction. In *Proc. of IEEE/ACM ASE'05.*

[75] W. Visser, C. S. Pasareanu, and R. Pelanek. Test input generation for Java containers using state matching. In *Proc. of ACM ISSTA'06.*

[76] C. H. Yang and D. L. Dill. Validation with guided search of the state space. In *Proc. of ACM/IEE DAC'98.*

[77] B. P. Zeigler, T. G. Kim, and H. Praehofer. *Theory of Modeling and Simulation.* Academic Press, 2^{nd} edition, 2000.

[78] H. Zhang and J. C. Hou. Maintaining sensing coverage and connectivity in large sensor networks. *Wireless Ad Hoc and Sensor Networks: An International Journal*, 1(1-2):89–123, January 2005.