Incremental State Space Exploration in J-Sim

Ahmed Sobeih and Steven Lauterburg

Department of Computer Science
University of Illinois at Urbana Champaign
201 North Goodwin Avenue
Urbana, IL 61801
{*sobeih, slauter2*}*@cs.uiuc.edu*

Abstract

In this report, we present an incremental state space exploration technique that aims to provide a speedup in exploring the state space created by the execution of the simulation model of a network protocol for the purpose of verifying the model. We analytically obtain necessary conditions for the incremental state space exploration technique to provide a speedup in state space exploration time when compared to a traditional (non-incremental) state space exploration technique. We have implemented the incremental state space exploration technique in the J-Sim state space explorer. We provide three case studies for the simulation models of three network protocols: (a) Ad-Hoc On-Demand Distance Vector (AODV) routing protocol for wireless ad hoc networks, (b) directed diffusion protocol for wireless sensor networks, and (c) Automatic Repeat reQuest (ARQ). We study scenarios in which code changes may or may not lead to behavioral changes.[1]

1 Non-incremental state space exploration procedure

Figure 1 shows the pseudo-code of the non-incremental state space exploration procedure SSExplore(). The two major data structures in SSExplore() are *ToBeExplored* (which stores the states from which no transition has been explored yet) and *AlreadyVisited* (which stores the hash codes of the states that have already been visited). Figure 1 presents an explicit-state stateful search that avoids visiting a state s_1 if another state s_2, having the same hash code as s_1, has already been visited before. SSExplore() interacts with three instances of a class called *GlobalState*, which implements the definition of the state of the network protocol. These three instances are *initialState* (the initial state), *currentState* (the current state being explored) and *nextState* (one of the possible successors of the current state).

Initially, *AlreadyVisited* contains the hash code of the initial state only (Figure 1, line 3) and *ToBeExplored* contains the initial state only (Figure 1, line 4). As long as *ToBeExplored* is not empty (Figure 1, line 5), SSExplore() removes a state from *ToBeExplored* and sets *currentState* to it (Figure 1, line 6). For each state being explored (*currentState*), SSExplore() determines the events that are enabled in *currentState* by invoking *GenerateEnabledEvents()* (Figure 1, line 7). In *GenerateEnabledEvents()*, the enabling function (Figure 1, line 26) returns the number of possible successor states for each event (zero if the event is disabled). *GenerateEnabledEvents()* returns *EnabledEvents*, which is a list of enabled events (Figure 1, line 29). Each entry in *EnabledEvents* stores the corresponding event information *EventInfo* (Figure 1, line 28). Specifically, let each protocol entity have a unique ID n (Figure 1, line 24), each event have a unique ID e (Figure 1, line 25), and each enabled event has a set of integer-valued parameters i

[1]This technical report shows detailed results for incremental state space exploration in J-Sim, and it is a part of a larger work [6] on incremental state space exploration, including evaluation in Java PathFinder (JPF).

```
 0. procedure SSExplore()
 1. initialState.depth = 0 ;
 2. initialState.computeHashCode() ;
 3. AlreadyVisited.add(initialState.HashCode()) ;
 4. ToBeExplored.add(initialState) ;
 5. while ( | ToBeExplored | > 0 ) {
 6.    currentState = ToBeExplored.remove() ;
 7.    EnabledEvents = GenerateEnabledEvents(currentState) ;
 8.    for ( int i = 0 ; i < EnabledEvents.size() ; i++ ) {
 9.       EventInfo E = EnabledEvents.get(i) ;
10.       nextState = GenerateNextState(currentState, E) ; /* X: execute a transition */
11.       nextState.setDepth(currentState.Depth() + 1) ;
12.       nextState.computeHashCode() ; /* H: compute a hash code */
13.       checkProperty = nextState.verifySafety() ;  /* Y: verify the safety property */
14.       if ( (checkProperty == false) AND (DoesCounterexampleContainEvent(nextState)) ) {
15.          Print("Counterexample ") ;
16.          printCounterexample(nextState) ;
17.          exit ;
18.       } else if ( (checkProperty) AND (nextState.depth < MAX_DEPTH) ) {
             /* v: safe transition; i.e., nextState satisfies the safety property. 1-v: unsafe transition. */
             /* d: deepest transition; i.e., depth of nextState == MAX_DEPTH. */
19.          if ( nextState.HashCode() does not exist in AlreadyVisited ) { /* A: search in AlreadyVisited */
                /* b: backward transition; i.e., nextState has already been visited. f=1-b-d: forward transition. */
20.             if ( search strategy is best-first ) nextState.computeBeFSTuple() ;  /* U: compute the BeFS tuple */
21.             AlreadyVisited.add(nextState.HashCode()) ;  /* R: add a hash code to AlreadyVisited */
22.             ToBeExplored.add(nextState) ;  /* N: add a state to ToBeExplored */
             }
          }
       }
    }

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

Figure 1: An explicit-state stateful state space exploration procedure. The symbols used in the comments are explained in Table 1.

where $0 \leq i \leq NumberOfNextStates - 1$ (Figure 1, line 27), then each instance of *EventInfo* stores n, e and i.

SSExplore() then generates the successor states (*nextState*) by calling the *GenerateNextState()* function (Figure 1, line 10) for each enabled event, which in turn invokes the event handler of that event. Following that, SSExplore() post-processes *nextState*. Specifically, SSExplore() computes the hash code of *nextState* (Figure 1, line 12) and then checks whether *nextState* violates a safety property (Figure 1, line 13). We here distinguish between two disjoint types of transitions:

1. *safe transition*: a transition that generates a *nextState* that does not violate any safety property.

2. *unsafe transition*: a transition that generates a *nextState* that violates a safety property.

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 1, line 14). (We have made use of this feature in our experiments in Sections 4-5.) 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 1, line 16), 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 (i.e., the case of a safe transition) and the depth of *nextState* in the state space graph is strictly less than a specified maximum depth MAX_DEPTH (Figure 1, line 18), SSExplore() checks whether *nextState* has been visited before by searching for the hash code of *nextState* in *AlreadyVisited* (Figure 1, line 19). We here distinguish between three disjoint types of transitions:

1. *forward transition*: a transition that generates a *nextState* whose depth is strictly less than MAX_DEPTH and that has not been visited before.
2. *backward transition*: a transition that generates a *nextState* whose depth is strictly less than MAX_DEPTH and that has already been visited before.
3. *deepest transition*: a transition that generates a *nextState* whose depth is equal to MAX_DEPTH.

Note that it is impossible that the depth of *nextState* is strictly greater than MAX_DEPTH. If the transition that generated *nextState* is a safe forward transition, *nextState* is added to *ToBeExplored* (Figure 1, line 22) in order to be explored later and its hash code is added to *AlreadyVisited* (Figure 1, line 21) to denote that it has been visited.

Depending on the order in which states are added to, and removed from, *ToBeExplored*, SSExplore() can employ breadth-first (BFS), depth-first (DFS) and best-first (BeFS) search strategies. A best-first search strategy is implemented by *state ranking*. Specifically, the user writes a function that assigns each state a BeFS tuple $< b_1, b_2 >$ (Figure 1, line 20) based on protocol-specific properties. The state space explorer then considers *ToBeExplored* as a priority queue in which states are ranked 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.

The performance of each of the three search strategies mentioned above depends on the order in which enabled events are added to the list of enabled events *EnabledEvents* (Figure 1, line 28). SSExplore() assumes a fixed search order; i.e., the order is the same each time the procedure executes. Specifically, the search order determined by the three for loops (Figure 1, lines 24-28) is: increasing order of protocol entity IDs n, increasing order of event IDs e and increasing order of event parameters i. However, it has been shown in [3] 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 SSExplore(). Similar to [2], randomization is achieved by shuffling the set of enabled events at each state being explored using a Fisher-Yates shuffling algorithm [5]. Hence, the order of enabled events in *EnabledEvents* is randomized each time the function *GenerateEnabledEvents()* executes (Figure 1, line 7). 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 randomized search strategies: BFS-SH, DFS-SH, BeFS-SH where "SH" stands for "Shuffle".

2 Incremental state space exploration procedure

Figure 2 shows the pseudo-code of IncrementalSSExplore(), the incremental version of the state space exploration procedure in Figure 1. The added or modified lines are shown in italic. In Figure 2, line 0, the two parameters *write* and *read* determine the mode of operation of the procedure.

If *write* is $false$ and *read* is $false$, it is easy to see that the operation of IncrementalSSExplore() is exactly the same as SSExplore() (Figure 1).

If *write* is *true*, IncrementalSSExplore() stores the state space graph in the $OUTPUT$ data structure (Figure 2, lines 41-43). $OUTPUT$ stores only the hash codes of the visited states that do *not* violate an assertion. Specifically, each executed safe transition $s \xrightarrow{E} s'$, where E is an instance of *EventInfo* and both s and s' are instances of *GlobalState*, is represented in $OUTPUT$ as $< h, E, h' >$ where h is the hash code of s and h' is the hash code of s'. Before IncrementalSSExplore() terminates, the $OUTPUT$ data structure is written to an output file (Figure 2, line 34 and line 44). Note that we assume that no distinct states have the same hash code.

If *read* is *true*, IncrementalSSExplore() first loads a state space graph from an input file and stores it in the $INPUT$ data structure (Figure 2, line 1). For each enabled event in *currentState*, IncrementalSSExplore() makes use of two decision variables *execute* and *postProcess* to determine the "workload" associated with this enabled event. If *execute* is $false$, this enabled event is not executed and hence no workload is associated with it. If *execute* is *true* (Figure 2, line 22), the event is executed and a *nextState* is generated (Figure 2, line 23). If *postProcess* is also *true* (Figure 2, line 25 and line 37), (i) the hash code of *nextState* has to be computed (Figure 2, line 26), (ii) IncrementalSSExplore() has to check whether *nextState* violates an assertion (Figure 2, line 27), and (iii) IncrementalSSExplore() may have to check whether the hash code of *nextState* exists in *AlreadyVisited* (Figure 2, line 37). Otherwise, none of these three operations need to be done.

The settings of *execute* and *postProcess* (Figure 2, line 37) are determined as follows. Initially, both are set to *true* (Figure 2, lines 11-12). IncrementalSSExplore() then checks whether the enabled event represents a non-modified event (Figure 2, line 14). If not, *execute* and *postProcess* remain *true* since the event has to be executed and the generated *nextState* has to be post-processed as explained in Section 1. On the other hand, if the enabled event represents a non-modified event, potential savings may be gained depending on whether or not information about this transition exists in $INPUT$. IncrementalSSExplore() searches for the hash code of *currentState* and the *EventInfo* E that corresponds to this transition in $INPUT$ (Figure 2, line 15). If it does not exist, then *execute* and *postProcess* remain *true* since this transition may generate a *nextState* that violates an assertion (i.e., the case of an unsafe transition) or a *nextState* that has not been visited before (i.e., the case of a safe forward transition). If it does exist, then *postProcess* is set to $false$ (Figure 2, line 16) because we can fetch the hash code of *nextState* from $INPUT$ (Figure 2, line 17) and we know that *nextState* does not violate an assertion because $INPUT$ does not store the hash codes of states that violate an assertion as explained above. Hence, we saved both the time of computing the hash code of *nextState* and the time of checking whether *nextState* violates an assertion.

Following that, IncrementalSSExplore() checks if the depth of *nextState* (which is one plus the depth of *currentState*) is equal to MAX_DEPTH. If so (i.e., the case of a safe deepest transition), *execute* is also set to $false$ because there is no need to execute an event that will generate a *nextState* that satisfies the assertion and will not be added to $ToBeExplored$. If not, IncrementalSSExplore() checks if the hash code of *nextState* exists in *AlreadyVisited* (Figure 2, line 20). If so (i.e., the case of a safe backward transition), *execute* is also set to $false$ for the same reason as in the former check.

Table 1: Notation used in this paper: a capital letter is used for the average time spent in an operation, and a small letter is used for the proportion of transitions that satisfy a certain condition.

X	Time to execute an enabled transition.
H	Time to compute the hash code of a generated state.
Y	Time to check whether or not a generated state satisfies the safety property.
v	Proportion of explored transitions that are safe. Hence, $1 - v$ is the proportion of explored transitions that are unsafe.
d	Proportion of explored transitions that are deepest.
b	Proportion of explored transitions that are backward.
f	Proportion of explored transitions that are forward. $f = 1 - b - d$
A	Time to search in *AlreadyVisited*.
U	Time to compute the BeFS tuple of a generated state. $U = 0$ if the search strategy is not best-first.
R	Time to add a hash code of a generated state to *AlreadyVisited*.
N	Time to add a generated state to *ToBeExplored*.
q	Proportion of enabled events that are modified. Hence, $1 - q$ is the proportion of enabled events that are not modified.
L	Time to search in $INPUT$.
p	Proportion of explored non-modified transitions whose information is found in $INPUT$. In other words, p is the proportion of searches in $INPUT$ that are successful.
F_{total}	Time to load a state space graph from the input file and store it in $INPUT$.
κ	Total number of explored transitions.
F	$\frac{F_{total}}{\kappa}$

It should be mentioned that if both $read = true$ and $write = true$, we use only one data structure for both $INPUT$ and $OUTPUT$ instead of two separate data structures. This design choice saves the time of inserting in $OUTPUT$ the transition information that already exists in $INPUT$. This explains why the conditions in the if statements (Figure 2, lines 41-43) include the condition $postProcess == true$.

3 Analysis

3.1 Analysis of the non-incremental state space exploration procedure

In this section, we analyze the average time T_{NonInc} spent in an iteration of the for loop in which enabled events get executed and the generated states are post-processed (Figure 1, lines 8-22). We focus on the operations that would help us compare between the non-incremental and the incremental state space exploration procedures. Table 1 shows the notations that we use in this paper. Some of the comments in Figure 1 make use of the symbols shown in Table 1. Analyzing Figure 1, we see that

$$T_{NonInc} = X + H + Y + vbA + vf(A + U + R + N)$$

Note that $b + f = 1 - d$. Hence, we have,

$$T_{NonInc} = X + H + Y + v(1 - d)A + vf(U + R + N)$$

3.2 Analysis of the incremental state space exploration procedure

In this section, we analyze the average time T_{Inc} spent in an iteration of the for loop in which enabled events may or may not be executed and the generated states may or may not be post-processed (Figure 2, lines 9-43). Again, we focus on the operations that would help us compare between the non-incremental and the incremental state space exploration procedures. The purpose of this analysis is to identify the necessary conditions for the incremental state space exploration procedure to provide a speedup in the state space exploration time. First we note that an obvious overhead of IncrementalSS-Explore() is the time F_{total} spent in loading the state space graph from the input file and storing it in $INPUT$ (Figure 2, line 1). Since we are interested in analyzing the average time spent in an iteration of the for loop (Figure 2, lines 9-43), we work with $F = \frac{F_{total}}{\kappa}$ where κ is the total number of explored transitions.

Obviously, if $read$ is $false$, the incremental state space exploration technique is disabled, and hence no speedup is expected. Furthermore, if $q = 1$ (i.e., all enabled events are modified) then the condition in the if statement (Figure 2, line 14) will always be false, and no speedup will be expected from the incremental state space exploration procedure. Similarly, if $p = 0$ (i.e., no information about the explored non-modified transitions is found in $INPUT$) then the condition in the if statement (Figure 2, line 15) will always be false, and no speedup will be expected from the incremental state space exploration procedure either. Therefore, $read = true$, $q \neq 1$ and $p \neq 0$ are three obvious necessary conditions. Another trivial necessary condition is $v \neq 0$.

Analyzing Figure 2, we see that the average time T_{Inc} spent in an iteration of the for loop (Figure 2, lines 9-43) is given by:

$$T_{Inc} = F + qT_{NonInc} + (1-q)\{L + (1-p)T_{NonInc} + pbA + pf[A + X + U + R + N]\}$$

In order to have a speedup, we must have $T_{Inc} < T_{NonInc}$. In other words,

$$F + qT_{NonInc} + (1-q)\{L + (1-p)T_{NonInc} + p(1-d)A + pf(X + U + R + N)\} < T_{NonInc}$$

If $q \neq 1$, we have

$$\frac{F}{1-q} + L + (1-p)T_{NonInc} + p(1-d)A + pfX + pf(U + R + N) < T_{NonInc}$$

If $p \neq 0$, we have

$$\frac{F}{p(1-q)} + \frac{L}{p} + (1-d)A + fX + f(U + R + N) < T_{NonInc}$$

Recall that $T_{NonInc} = X + H + Y + v(1-d)A + vf(U + R + N)$. Hence, we have

$$\frac{F}{p(1-q)} + \frac{L}{p} + (1-d)A + fX + f(U + R + N) < X + H + Y + v(1-d)A + vf(U + R + N)$$

$$\frac{F}{p(1-q)} + \frac{L}{p} + (1-d)(1-v)A + f(1-v)(U + R + N) < H + Y + (1-f)X$$

Since in the state spaces of the sophisticated simulation models of network protocols, the proportion of explored transitions that are unsafe is usually significantly small, it is reasonable to assume that $v \approx 1$. This is especially true if the state space explorer searches for a state that violates the safety property and terminates the state space exploration once one is found. Therefore, and since we are only interested in a necessary condition and the two terms $(1-d)(1-v)A$ and $f(1-v)(U + R + N)$ are non-negative, if both A and $(U + R + N)$ are of the same order[2] as $\frac{F}{p(1-q)} + \frac{L}{p}$, we can drop the two terms $(1-d)(1-v)A$ and $f(1-v)(U + R + N)$ and get the following (simplified) necessary condition:

$$\frac{F}{p(1-q)} + \frac{L}{p} < H + Y + (1-f)X$$

[2]We will empirically validate this assumption in the evaluation section.

Informally, this condition means that incremental state space exploration cannot provide a speedup unless the costs of executing transitions, computing hash codes, and checking the safety properties are high and there is a small proportion of forward transitions. Furthermore, this condition also says that the larger the value of p and the smaller the value of q, the more likely it is for the incremental state space exploration to provide a speedup. In addition, this condition also suggests that a reduction in p (with q fixed) will have a more adversial effect on the incremental state space exploration than an increase in q (with p fixed). Intuitively, this is true because the overhead associated with a modified event is executing the transition and post-processing *nextState* while the overhead associated with a non-modified event whose transition information is not found in $INPUT$ is executing the transition, post-processing *nextState*, *and* the search in $INPUT$ (Figure 2, line 15).

In summary, the necessary conditions for IncrementalSSExplore() (Figure 2) to provide a speedup in state space exploration when compared with SSExplore() (Figure 1) are:

1. $read = true$
2. $q \neq 1$, $p \neq 0$, and $v \neq 0$
3. $\frac{F}{p(1-q)} + \frac{L}{p} < H + Y + (1-f)X$

The J-Sim state space explorer dynamically estimates the values of the symbols in Table 1 and checks whether or not the necessary conditions are satisfied. The user may run a sample run of IncrementalSSExplore() to know whether using the incremental state space exploration procedure can provide a speedup.

The analysis in this section assumed $write = false$. The case of $write = true$ can be done in a similar way to what we did above.

4 Evaluation and Results: AODV Routing

We apply the non-incremental and the incremental state space exploration procedures to the J-Sim simulation model of the Ad-Hoc On-Demand Distance Vector (AODV) routing protocol [9] for wireless ad hoc networks. In this section, we give an overview of AODV key functionality, describe the steps that we follow to verify its simulation model and present the results of this verification using the two techniques.

4.1 Overview of AODV

AODV 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) [8].

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., $hops_{n,d} > hops_{m,d}$.

4.2 Verifying the simulation model of AODV

States 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., $\mathit{nexthop}_{n,d} = m$. The loop-free property can be expressed as follows [1, 7]:

$$((\mathit{seqno}_{n,d} < \mathit{seqno}_{m,d}) \lor (\mathit{seqno}_{n,d} == \mathit{seqno}_{m,d} \land \mathit{hops}_{n,d} > \mathit{hops}_{m,d}))$$

The hash code of a state is computed by first constructing an integer array representation of the state. Following that, an integer-valued hash code of this integer array representation is computed using the Jenkins' hash function that we borrowed from JPF. (We slightly modified this function to return an integer instead of a long value.) The integer array representation of a state depends on the protocol-specific information such as the AODV packet headers and payloads, and each node's seqno_n, bid_n, routing table entries and broadcast ID cache entries.

Events Next, we specify the set of events, when each event is enabled and the corresponding *Enabling-Function()* (Figure 1, line 26), and how each event is handled. The events are listed as follows:

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 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_1*)* 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_2 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_2*)* returns 1. The event is handled by invalidating this RTE.

T_3 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_3*)* 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_4 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_4*)* 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.

T_5 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_5*)* always returns 1. The event is handled by reinitializing the state of the AODV process at node n.

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, $\ldots$, 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. In addition, the BeFS heuristic that we use considers a state s_1 better than a state s_2 if the number of valid RTEs *to the destination* n_{N-1} 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 n_{N-1} in s, and b_2 is the number of valid RTEs to any node in s. Although this network topology 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.

While verifying the J-Sim simulation model of AODV in this chain network topology, we manually injected two errors (which we call Counterexamples 1 and 2 respectively): in Counterexample 1, $seqno_{n,d}$ is not incremented when a RTE is invalidated and in Counterexample 2, a RTE is deleted (instead of invalidated) when its lifetime expires. The state space exploration framework was able to find these two errors.[3] A routing loop may occur due to either of these two errors. Consider for example a chain network topology consisting of $N = 3$ nodes. In the case that $nexthop_{0,2} = 1$ and a route timeout event takes place at node n_1, in either Counterexample 1 or 2, if n_1 is later offered a route to node n_2 by node n_0, this route will be accepted (because in Counterexample 1, $hops_{1,2} = \infty$; hence, $hops_{1,2} > hops_{0,2}$; whereas in Counterexample 2, $seqno_{0,2} > seqno_{1,2}$). The interested reader is referred to [14] for detailed traces (along with the explanations) of the two counterexamples.

We study the performance of the incremental state space exploration procedure in three different scenarios: (i) best-case scenario: no events are modified, (ii) practical-case scenarios: one event (the route timeout event T_2) or two events (T_2 and T_3), or three events (T_0, T_2 and T_3), or four events (T_0, T_1, T_2 and T_3) are modified, and (iii) worst-case scenario: all six events are modified. Furthermore, we compare between the non-incremental and the incremental state space exploration procedures in each of these three scenarios using the three randomized search strategies BFS-SH, DFS-SH, and BeFS-SH. To isolate the savings gained (or overhead incurred) by the incremental state space exploration procedure, the modified events, if any, are just "flagged" as modified but their implementations are not changed. Practically, this corresponds to code changes that do *not* incur any behavioral changes; e.g., a code refactoring where any changes to the code improve its readability or simplify its structure but do not change its results.

Table 2 gives the time needed to find an assertion violation (Counterexample 1 or 2) if any exists. In Table 2, "Non-incremental SSE" refers to Figure 2 with $read = false$ and $write = false$, while "Incremental SSE" refers to Figure 2 with $read = true$ and $write = false$. Before running the "Incremental SSE", we had to execute Figure 2 with $read = false$ and $write = true$ in order to generate the file that contains the state space graph; however, the time needed for this intermediate step is not reported. For each experiment, we ran 10 replications. Each replication has a different seed. The average time for incremental state space exploration is shown as a percentage of the corresponding average non-incremental state space exploration time. The last column of Table 2 shows f, which is the proportion of explored transitions that are forward.

As shown in Table 2, the incremental state space exploration technique can indeed provide savings

[3]In all the experiments of the AODV case study, we require that the counterexample contain at least one state that is generated due to the route timeout event, T_2. In order to achieve that, we made use of the *DoesCounterexampleContainEvent()* function provided by the state space exploration framework (Figures 1-2).

Table 2: AODV case study: Average time (sec.) for non-incremental state space exploration. The average time for incremental state space exploration is shown as a percentage of the corresponding average non-incremental state space exploration time. The ad hoc network consists of N nodes arranged in a chain network topology. The state space explorer terminates state space exploration if an assertion violation is detected. $p = 0.999$

Correct T_2 $N = 3$ $MAX\_DEPTH = 9$	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE ($read = true$, $write = false$) Time is shown as a percentage of non-incremental SSE time						
Search Strategy		No Modified Events (q=0)	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	All Modified Events (q=1)	f
BFS-SH	155.4061	34.91 %	36.21 %	59.4 %	69.38 %	77.02 %	108.66 %	0.0698
DFS-SH	165.5496	35.21 %	36.51 %	59.08 %	69.1 %	77.21 %	110.44 %	0.0704
BeFS-SH	169.6521	34.79 %	35.87 %	58.41 %	68.13 %	75.75 %	107.19 %	0.0706

Counterexample 1 $N = 3$ $MAX\_DEPTH = 9$	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE ($read = true$, $write = false$) Time is shown as a percentage of non-incremental SSE time						
Search Strategy		No Modified Events (q=0)	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	All Modified Events (q=1)	f
BFS-SH	77.6531	45.5 %	46.18 %	65.28 %	74.44 %	80.01 %	107.19 %	0.1635
DFS-SH	45.9464	36.43 %	37.45 %	59.72 %	69.67 %	77.95 %	109.59 %	0.0766
BeFS-SH	6.5160	42.67 %	44.08 %	62.16 %	71.46 %	79.32 %	105.76 %	0.2094

Counterexample 2 $N = 3$ $MAX\_DEPTH = 9$	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE ($read = true$, $write = false$) Time is shown as a percentage of non-incremental SSE time						
Search Strategy		No Modified Events (q=0)	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	All Modified Events (q=1)	f
BFS-SH	74.033	46.44 %	46.91 %	65.79 %	75.46 %	80.86 %	108.0 %	0.1687
DFS-SH	46.4601	36.38 %	37.62 %	59.11 %	68.78 %	77.38 %	108.77 %	0.0755
BeFS-SH	6.4103	42.61 %	43.9 %	62.96 %	72.06 %	79.48 %	104.94 %	0.2094

Counterexample 2 Search Strategy is BeFS-SH		Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE ($read = true$, $write = false$) Time is shown as a percentage of non-incremental SSE time						
N	MAX_DEPTH		No Modified Events (q=0)	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	All Modified Events (q=1)	f
3	15	0.6666	85.46 %	88.4 %	91.02 %	92.37 %	94.8 %	101.68 %	0.5492
4	20	1.8586	78.49 %	81.75 %	85.82 %	87.38 %	93.64 %	105.32 %	0.4832
5	25	17.7226	55.14 %	62.85 %	73.86 %	74.74 %	84.99 %	105.32 %	0.3991

in all the practical-case scenarios ($q < 1$). In the case of all events are modified ($q = 1$), an extra overhead is incurred because of reading the input file (Figure 2, line 1); hence, the incremental state space exploration technique is slower than the non-incremental technique because the necessary condition $q \neq 1$ is violated. Note that we implemented our own read and write functions because we noticed an unreasonably large overhead when we used Java serialization.

In order to understand why the incremental state space exploration procedure provided a speedup, we show in Table 3 a breakdown of the average state space exploration time spent in some selected operations taking the first row in Table 2 as an example. As shown in Table 3, the costs of executing transitions (i.e., the sum of copying from the verification model to the simulation model, executing events, and copying from the simulation model to the verification model), computing hash codes, and checking the safety property are considerably high taking together more than 76 % of the total average time in the non-incremental state space exploration. In contrast, the times spent in these three operations in the incremental state space exploration procedure ($q < 1$) are smaller than their counterparts in the non-incremental exploration. Furthermore, other operations that are only done in the incremental state space exploration procedure (e.g., reading from the input file and searching in $INPUT$) take a comparatively

Table 3: AODV case study: Breakdown of the average state space exploration time (sec.) spent in some selected operations. The example shown corresponds to the first row in Table 2. $p = 0.999$

Correct T_2 $N = 3$ $MAX\_DEPTH = 9$ Search Strategy is BFS-SH	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE ($read = true$, $write = false$) Time (sec.)					
Operation		No Modified Events (q=0)	One Modified Event (q=0.0108)	Two Modified Events (q=0.2856)	Three Modified Events (q=0.4143)	Four Modified Events (q=0.513)	All Modified Events (q=1)
Copy from V model to S model	14.7775	1.6033	1.4299	5.4716	7.0094	8.6549	15.1812
Execute events	50.6706	5.8223	6.929	22.6846	29.0423	33.0216	50.9855
Copy from S model to V model	13.6867	1.3049	1.5328	5.8027	7.0208	8.551	13.6322
Compute a hash code	31.439	0.0929	0.4651	11.1563	15.4838	18.5879	32.3965
Verify an assertion	8.1315	0.0014	0.124	2.4905	3.529	4.3993	8.5203
Read from input file	0	10.6054	10.742	10.5592	10.6757	10.5942	10.5032
Search in $INPUT$	0	3.2134	3.2405	2.6723	2.3589	2.1829	0
Insert in *AlreadyVisited*	4.8876	6.799	6.6914	5.7695	5.8479	5.633	5.2587
Search in *AlreadyVisited*	1.1439	0.9103	0.8977	0.965	1.0453	1.0436	1.1737
Enabling function	6.7246	7.5667	7.7964	7.0842	6.8388	7.0931	6.9953
Insert in *ToBeExplored*	0.1575	0.194	0.1595	0.1623	0.1578	0.1632	0.1857
Compute the BeFS tuple	0	0	0	0	0	0	0
Subtotal	131.6189	38.1136	40.0083	74.8182	89.0097	99.9247	144.8323
Other Operations	23.7872	16.1468	16.2685	17.5077	18.8143	19.7792	24.0416
Total Time	155.4061	54.2604	56.2768	92.3259	107.824	119.7039	168.8739
Speedup		**2.86x**	**2.76x**	**1.68x**	**1.44x**	**1.3x**	**none**
Number of executed transitions	460150	32114	36228	151658	206985	247422	460150

small amount of time (between 6 % and 26 % of the incremental state space exploration time). Note that in this particular example, $f = 6.98\%$; i.e., the proportion of transitions that are forward is small. Furthermore, $p = 0.999$; i.e., almost all of the searches in $INPUT$ are successful. All these observations explain why incremental state space exploration procedure ($q < 1$) provided a speedup. The second-to-last row in Table 3 shows the value of this speedup, which is defined as the non-incrementatal state space exploration time divided by the incrementatal state space exploration time. The last row in Table 3 gives the number of the executed transitions. Note that although $f = 6.98\%$, the time of executing transitions in the incremental state space exploration with no modified events is larger than 6.98% of the time of executing transitions in the non-incremental state space exploration. We have found that the reason for this observation is that the average time of executing a forward transition is more than the average time of executing a backward or deepest transition.

It should also be noticed that the times of the operations that do not appear in the simplified necessary condition (e.g., inserting the hash code of a state in *AlreadyVisited*, inserting a state in *ToBeExplored*, searching in *AlreadyVisited*, and executing the enabling function) are almost equal in both the non-incremental and incremental state space exploration techniques. Furthermore, the sum of the times to search in *AlreadyVisited*, insert a hash code in *AlreadyVisited*, and insert a state in *ToBeExplored* is of the same order as the sum of the times to search in $INPUT$ and read from the input file in the incremental state space exploration technique. This justifies the assumption that we made in Section 3.2.

In Table 2, the state space explorer terminates state space exploration if an assertion violation is detected. Table 4 gives the results for Counterexamples 1 and 2 in the case where state space exploration does *not* terminate when an assertion violation is detected. Instead, the state space explorer continues exploring the state space till the maximum specified depth MAX_DEPTH, but does not explore transitions from those states that violate the assertion. The percentages in Table 4 are smaller than their counterparts in Table 2 because of the reduction in f, which is the proportion of explored transitions

Table 4: AODV case study: Average time (sec.) for non-incremental state space exploration. The average time for incremental state space exploration is shown as a percentage of the corresponding average non-incremental state space exploration time. The ad hoc network consists of N nodes arranged in a chain network topology. The state space explorer does NOT terminate state space exploration if an assertion violation is detected. $p = 0.999$

Counterexample 1 $N = 3$ $MAX\_DEPTH = 9$ Search Strategy	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE ($read = true$, $write = false$) Time is shown as a percentage of non-incremental SSE time No Modified Events (q=0)	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	All Modified Events (q=1)	f
BFS-SH	154.5043	35.25 %	36.08 %	60.08 %	70.52 %	77.64 %	108.66 %	0.0698
DFS-SH	166.5466	35.19 %	36.1 %	58.58 %	68.48 %	76.42 %	108.46 %	0.0704
BeFS-SH	166.9943	35.48 %	36.38 %	58.88 %	69.3 %	76.99 %	108.19 %	0.0706

Counterexample 2 $N = 3$ $MAX\_DEPTH = 9$ Search Strategy	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE ($read = true$, $write = false$) Time is shown as a percentage of non-incremental SSE time No Modified Events (q=0)	One Modified Event	Two Modified Events	Three Modified Events	Four Modified Events	All Modified Events (q=1)	f
BFS-SH	152.5928	34.93 %	35.86 %	59.97 %	70.06 %	77.18 %	108.46 %	0.0696
DFS-SH	164.3032	35.12 %	35.93 %	58.56 %	68.39 %	76.53 %	108.08 %	0.0703
BeFS-SH	165.6676	35.14 %	35.79 %	58.52 %	68.63 %	76.19 %	107.53 %	0.0705

that are forward, as indicated in the last column of Table 4. Intuitively, f is higher when the state space explorer terminates state space exploration if an assertion violation is detected (Table 2) because the forward transitions are the ones that "make change" and cause the assertion to be violated. This is especially true in the case of the BeFS-SH strategy (see Table 2) because the goal of the best-first search is to guide the state space explorer towards paths that lead to the assertion violation in less time. As indicated by the last row in Table 4, the incremental state space exploration technique was able to provide a 2.79x speedup with one code change (i.e., one modified event) that does not incur a behavioral change.

To further evaluate the incremental state space exploration technique, we evaluate its performance in another scenario; namely, one in which the implementation of a modified event *does* change. Practically, this corresponds to code changes that do incur behavioral changes. Specifically, we simulate the behavior of a user who tries to implement the route timeout event T_2 correctly. First, the user implements the route timeout event T_2 by deleting an RTE instead of invalidating it causing Counterexample 2 to occur (we call this implementation Version C), then the user figures out that the RTE has to be invalidated instead of being deleted but forgets to increment the destination sequence number causing Counterexample 1 to occur (we call this implementation Version B). Following that, the user figures out the correct implementation, which includes invalidating the RTE and incrementing the destination sequence number (we call this implementation Version A). Table 5 shows the state space exploration time under this scenario using both the non-incremental and incremental techniques. We also show the results for the case where the implementation changes from Version B, to C and finally to A. (In fact, we have also considered the case where the implementation changes from Version A, to B and finally to C, and the case where the implementation changes from Version A, to C and finally to B, but we do not report the results for these cases because they are similar to the results for the cases reported here.)

We distinguish between two cases: Case I, which we call WriteEachVersion, and Case II, which we call WriteFirstVersion. In Case I, storing the state space graph in the $OUTPUT$ data structure and writing the $OUTPUT$ data structure to an output file occurs in each version using the incremental state space exploration technique. Specifically, $read = false$ and $write = true$ in the first version while $read = true$ and $write = true$ in the second and third versions. On the other hand, in Case II, storing the state space graph in the $OUTPUT$ data structure and writing the $OUTPUT$ data structure to an output file occurs in only the first version using the incremental state space exploration technique. Specifically, $read = false$ and $write = true$ in the first version while $read = true$ and $write = false$ in the second and third versions.

As shown in Table 5, incremental state space exploration is able to provide up to 1.52x speedup in state space exploration time. The performance results in Case II are better than those in Case I because the second and third versions of Case II avoid the operations associated with $write = true$; namely, inserting transition information in $OUTPUT$ and writing the state space graph to the output file. Note that the values of p in Case II are very close to the corresponding ones in Case I; i.e., no significant harm was done by setting $write = false$ in the second and third versions of Case II.

In both Cases I and II, the performance results under the scenario B $\rightarrow$ C $\rightarrow$ A are very close to the corresponding ones under the scenario C $\rightarrow$ B $\rightarrow$ A. This is due to the observation that the values of p and q in the former scenario are very close to the corresponding ones in the latter.

It is also interesting to see how the speedup dropped from 2.76x (Table 3, case of one modified event) to 1.52x (Table 5, case of WriteFirstVersion) although the value of $q = 0.0108$ is the same in both cases. This reduction in the speedup is due to the reduction in the value of p from $p = 0.999$ (Table 3) to $p = 0.9404$ (Table 5).

As indicated by the last row in Table 5, the incremental state space exploration technique was able to provide a 1.52x speedup with one code change (i.e., one modified event) that does incur a behavioral change.

5 Evaluation and Results: Directed Diffusion

We apply the non-incremental and the incremental state space exploration procedures to the J-Sim simulation model of the Directed Diffusion [4] data dissemination protocol for wireless sensor networks. In this section, we give an overview of the key functionality of directed diffusion, describe the steps that we follow to verify its simulation model and present the results of this verification using the two techniques.

5.1 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 [10–12]. In this section, we describe the J-Sim simulation model of directed diffusion.

Directed diffusion [4] 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

Table 5: AODV case study: Time (sec.) for both the non-incremental and the incremental state space exploration techniques. The ad hoc network consists of 3 nodes arranged in a chain network topology. $MAX\_DEPTH = 9$. Search strategy is BFS-SH. The state space explorer does NOT terminate state space exploration if an assertion violation is detected.

Case I (WriteEachVersion): In incremental state space exploration: *read* and *write* are set as follows:
$read = false$ and $write = true$ (first version in each set).
$read = true$ and $write = true$ (second and third versions in each set). Same data structure is used for $INPUT$ and $OUTPUT$.

C → B → A	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE Time (sec.) (see caption of Case I)	Inc. / Non. Percentage	Observations on Incremental SSE
Version C (Counterexample 2)	152.7506	176.7867	115.7355 %	$v = 0.99996$. No speedup if $read = false$
Version B (Counterexample 1)	154.3521	64.9231	42.0617 %	$p = 0.9403$, $q = 0.0108$
Version A (Correct T_2)	155.4537	65.5253	42.151 %	$p = 0.9404$, $q = 0.0108$
Sum of Versions C, B, and A	462.5564	307.2351	66.4211 %	**1.5x overall speedup**

B → C → A	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE Time (sec.) (see caption of Case I)	Inc. / Non. Percentage	Observations on Incremental SSE
Version B (Counterexample 1)	154.3521	179.9178	116.5632 %	$v = 0.99997$. No speedup if $read = false$
Version C (Counterexample 2)	152.7506	62.1906	40.7138 %	$p = 0.9521$, $q = 0.0109$
Version A (Correct T_2)	155.4537	65.5781	42.185 %	$p = 0.9404$, $q = 0.0108$
Sum of Versions B, C, and A	462.5564	307.6865	66.5187 %	**1.5x overall speedup**

Case II (WriteFirstVersion): In incremental state space exploration: *read* and *write* are set as follows:
$read = false$ and $write = true$ (first version in each set).
$read = true$ and $write = false$ (second and third versions in each set).

C → B → A	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE Time (sec.) (see caption of Case II)	Inc. / Non. Percentage	Observations on Incremental SSE
Version C (Counterexample 2)	152.7506	179.0224	117.1991 %	$v = 0.99996$. No speedup if $read = false$
Version B (Counterexample 1)	154.3521	62.3558	40.3984 %	$p = 0.9403$, $q = 0.0108$
Version A (Correct T_2)	155.4537	62.3996	40.1403 %	$p = 0.9403$, $q = 0.0108$
Sum of Versions C, B, and A	462.5564	303.7778	65.6737 %	**1.52x overall speedup**

B → C → A	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE Time (sec.) (see caption of Case II)	Inc. / Non. Percentage	Observations on Incremental SSE
Version B (Counterexample 1)	154.3521	181.0826	117.3179 %	$v = 0.99997$. No speedup if $read = false$
Version C (Counterexample 2)	152.7506	60.0533	39.3146 %	$p = 0.9521$, $q = 0.0109$
Version A (Correct T_2)	155.4537	62.4221	40.1548 %	$p = 0.9404$, $q = 0.0108$
Sum of Versions B, C, and A	462.5564	303.558	65.6262 %	**1.52x overall speedup**

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. 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).

5.2 Verifying the simulation model of directed diffusion

States We use the same definitions of *GlobalState* and network cloud that were introduced in Section 4.2. 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:

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

Similar to what we did in Section 4.2, the hash code of a state is computed by first constructing an integer array representation of the state. Following that, an integer-valued hash code of this integer array representation is computed using the Jenkins' hash function that we borrowed from JPF. The integer array representation of a state depends on the protocol-specific information such as the packet headers and payloads, and each node's interest cache entries and data cache entries.

Events Next, we specify the set of events, when each event is enabled and the corresponding *EnablingFunction()*, and how each event is handled.

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[4] 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.

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.

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). Since a reinforced gradient is established upon receiving a positive reinforcement packet, the BeFS heuristic that we use 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. Furthermore, 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.

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

[4]For practical reasons, previously received DATA packets can not be kept in the data cache for an indefinitely long time; otherwise, the size of the data cache can increase arbitrarily. In the 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.

(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).[5] The interested reader is referred to [15] 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.

We study the performance of the incremental state space exploration procedure in two different scenarios: (i) best-case scenario: no events are modified, and (ii) practical-case scenarios: one event (the packet delivery event T_4) or two events (T_2 and T_4) are modified, Furthermore, we compare between the non-incremental and the incremental state space exploration procedures in each of these two scenarios using the randomized search strategy BeFS-SH. To isolate the savings gained (or overhead incurred) by the incremental state space exploration procedure, the modified events, if any, are just "flagged" as modified but their implementations are not changed. Again, this corresponds to code changes that do *not* incur any behavioral changes.

Table 6 gives the time needed to find an assertion violation (Counterexample 1 or 2) for different values of N and MAX_DEPTH. In Table 6, "Non-incremental SSE" refers to Figure 2 with $read = false$ and $write = false$, while "Incremental SSE" refers to Figure 2 with $read = true$ and $write = false$. Before running the "Incremental SSE", we had to execute Figure 2 with $read = false$ and $write = true$ in order to generate the file that contains the state space graph; however, the time needed for this intermediate step is not reported. For each experiment, we ran 10 replications. Each replication has a different seed. The average time for incremental state space exploration is shown as a percentage of the corresponding average non-incremental state space exploration time. As shown in Table 6, the incremental state space exploration technique can indeed provide savings in the two practical-case scenarios. For example, as indicated by the second-to-last row in Table 6, the incremental state space exploration technique was able to provide a 1.86x speedup with one code change (i.e., one modified event) that does not incur a behavioral change.

In order to understand why the incremental state space exploration procedure provided a speedup, we show in Table 7 a breakdown of the average state space exploration time spent in some selected operations taking the last row in Table 6 as an example. As shown in Table 7, the costs of executing transitions (i.e., the sum of copying from the verification model to the simulation model, executing events, and copying from the simulation model to the verification model), computing hash codes, and checking the safety properties are considerably high taking together more than 71 % of the total average time

[5] 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 6: Directed diffusion case study: Average time (sec.) for non-incremental state space exploration. The average time for incremental state space exploration is shown as a percentage of the corresponding average non-incremental state space exploration time. The wireless sensor network consists of N nodes arranged in a chain network topology. The state space explorer terminates state space exploration if an assertion violation is detected.

Search Strategy is BeFS-SH			Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE ($read = true$, $write = false$) Time is shown as a percentage of non-incremental SSE time			
N	MAX_DEPTH	Counterexample		No Modified Events (q=0)	One Modified Event	Two Modified Events	f
4	15	1	6.1485	49.42 %	70.0 %	84.18 %	0.1888
5	20	1	15.3852	40.62 %	57.85 %	72.85 %	0.1927
6	25	1	63.4459	35.2 %	53.84 %	71.37 %	0.1271
4	20	2	198.9493	37.51 %	62.8 %	73.42 %	0.0972

in the non-incremental state space exploration. In contrast, the times spent in these three operations in the incremental state space exploration procedure are smaller than their counterparts in the non-incremental exploration. Furthermore, other operations that are only done in the incremental state space exploration procedure (e.g., reading from the input file and searching in $INPUT$) take a comparatively small amount of time (between 7 % and 18 % of the incremental state space exploration time). Note that in this particular example, $f = 9.72\%$; i.e., the proportion of transitions that are forward is small. Furthermore, $p = 0.999$; i.e., almost all of the searches in $INPUT$ are successful. All these observations explain why incremental state space exploration procedure provided a speedup. It should also be noticed that the times of the operations that do not appear in the simplified necessary condition (e.g., inserting the hash code of a state in *AlreadyVisited*, inserting a state in *ToBeExplored*, searching in *AlreadyVisited*, computing the BeFS tuple, and executing the enabling function) are almost equal in both the non-incremental and incremental state space exploration techniques. Furthermore, the sum of the times to search in *AlreadyVisited*, compute the BeFS tuple, insert a hash code in *AlreadyVisited*, and insert a state in *ToBeExplored* is of the same order as the sum of the times to search in $INPUT$ and read from the input file in the incremental state space exploration technique. This justifies the assumption that we made in Section 3.2.

To further evaluate the incremental state space exploration technique, we evaluate its performance in another scenario; namely, one in which the implementation of a new event is added. Similar to Section 4.2, this corresponds to code changes that do incur behavioral changes. However, in Section 4.2, the change was a modification of an existing behavior, but in this section, the change is an addition of a new behavior. We study four examples that are explained in Table 8. As indicated by Example 2 in Table 8, the incremental state space exploration technique was able to provide a 1.25x speedup with one code change (i.e., one added event) that does incur a behavioral change.

6 Evaluation and Results: Automatic Repeat reQuest (ARQ)

We apply the non-incremental and the incremental state space exploration procedures to the J-Sim simulation model of an Automatic Repeat reQuest (ARQ) protocol. In this section, we give an overview of ARQ key functionality, describe the steps that we follow to verify its simulation model and present the results of this verification using the two techniques.

Table 7: Directed diffusion case study: Breakdown of the average state space exploration time (sec.) spent in some selected operations. The example shown corresponds to the last row in Table 6. $p = 0.9999$

Counterexample 2 $N = 4$ $MAX\_DEPTH = 20$ Search Strategy is BeFS-SH Operation	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE ($read = true$, $write = false$) Time (sec.) No Modified Events (q=0)	One Modified Event (q=0.2919)	Two Modified Events (q=0.4546)
Copy from V model to S model	19.2949	2.3997	8.696	10.5929
Execute events	65.2143	8.6028	33.9594	41.546
Copy from S model to V model	17.7139	3.1262	8.1014	10.9446
Compute a hash code	17.4309	0.0181	6.4203	9.2332
Verify an assertion	23.1073	0.0007	7.7124	12.1033
Read from input file	0	7.1539	7.0479	7.1657
Search in $INPUT$	0	5.7184	4.2303	3.3842
Insert in *AlreadyVisited*	6.2191	10.5322	7.7189	7.2765
Search in *AlreadyVisited*	3.1925	3.046	3.0927	3.0758
Enabling function	7.0964	8.2709	7.8069	7.8257
Insert in *ToBeExplored*	0.5372	0.5134	0.5415	0.5186
Compute the BeFS tuple	0.9162	0.9344	0.9346	0.9456
Subtotal	160.7227	50.3167	96.2623	114.6121
Other Operations	38.2266	24.3193	28.6791	31.4692
Total Time	198.9493	74.636	124.9414	146.0813
Speedup		**2.67x**	**1.59x**	**1.36x**
Number of executed transitions	1293732	124492	491101	649234

6.1 Overview of ARQ

Automatic Repeat reQuest (ARQ) is a well-known error control protocol that has several variations. The simplest form of the ARQ protocol is stop-and-wait ARQ in which the sender sends a single data packet and then waits for a positive acknowledgment (ACK) before it advances to the next data packet. The receiver replies with an ACK if the data packet is correctly received. As either the data packet or the corresponding ACK may be lost/corrupted in transit, after the sender sends a data packet, it sets a retransmission timer. If no ACK is received before the retransmission timer expires, the sender retransmits the data packet. It should be mentioned that the setting of the timeout interval at the sender is very important, and is a trade-off between premature timeout and prolonged retransmission.

For the receiver to distinguish between a data packet that is sent for the first time and a retransmission, a sequence number is included in the header of each data packet. For stop-and-wait ARQ, it is sufficient that the sequence number be 1-bit (i.e., either 0 or 1) because the only ambiguity is between a data packet and its immediate predecessor and successor, but not between the predecessor and successor themselves [16]. For similar reasons, each ACK should also contain a sequence number. In the common practice, the sequence number in the ACK is the sequence number of the next expected data packet rather than the sequence number of the data packet that has been recently received. If the receiver receives a data packet whose sequence number is not equal to the sequence number it is expecting, the receiver discards this duplicate data packet and retransmits an ACK announcing the sequence number of the next expected data packet. Upon receiving the ACK, the sender checks the sequence number in the ACK to determine whether a new data packet or a retransmission should be sent.

Stop-and-wait ARQ ensures that every data packet sent by the sender will eventually be received correctly by the receiver and that the receiver will get the data packets in order, i.e., it is an in-order reliable unicast protocol.

Table 8: Directed diffusion case study: Time (sec.) for both the non-incremental and the incremental state space exploration techniques. The wireless sensor network consists of 4 nodes arranged in a chain network topology. $MAX\_DEPTH = 10$. Search strategy is BFS-SH. No assertion violations.

	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE Time (sec.) (Write strategy is WriteFirstVersion)	Inc. / Non. Percentage	Observations on Incremental SSE
Example 1: Version B_1 includes the events T_0, T_1, T_3, T_4, and T_5. Version A_1 adds T_2.				
Version B_1	23.573	25.648	108.8024 %	
Version A_1	33.387	20.23	60.5924 %	p = 0.7505, q = 0.1321
Sum of Versions B_1 and A_1	56.96	45.878	80.5442 %	**1.24x overall speedup**
Example 2: Version B_2 includes the events T_0, T_1, T_4, and T_5. Version A_2 adds T_2.				
Version B_2	20.761	23.093	111.2326 %	
Version A_2	31.487	18.669	59.2911 %	p = 0.7545, q = 0.1326
Sum of Versions B_2 and A_2	52.248	41.762	79.9303 %	**1.25x overall speedup**
Example 3: Version B_3 includes the events T_0, T_1, T_2, T_3, and T_4. Version A_3 adds T_5.				
Version B_3	20.864	23.225	111.3161 %	
Version A_3	32.9	22.987	69.8693 %	p = 0.7623, q = 0.2783
Sum of Versions B_3 and A_3	53.764	46.212	85.9534 %	**1.16x overall speedup**
Example 4: Version B_4 includes the events T_0, T_1, T_2, and T_4. Version A_4 adds T_5.				
Version B_4	18.203	20.645	113.4154 %	
Version A_4	30.239	21.39	70.7365 %	p = 0.7576, q = 0.2866
Sum of Versions B_4 and A_4	48.442	42.035	86.7739 %	**1.15x overall speedup**

6.2 Verifying the simulation model of ARQ

States We define *GlobalState* as a tuple that has two components, namely the protocol state and the network cloud. The protocol state contains *SeqNoSent* (the sequence number of the data packet that was most recently sent by the sender and whose ACK the sender is waiting for), *SeqNoExpected* (the sequence number of the data packet that the receiver is expecting), *NumDistinctDataMsgSent* (the total number of distinct data packets sent by the sender), and *NumDistinctDataMsgReceived* (the total number of distinct data packets received by the receiver). The network cloud models the network as an unbounded set that contains the data and ACK packets. The initial state is the state in which the sender has just sent the first data packet (denoted as *D0*), the receiver is expecting *D0* and the network contains *D0*.

An important safety property for any reliable unicast protocol is that the receiver does not miss any data packet that the sender believes to have been received by the receiver. In an ARQ protocol that uses a 1-bit sequence number, this safety property translates to the requirement that the difference between the total number of distinct data packets transmitted by the sender (*NumDistinctDataMsgSent*) and the total number of distinct data packets received by the receiver (*NumDistinctDataMsgReceived*) is always less than or equal to 2. This safety property can be expressed as follows:

$$(NumDistinctDataMsgSent - NumDistinctDataMsgReceived) <= 2$$

The hash code of a state is computed by first constructing an integer array representation of the state. Following that, an integer-valued hash code of this integer array representation is computed using the Jenkins' hash function that we borrowed from JPF. (We slightly modified this function to return an integer instead of a long value.) The integer array representation of a state depends on the protocol-specific information such as the sequence numbers in the headers of the data and ACK packets, the values of *SeqNoSent*, *SeqNoExpected*, *NumDistinctDataMsgSent*, and *NumDistinctDataMsgReceived*.

Events Next, we specify the set of events, when each event is enabled, and how each event is handled. The events are listed as follows:

T_0 Delivering a data packet: This event is enabled if the network contains at least one data packet. The event is handled by removing this packet from the network and forwarding it to the receiver.

T_1 Delivering an ACK packet: This event is enabled if the network contains at least one ACK packet. The event is handled by removing this packet from the network and forwarding it to the sender.

T_2 Timeout of the retransmission timer at the sender: This event is enabled if the network does not contain any data packets. This condition corresponds to either the case that the most recently sent data packet was lost and hence needs to be retransmitted or the case that the ACK of the most recently sent data packet is still in transit from the receiver to the sender. The event is handled by having the sender retransmit the data packet that was most recently sent and whose ACK the sender is waiting for.

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

T_4 Loss of an ACK packet: This event is enabled if the network contains at least one ACK packet. The event is handled by removing this packet from the network.

Results of the verification We manually injected the following error (which we call Counterexample 1) in the ACK packet delivery event T_1. In Counterexample 1, the sender does not check the sequence number in the ACK before sending a data packet; i.e., the sender always sends a new data packet whenever an ACK is received. This error may lead to a violation of the safety property because it may make the sender think that a lost/corrupted data packet has been received by the receiver. We call this assertion violation as Counterexample 1. The interested reader is referred to [13] for a detailed trace (along with an explanation) of this counterexample.

To evaluate the incremental state space exploration technique, we evaluate its performance in a scenario in which code changes do incur behavioral changes. Specifically, we simulate the behavior of a user who tries to implement the ACK packet delivery event T_1 correctly. First, the user forgets to make the sender check the sequence number in the ACK before sending a data packet causing Counterexample 1 to occur (we call this implementation Version B). Following that, the user figures out the correct implementation, which requires checking the sequence number in the ACK to determine whether a new data packet or a retransmission should be sent (we call this implementation Version A). Table 9 shows the state space exploration time under this scenario using both the non-incremental and incremental techniques. For each experiment, we ran 100 replications. Each replication has a different seed. We distinguish between the two cases: Case I (WriteEachVersion) and Case II (WriteFirstVersion). As shown in Table 9, incremental state space exploration is *not* able to provide up a speedup in state space exploration time.

In order to understand why the incremental state space exploration procedure did not provide a speedup, we show in Table 10 a breakdown of the average state space exploration time spent in some selected operations taking the second-to-last row in Table 9 as an example. As shown in Table 10, the costs of executing transitions (i.e., the sum of copying from the verification model to the simulation model, executing events, and copying from the simulation model to the verification model), computing hash codes, and checking the safety property are *not* high taking together less than 52 % of the total average time in the non-incremental state space exploration. Furthermore, due to the relatively large value of $q = 0.2498$, extremely small value of $p = 0.0112$, and the relatively large value of $f = 0.2508$,

Table 9: ARQ case study: Time (sec.) for both the non-incremental and the incremental state space exploration techniques. Search strategy is BFS-SH. The state space explorer does NOT terminate state space exploration if an assertion violation is detected.

Case I (WriteEachVersion): In incremental state space exploration: $read$ and $write$ are set as follows:
$read = false$ and $write = true$ (first version in each set).
$read = true$ and $write = true$ (second version in each set). Same data structure is used for $INPUT$ and $OUTPUT$.

B → A $MAX\_DEPTH = 30$	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE Time (sec.) (see caption of Case I)	Inc. / Non. Percentage	Observations on Incremental SSE
Version B (Counterexample 1)	4.5265	5.2485	115.9505 %	
Version A (Correct T_1)	6.0604	7.5355	124.34 %	$p = 0.05$, $q = 0.2498$
Sum of Versions B and A	10.5869	12.784	120.753 %	**no speedup**

B → A $MAX\_DEPTH = 40$	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE Time (sec.) (see caption of Case I)	Inc. / Non. Percentage	Observations on Incremental SSE
Version B (Counterexample 1)	52.5009	66.5353	126.7317 %	
Version A (Correct T_1)	64.5754	88.3622	136.8357 %	$p = 0.0113$, $q = 0.2498$
Sum of Versions B and A	117.0763	154.8975	132.3047 %	**no speedup**

Case II (WriteFirstVersion): In incremental state space exploration: $read$ and $write$ are set as follows:
$read = false$ and $write = true$ (first version in each set).
$read = true$ and $write = false$ (second version in each set).

B → A $MAX\_DEPTH = 30$	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE Time (sec.) (see caption of Case II)	Inc. / Non. Percentage	Observations on Incremental SSE
Version B (Counterexample 1)	4.5265	5.2485	115.9505 %	
Version A (Correct T_1)	6.0604	6.4362	106.2009 %	$p = 0.05$, $q = 0.2498$
Sum of Versions B and A	10.5869	11.6847	110.3694 %	**no speedup**

B → A $MAX\_DEPTH = 40$	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE Time (sec.) (see caption of Case II)	Inc. / Non. Percentage	Observations on Incremental SSE
Version B (Counterexample 1)	52.5009	66.5353	126.7317 %	
Version A (Correct T_1)	64.5754	75.728	117.2707 %	$p = 0.0112$, $q = 0.2498$
Sum of Versions B and A	117.0763	142.2633	121.5133 %	**no speedup**

more than 99 % of the transitions are executed in the incremental state space exploration procedure as shown in the last row of Table 10. The last column of Table 10 shows the estimates of the values that appear in the third necessary condition, which is clearly violated in this case study. The J-Sim state space explorer outputs a message informing the user that the third necessary condition is violated; hence, using the incremental state space exploration procedure in this case study is discouraged.

References

[1] 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.

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

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

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

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

Table 10: ARQ case study: Breakdown of the average state space exploration time (sec.) spent in some selected operations. The example shown corresponds to the second-to-last row in Table 9.

Version A Correct T_1 $MAX\_DEPTH = 40$ Search Strategy is BFS-SH	Non-incremental SSE Time (sec.) ($read = false$, $write = false$)	Incremental SSE Time (sec.) ($read = true$, $write = false$)	Observations
Copy from V model to S model	4.5998	5.2318	$f = 0.2508$
Execute events	11.9115	13.4024	$q = 0.2498$
Copy from S model to V model	4.5162	4.9137	$p = 0.0112$
Compute a hash code	9.3363	10.8074	$X = 20.959\ \mu s$
Verify an assertion	2.7213	2.9469	$H = 9.3059\ \mu s$
Read from input file	0	3.5704	$Y = 2.7125\ \mu s$
Search in $INPUT\_SSEG$	0	1.0653	$L = 5.8764\ \mu s$
Insert in *AlreadyVisited*	4.2419	5.2016	$F = 3.5584\ \mu s$
Search in *AlreadyVisited*	3.6985	3.7237	Clearly, the third necessary condition
Enabling function	6.8969	7.2344	is violated.
Insert in *ToBeExplored*	0.2759	0.2796	
Compute the BeFS tuple	0	0	
Subtotal	48.1983	58.3772	
Other Operations	16.3771	17.3507	
Total Time	64.5754	75.728	
Number of executed transitions	1003272	997992	

[6] Steven Lauterburg, Ahmed Sobeih, Mahesh Viswanathan, and Darko Marinov. Incremental state-space exploration for programs with dynamically allocated data. Submitted for publication. September 2007.

[7] 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*.

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

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

[10] 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)*.

[11] 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.

[12] 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.

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

[14] 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.

[15] 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.

[16] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall International Inc., 1996.

```
 0. procedure IncrementalSSExplore(boolean write, boolean read)
 1. if ( read ) INPUT = readFromInputSSEGFile() ;
 2. initialState.depth = 0 ;
 3. initialState.computeHashCode() ;
 4. AlreadyVisited.add(initialState.HashCode()) ;
 5. ToBeExplored.add(initialState) ;
 6. while ( | ToBeExplored | > 0 ) {
 7.    currentState = ToBeExplored.remove() ;
 8.    EnabledEvents = GenerateEnabledEvents(currentState) ;  /* See Figure 1 for GenerateEnabledEvents() */
 9.    for ( int i = 0 ; i < EnabledEvents.size() ; i++ ) {
10.       EventInfo E = EnabledEvents.get(i) ;
11.       execute = true ;
12.       postProcess = true ;
13.       if ( read ) {
14.          if ( isNonModifiedEvent(E.getEventID()) ) {
                /* q: event is modified. 1-q: event is not modified. */
15.             if ( <currentState.HashCode(), E> exists in INPUT ) { /* L: search in INPUT */
                   /* p: transition information is found in INPUT. 1-p: not found in INPUT. */
16.                postProcess = false ;
17.                nextStateHashCode = <currentState.HashCode(), E>.getNextStateHashCode() ;
18.                if ( (currentState.Depth() + 1) == MAX_DEPTH ) {
                      /* d: deepest transition; i.e., depth of nextState == MAX_DEPTH. */
19.                   execute = false ;
20.                } else if ( nextStateHashCode exists in AlreadyVisited ) { /* A: search in AlreadyVisited */
                         /* b: backward transition; i.e., nextState has already been visited. f=1-b-d: forward transition. */
21.                      execute = false ;
                   }
                }
             }
          }
22.       if ( execute ) {
23.          nextState = GenerateNextState(currentState, E) ; /* X: execute a transition */
24.          nextState.setDepth(currentState.Depth() + 1) ;
25.          if ( postProcess ) {
26.             nextState.computeHashCode() ; /* H: compute a hash code */
27.             checkProperty = nextState.verifySafety() ; /* Y: verify the safety property */
28.          } else {
29.             nextState.setHashCode(nextStateHashCode) ; /* Hash code of nextState was obtained from INPUT. */
30.             checkProperty = true ; /* INPUT does not store hash codes of states that violate an assertion */
             }
31.          if ( (checkProperty == false) AND (DoesCounterexampleContainEvent(nextState)) ) {
32.             Print("Counterexample ") ;
33.             printCounterexample(nextState) ;
34.             if ( write ) writeToOutputSSEGFile(OUTPUT) ;
35.             exit ;
36.          } else if ( (checkProperty) AND (nextState.Depth() < MAX_DEPTH) ) {
                /* v: safe transition; i.e., nextState satisfies the safety property. */
                /* d: deepest transition; i.e., depth of nextState == MAX_DEPTH. */
37.             if ( (postProcess == false) OR (nextState.HashCode() does not exist in AlreadyVisited ) {
                   /* A: search in AlreadyVisited */
                   /* b: backward transition; i.e., nextState has already been visited. f=1-b-d: forward transition. */
38.                if ( search strategy is best-first ) nextState.computeBeFSTuple() ; /* U: compute the BeFS tuple */
39.                AlreadyVisited.add(nextState.HashCode()) ; /* R: add a hash code to AlreadyVisited */
40.                ToBeExplored.add(nextState) ; /* N: add a state to ToBeExplored */
41.                if ( write AND postProcess ) OUTPUT.add(<currentState.HashCode(), E, nextState.HashCode()>) ;
42.             } else if ( write AND postProcess ) OUTPUT.add(<currentState.HashCode(), E, nextState.HashCode()>) ;
43.          } else if ( checkProperty AND write AND postProcess ) OUTPUT.add(<currentState.HashCode(), E, nextState.HashCode()>) ;
          }
       }
    }
44. if ( write ) writeToOutputSSEGFile(OUTPUT) ;
```

Figure 2: An incremental version of the state space exploration procedure in Figure 1. Added or modified lines are shown in italic. The symbols used in the comments are explained in Table 1.