Efficient systematic testing of network protocols with temporal uncertain events

Minh Vu*, Lisong Xu*, Sebastian Elbaum†, Wei Sun*, Kevin Qiao‡

* Computer Science and Engineering, University of Nebraska-Lincoln, Lincoln, NE 68588-0115, {mvu, xu, wsun}@cse.unl.edu

† Dept of Computer Science, University of Virginia, Charlottesville, Virginia, selbaum@virginia.edu

‡ Dept of Computer Science, University of Maryland, College Park, MD, kqiao@umd.edu

***Abstract*—The correctness of network protocol implementations is difficult to test mainly because of the temporal uncertain nature of network events. In order to test the correctness of a network protocol implementation using network simulators, we need to systematically simulate the behavior of the network protocol under all possible cases of temporal uncertain events, which is very time consuming. The recently proposed Symbolic Execution based Interval Branching (SEIB) simulates a group of uncertain cases together in a single simulation branch, and thus is more efficient than brute force testing. In this paper, we argue that the efficiency of SEIB could be further exponentially improved by eliminating unnecessary comparisons of the event timestamps. Specifically, we summarize and present three general types of unnecessary comparisons when SEIB is applied to a general network simulator, and then correspondingly propose three novel techniques to eliminate them. Our extensive simulations show that our techniques can improve the efficiency of SEIB by several orders of magnitude, such as from days to minutes.**

***Index Terms*—Network protocol testing; symbolic execution; temporal uncertainty; discrete event simulator.**

I. Introduction

The correctness of network protocols is difficult to test mainly because of the temporal uncertain nature of network events. An event is called a temporal uncertain event (or just uncertain event for short), if it may occur at anytime in an interval instead of a single time instant. For example, the arrival event of a packet at a node is an uncertain event, if the packet may experience an uncertain delay in a network. Many network protocol bugs are related to uncertain events and can be detected only in corner cases with low probabilities. For example, most of the recently found TCP bugs [6] are related to low-probability uncertain events. As another example, many bugs of wireless sensor networks [15] are related to low-probability uncertain events, and they are hard to detect and costly to fix once deployed in the fields.

In this paper, we consider the type of correctness testing methods using network simulators, such as NS-3 [12], because network simulators are popular tools for testing network protocol implementations before their deployment, and can be used to detect not only the design bugs but also the implementation bugs. In order to check the correctness of a network protocol with uncertain events, we need to simulate and check the behavior of the network protocol under all possible cases of uncertain events, which is, however, **very challenging**. Let's consider a network protocol with n packets, each experiencing k possible packet delays. For this example, we need to simulate and check the behavior of the network protocol for all k^n possible uncertainty cases (i.e., packet delay combinations), which is very time consuming as k is usually very large. For instance, for a packet delay in (0, 1] second, k is 10^3 with a millisecond resolution, and 10^6 with a microsecond resolution.

State of the art: There are two classes of testing methods using network simulators. 1) *Random testing* simulates randomly selected uncertainty cases, such as random packet delays according to a distribution. 2) *Systematic testing* aims to enumerate and simulate all uncertainty cases. Random testing is cost-effective at evaluating the performance of a network protocol in cases that fit the utilized distribution, but it cannot guarantee the correctness of the network protocol. In contrast, systematic testing can be more cost-effective when checking the correctness of a network protocol under all possible cases.

Systematic testing methods can be further classified into two categories. 1) *Brute force testing* separately simulates and checks a network protocol for each uncertainty case. It is inefficient, but it is simple and can be used for any general network simulator. 2) *Interval branching* [13] simulates multiple uncertainty cases together by associating the timestamp of a simulation event with an interval. The interval of an event indicates the set of all possible occurrence times of the event, and two events overlap if the intersection of their intervals is not empty. When overlapping, internal branching forks to cover all possible occurrence orders of the events, and each branch continues with updated timestamp intervals for the involved events. Therefore, interval branching is more efficient than brute force testing.

Interval branching can be implemented in two different ways. 1) *Direct interval branching*: Interval branching was originally implemented by directly modifying a simulator [13], however, it requires substantial changes to the simulator, especially nontrivial work for forking. This cumbersome implementation has considerably slowed the adoption of interval branching by the networking community. 2) *Symbolic Execution based Interval Branching (SEIB)*: Interval branching has been recently implemented [8], [17], [18] by leveraging symbolic execution [4], a popular program analysis technique in software testing and verification community. Conceptually, a tester declares the timestamp variable of an event in a simulator as a symbolic variable that can take multiple values, and then executes the simulator using a symbolic execution engine that automatically takes care of forking the simulator when comparing overlapping symbolic variables. As SEIB greatly simplifies the implementation, it is more likely to be widely adopted than direct interval branching.

The efficiency of SEIB mainly depends on the total number of generated SEIB branches, which in turn mainly depends on the total number of comparisons of overlapping timestamps in the simulation. Specifically, the number of branches is approximately an exponential function of the number of comparisons of overlapping timestamps, and thus it is still time-consuming for SEIB to test the correctness of a network protocol.

Our work: In this paper, we argue that the efficiency of SEIB could be significantly improved by eliminating unnecessary comparisons of overlapping timestamps. By doing so, we can potentially exponentially reduce the number of generated branches and then the testing time. This is because an unnecessary comparison, if not eliminated, generates a new branch that may continuously fork and generate new branches. Specifically, we summarize and present three general types of unnecessary comparisons when SEIB is applied to a general network simulator (e.g., NS-3), and then correspondingly propose three techniques to modify the simulator in order to eliminate these unnecessary comparisons.

First, unnecessary comparisons due to *simultaneous events*. We find that a simulator may compare the timestamps of two events for multiple times instead of once, in order to check the special case where two events happen at exactly the same time instant. These unnecessary comparisons can be eliminated by reorganizing the comparison code of the simulator.

Second, unnecessary comparisons due to *conditional ineffective events*. We find that a simulator may have various types of conditional ineffective events, which have no impact on the simulation result under some conditions but their timestamps are unnecessarily compared with other events. For example, an uncertain event which might happen after the end of a simulation, and a TCP retransmission timeout event that might be canceled by an uncertain ACK. These unnecessary comparisons can be eliminated by identifying when these conditional ineffective events become ineffective and then removing them from the simulation.

Third, unnecessary comparisons due to *independent events*. Two events on different network nodes are independent, if they do not have any impact on each other. As a result, two independent events can be executed in any order in a simulation, and it is not necessary to compare their timestamps. The general idea of exploring independent events to speed up software and network testing (e.g., model checking) is not new. The novelty of our work is that we apply it to SEIB and propose to eliminate unnecessary comparisons of independent events on different nodes by decomposing the network simulation into multiple synchronized node simulations. Our decomposition technique is similar to and inspired by the traditional parallel simulation methods. But different from a parallel simulator that runs on multiple processors with the aim to speed up the parallel simulation, our work still runs on a single processor with the aim to reduce the number of branches.

Pseudocode 1 A discrete-event network simulator

```
 1: variables for network state
 2: array: $list[\,]$                                  ▷ global event list
 3: variable: $clock$                                  ▷ global clock
 4: function $Main$
 5:     repeat
 6:         $e \leftarrow FindAnEvent()$
 7:         $ExecuteAnEvent(e)$
 8:     until $list$ is empty or $e$ is a simulation end event
 9: function $FindAnEvent$
10:     $e \leftarrow list[0]$                         ▷ The earliest one
11:     Remove $list[0]$ from $list$
12:     return $e$
13: function $ExecuteAnEvent(e)$
14:     $clock \leftarrow e.t$                         ▷ Advance the clock
15:     Update related state variables
16:     Insert newly generated events to $list$
```

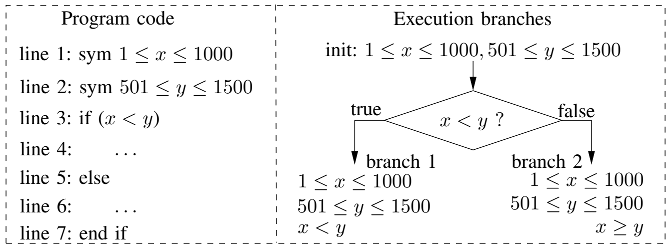
Our contributions are threefold. First, we propose three novel techniques to potentially exponentially improve the efficiency of SEIB for testing the correctness of a network protocol under all possible cases of temporal uncertain events. For each proposed technique, its correctness and efficiency can be formally proved. Second, we implement our proposed techniques by modifying the popular general network simulator, NS-3. To the best of our knowledge, this is the first time that SEIB is applied to a large general network simulator that has been widely used in the networking community. Third, we evaluate the efficiency of our proposed techniques by comparing the modified NS-3 with the original NS-3 using various network topologies and protocols including TCP, UDP, and IP routing. The results show that when executed by SEIB, the modified NS-3 achieves several orders of magnitude shorter testing times than the original NS-3, for example, from days to minutes. Our evaluation also shows that our techniques are several orders of magnitude more efficient than traditional parallel simulation methods when executed by SEIB.

II. Background

A. Network simulation

Network simulation is usually conducted using a discrete-event network simulator, which simulates a network using a sequence of discrete events in time, and updates the simulation variables only when an event occurs.

Pseudocode 1 shows the major data structures and functions of a discrete-event network simulator. It maintains three types of data structures. 1) The network state variables, which describe the current state of the whole network. 2) The global $list$ of pending events in the whole network, which are sorted in the ascending order of their timestamps. 3) The global $clock$, which is the current time in a simulation. The simulator $Main$ function repeatedly finds and executes an event e in $list$ until the last event or the end of the simulation. Function $FindAnEvent$ finds the first event in $list$ that is the one with the earliest timestamp in order to avoid *causal violations*, which happen when a future event affects a past



```
line 1: sym 1 ≤ x ≤ 1000
line 2: sym 501 ≤ y ≤ 1500
line 3: if (x < y)
line 4:        ...
line 5: else
line 6:        ...
line 7: end if
```

Fig. 1. A symbolic execution example with symbolic variables x and y.

event. Function $ExecuteAnEvent$ advances the global $clock$ to the timestamp $e.t$ of event e, updates related network state variables, and generates zero or more new events that will be inserted into $list$.

In this paper, we consider NS-3 [12], which is a discrete-event network simulator widely used in the networking community. It can simulate many networking protocols, and can also run the original Linux networking stack.

B. Symbolic execution

Instead of running a program directly, symbolic execution runs a program with symbolic variables using a symbolic execution engine. Different from normal program variables that take concrete values, a *symbolic variable* takes a symbolic value represented as symbolic constraints. That is, a symbolic variable can take all possible values satisfying the symbolic constraints. Fig. 1 shows an example. The first two lines of the program declare two symbolic variables x and y with their initial constraints. For example, x can take any integer values between 1 and 1000. Once the execution reaches an $if(cond)$ statement involving symbolic variables, the symbolic execution engine queries a constraint solver to check the feasibility of both possibilities (i.e., $cond$ = true or false) under the current constraints. For example, for $cond$ = "$x < y$" in line 3, because both possibilities are feasible, the current execution forks into two branches. The true branch continues with additional constraint $x < y$, and the false branch continues with additional constraint $x \ge y$.

Symbolic execution is a powerful technique widely used in the software testing and verification community, because it can automatically divide all possible combinations of the symbolic variable values into equivalence classes. The combinations in the same equivalence class have the same execution path, and are executed together using the same branch. For Fig. 1, there are a total of $1000 \times 1000 = 10^6$ combinations of x and y. Without symbolic execution, we need to execute the program for 10^6 times, one for each combination, in order to check all possible behaviors of the program. With symbolic execution, we execute the program using only two branches. For example, all combinations satisfying constraints $1 \le x \le 1000$, $501 \le y \le 1500$, and $x < y$ have the same execution path (i.e., lines 1, 2, 3, 4, 7), and are executed together as branch 1.

In this paper, we use S^{2}E [7], which is a powerful symbolic execution platform that can symbolically execute NS-3 in a virtual machine. The virtual machine is emulated using the QEMU machine emulator, and the symbolic execution is conducted using the KLEE symbolic execution engine [3].

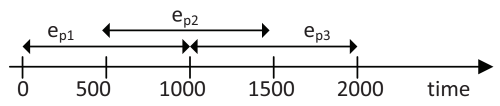


Fig. 2. Example: Three uncertain packet arrival events at the destination node. The double-headed arrows indicate their timestamp intervals.

III. Symbolic Execution based Interval Branching

This section introduces basic definitions, explains how SEIB works, and discusses the advantage and limitation of SEIB

A. Definitions and notation

In this paper, we consider only temporal uncertain events caused by uncertain network delays, which are the major uncertainty source for network protocols. Below, let's consider an example, where two nodes are connected with a link, and a node sends three packets p_i, $i \in [1, 3]$, to the other one.

For each packet p_i, let d_{p_i} denote its *delay* over the link, and *delay interval* D_{p_i} denote the discrete set of all possible values of d_{p_i}. We say that delay d_{p_i} is *uncertain*, if D_{p_i} contains more than one value (i.e., $|D_{p_i}| > 1$). The *delay space* $\mathbb{D}$ of a simulation is the cross product of all delay intervals in the simulation, and a vector $\vec{d} \in \mathbb{D}$ is called a *delay vector*. For the example, $\mathbb{D} = D_{p_1} \times D_{p_2} \times D_{p_3}$ is a three-dimensional space, and $\vec{d} = (d_{p_1}, d_{p_2}, d_{p_3})$. Suppose that each packet p_i has the same $D_{p_i} = [1, 1000]$ ms assuming a millisecond resolution, then $|D_{p_i}| = 1000$ and $|\mathbb{D}| = 10^9$. That is, $\mathbb{D}$ has a total of 10^9 possible delay vectors.

For each event e in a simulation, let $e.t$ denote its timestamp, and *timestamp interval* $[e.t]$ denote the discrete set of all possible values of $e.t$. We say that event e or timestamp $e.t$ is uncertain, if $[e.t]$ contains more than one value. To simplify our discussion in this section, let's consider only the arrival events of these packets in the example. For each packet p_i, let e_{p_i} denote its *arrival event* at the destination node, and then $e_{p_i}.t$ is the packet arrival time. Suppose that the three packets in the example depart from their source node at 0, 500, and 1000 ms, respectively, and have the same $D_{p_i} = [1, 1000]$ ms. We have $e_{p_1}.t = 0 + d_{p_1}$, $e_{p_2}.t = 500 + d_{p_2}$, and $e_{p_3}.t = 1000 + d_{p_3}$. Therefore, $[e_{p_1}.t] = [1, 1000]$, $[e_{p_2}.t] = [501, 1500]$, and $[e_{p_3}.t] = [1001, 2000]$, as shown in Fig. 2.

We say that two timestamps *overlap*, if their timestamp intervals overlaps (i.e., nonempty intersection). For example, $e_{p_1}.t$ and $e_{p_2}.t$ overlap, because $[e_{p_1}.t] \cap [e_{p_2}.t] = [1, 1000] \cap [501, 1500] = [501, 1000]$. Intuitively, this means e_{p_1} and e_{p_2} may occur in different orders. As another example, $e_{p_1}.t$ and $e_{p_3}.t$ do not overlap, and this means that e_{p_1} and e_{p_3} may occur in only one order.

Note that, the uncertain delay of a packet has an impact not only on the packet itself but also on all the following events triggered by the packet. For example, the uncertain delay of a TCP data packet also affects the transmission event and arrival event of the ACK packet triggered by the data packet, and affects the simulation clock, the calculated round-trip time, the calculated timeout period, and then the following retransmission timeout events.

B. SEIB

We use Pseudocode 2 to explain how SEIB works, which shows part of a possible simulation code for the three-packet example. The $Main$ function first (lines 10 to 12) declares each delay d_{p_i} as a symbolic variable with the initial constraints defined according to its delay interval D_{p_i}. As a result, all other variables depending on these symbolic variables are automatically handled as symbolic variables by the symbolic execution engine of SEIB. For example, timestamp $e_{p_1}.t$ in line 14 is also a symbolic variable, and its timestamp interval $[e_{p_1}.t]$ is implicitly defined by the constraints of d_{p_1}. Lines 14 to 16 call function $InsertEvent$ to insert the three events to $list$. Functions $FindAnEvent$ and $ExecuteAnEvent$ of Pseudocode 1 are not shown here. Finally, line 18 checks the correctness of the simulation.

Fig. 3 shows the execution of lines 15 and 16, when Pseudocode 2 is executed by SEIB. Before executing line 15, $list = (e_{p_1})$. When executing line 15, function $InsertEvent$ compares whether e_{p_2} happens before e_{p_1} at the if statement in line 4. SEIB finds out that both possibilities are feasible according to the current constraints. As a result, SEIB forks the current execution into two branches: the true branch continues to line 5 and the false branch to line 7. Each branch then continues with different $list$'s (shown in Fig. 3) and different updated constraints (not shown in Fig. 3).

We can see that the total number of branches depends on the number of comparisons of overlapping timestamps, which are indicated by shaded diamonds in Fig. 3. Finally, a total of three branches are generated due to two comparisons of overlapping timestamps. This is because a comparison of non-overlapping timestamps does not generate any new branches. For example, $[e_{p_1}.t] = [1, 1000]$ does not overlap with $[e_{p_3}.t] = [1001, 2000]$, and thus $e_{p_3}.t < e_{p_1}.t$ is always false. Note that when $InsertEvent(e_{p_3})$ is called in branch 1, $[e_{p_3}.t]$ and $[e_{p_2}.t]$ do not overlap anymore and specifically $e_{p_3}.t < e_{p_2}.t$ is always false. This is because the constraints of branch 1 have been updated with additional constraint $e_{p_2}.t < e_{p_1}.t$ after calling $InsertEvent(e_{p_2})$.

C. Advantage and limitation of SEIB

SEIB is more efficient than brute force testing when checking the correctness in all possible cases of uncertain events. For the example, brute force testing needs to run the simulation for a total of $|\mathbb{D}| = 10^9$ times, one for each delay vector by changing lines 10 to 12 of Pseudocode 2 to specific delays. In contrast, SEIB only needs to execute the simulation once with three generated branches, and the assertion at line 18 is checked for each branch. However, the number of SEIB branches still increases quickly and is approximately an exponential function of the number of comparisons of overlapping timestamps, leading to poor efficiency.

IV. Our method

A. Overview

Current SEIB works [8], [17], [18] demonstrate promising potential of SEIB, but they use only small and simple network

Pseudocode 2 Part of a simulation code for the three-packet example in Section III

```
1:  array: $list[\,]$
2:  function $InsertEvent(new\_e)$
3:      for $k \leftarrow 0$; $k < list.size$; $k \leftarrow k+1$ do
4:          if $new\_e.t < list[k].t$ then
5:              Insert $new\_e$ to position $k$ in $list$
6:              return
7:          end if
8:      Append $new\_e$ to the end of $list$
9:  function $Main$
10:     sym $1 \le d_{p_1} \le 1000$                ▷ symbolic variable
11:     sym $1 \le d_{p_2} \le 1000$                ▷ symbolic variable
12:     sym $1 \le d_{p_3} \le 1000$                ▷ symbolic variable
13:     ...
14:     $e_{p_1}.t \leftarrow 0 + d_{p_1}$; $InsertEvent(e_{p_1})$
15:     $e_{p_2}.t \leftarrow 500 + d_{p_2}$; $InsertEvent(e_{p_2})$
16:     $e_{p_3}.t \leftarrow 1000 + d_{p_3}$; $InsertEvent(e_{p_3})$
17:     ...
18:     assert(checking correctness)
```

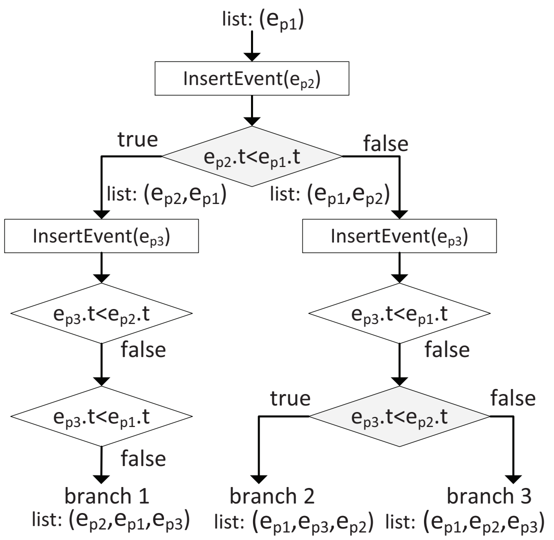


Fig. 3. Three branches generated when Pseudocode 2 is executed by SEIB, due to two comparisons of overlapping timestamps (shaded).

simulators. For example, SPD [17] writes a toy simulator to simulate only two nodes connected by a link. In this paper, for the first time, we apply SEIB to a large, general, and widely-used network simulator, NS-3. We find that the efficiency of SEIB when applied to NS-3 can be significantly improved by eliminating unnecessary comparisons of overlapping timestamps. We observe that these unnecessary comparisons are due to three general types of events: simultaneous, conditional ineffective, and independent events, which will be explained below. Then we propose three novel techniques to modify NS-3 in order to eliminate these unnecessary comparisons.

Our proposed techniques have the potential to exponentially reduce the number of branches, because an unnecessary comparison, if not eliminated, forks the current branch into two branches, each of which continuously forks for all the remaining comparisons of overlapping timestamps. For each proposed technique, we prove its correctness and efficiency. First, a technique is *correct*, if the modified simulator always

Pseudocode 3 Original NS-3 code for comparing two events

```
function $Before(e_1, e_2)$
    if $e_1.t < e_2.t$ then
        return True
    else if $e_1.t = e_2.t$ then
        if $e_1.id < e_2.id$ then
            return True
    return False
```

Pseudocode 4 Modified NS-3 code for comparing two events

```
function $Before(e1, e2)$
    if $e_1.id < e_2.id$ then
        if $e_1.t \le e_2.t$ then
            return True
    else
        if $e_1.t < e_2.t$ then
            return True
    return False
```

generates the same simulation result as the original one. Second, a technique is *efficient*, if the SEIB branches of the modified simulator is no more than that of the original one.

Note that we do not propose a new search algorithm to insert a new event new_e into a sorted event list $list$, because different search algorithms (e.g., sequential search in Pseudocode 2, or binary search) generate the same number of branches. Specifically, different search algorithms might have different total numbers of comparisons of timestamps, but have the same number of comparisons of overlapping timestamps.

B. Unnecessary comparisons due to simultaneous events

1) Simultaneous events: We say that two events are *simultaneous*, if they occur at the same time instant. Simultaneous events are handled differently by different network simulators. When comparing two simultaneous events, NS-3 puts the one with a smaller event ID before the other one in the event list. NS-3 uses a function the same as the $InsertEvent$ function in Pseudocode 2 to insert a new event to its event list, except that the if statement at line 4 uses the $Before$ function defined in Pseudocode 3 to compare two events.

2) Our technique: Pseudocode 3 compares timestamps $e_1.t$ and $e_2.t$ twice at lines 2 and 4. We propose Pseudocode 4 to compare the timestamps only once at either line 3 or line 6.

3) Correctness: We prove the correctness below.

***Theorem** 1:* Pseudocode 4 and 3 always generate the same simulation result.

Proof: There are four possible cases. Case 1: when $e_1.t < e_2.t$, both return true. Case 2: When $e_1.t > e_2.t$, both false. Case 3: When $e_1.t = e_2.t$ and $e_1.id < e_2.id$, both true. Case 4: When $e_1.t = e_2.t$ and $e_1.id > e_2.id$, both false. ■

4) Efficiency: For non-overlapping $e_1.t$ and $e_2.t$, both pseudocode generate only one branch. The following theorem considers overlapping $e_1.t$ and $e_2.t$.

***Theorem** 2:* Pseudocode 4 always generates no more branches than Pseudocode 3 for overlapping timestamps.

Proof: In the general case of overlapping $e_1.t$ and $e_2.t$, Pseudocode 3 forks twice and generates three branches. For example, if $[e_1.t] = [1, 1000]$ and $[e_2.t] = [501, 1500]$, Pseudocode 3 generates three branches corresponding to three cases: e_1 occurs before, at the same time, or after e_2. In this case, Pseudocode 4 generates only two branches.

A special case for overlapping $e_1.t$ and $e_2.t$ is when one timestamp interval contains only one time instant and is the left end or right end of another timestamp interval. For example, if $[e_1.t] = [1, 1000]$ and $[e_2.t] = [1000]$, Pseudocode 3 generates two branches. In this special case, Pseudocode 4 generates one or two branches depending on their event ids. ■

As an example of the general case, if every branch calls $Before$ on two overlapping timestamps for c times, Pseudocode 3 generates 3^c branches whereas our Pseudocode 4 generates 2^c branches.

C. Unnecessary comparisons due to cond. ineffective events

1) Conditional ineffective events: A simulator may have various types of conditional ineffective events, which have no impact on the simulation results under some conditions. We have identified two major types of conditional ineffective events. First, an uncertain event which might happen after the end of a simulation. NS-3 function $Simulator :: Stop(t)$ creates a special simulation end event with timestamp t so that the simulation stops at time t (see line 8 in Pseudocode 1). If the timestamp interval of an event is sufficiently long and contains t, it might happen after t and thus has no impact on simulation result. Second, a TCP retransmission timeout event that might be canceled by an uncertain ACK. If canceled, NS-3 only sets a flag of the event to indicate that it is canceled, but does not remove it from $list$.

2) Our technique: These unnecessary comparisons can be eliminated by identifying when these conditional ineffective events become ineffective, and then removing them from the simulation. Pseudocode 5 shows the pseudocode to identify (line 6) and discard the first type of conditional ineffective events when inserting a new event to $list$. The second type is handled in a similar manner.

3) Correctness: The correctness of Pseudocode 5 can be proved by the fact that a conditional ineffective event is removed only when it becomes ineffective.

4) Efficiency: Pseudocode 5 reduces the number of branches, because a conditional ineffective event once removed will not be compared with any other events. The reduction could be significant, when there are a large number of uncertain events overlapping with a simulation end event (often for long uncertain delay ranges), or when there are a large number of canceled timeout events (often for TCP).

D. Unnecessary comparisons due to independent events

1) Independent events: We first define the node associated with an event e. There are two general types of events: link events and node events. First, a link event e simulates the propagation of a packet over a link from a source node $e.src$ to one (or more) destination node $e.dst$. Event e is usually called a packet arrival event at node $e.dst$, and we say that it is associated with node $e.node = e.dst$. Second, a node event e simulates an event at a node i, and we say it is associated with node $e.node = i$. For example, a timeout event at a node, and an application event at a node.

Pseudocode 5 Handling conditional ineffective events

```
1: function $InsertEvent(new\_e)$
2:     for $k \leftarrow 0$; $k < list.size$; $k \leftarrow k + 1$ do
3:         if $Before(new\_e, list[k])$ then
4:             Insert $new\_e$ to position $k$ in $list$
5:             return
6:         else if $list[k]$ is a simulation end event then
7:             return
8:         end if
9:     Append $new\_e$ to the end of $list$
```

We use a general event dependency model [10] for general networking protocols. We say that two event e_i and e_j are *independent* of each other, if neither $e_i \to e_j$ nor $e_j \to e_i$ holds, where $\to$ is a relation defined by the following three cases. 1) $e_i \to e_j$, if $e_i.node = e_j.node$ and $Before(e_i, e_j)$. 2) $e_i \to e_j$ for a link event e_j, if e_i generates e_j (then $e_i.node = e_j.src$). 3) $e_i \to e_j$, if there exists an event e_k such that $e_i \to e_k$ and $e_k \to e_j$. Intuitively, $e_i \to e_j$ means that e_i has an impact on e_j. If e_i and e_j are independent, they do not have any impact on each other. Therefore, two independent events can be executed in any order in a simulation, and it is not necessary to compare their timestamps.

2) Overview: NS-3 sorts all events using relation $Before$, which is a strict total order (i.e., irreflexive, antisymmetric, transitive, and connex). When NS-3 is executed by SEIB, the number of branches is in the order of the number of different total orders of the events with respect to relation $Before$.

We propose to modify NS-3 to sort all events using relation $\to$, which is a strict partial order (i.e., irreflexive, antisymmetric, and transitive). As result, when the modified NS-3 is executed by SEIB, the number of branches is in the order of the number of different partial orders of the events with respect to relation $\to$.

The general idea of exploring partial ordering of event dependency to speed up software and network testing (e.g., in model checking) is not new. The novelty of our work is that we apply it to SEIB and we propose to achieve partial ordering for SEIB by decomposing the network simulation into multiple synchronized node simulations.

3) Differences from traditional parallel simulation: Our decomposition technique is similar to and inspired by the traditional parallel simulation methods [9]. Both our decomposition technique and parallel simulation decompose the simulation of a network into multiple simulations of the nodes. But different from a parallel simulator which runs on multiple parallel processors with the aim to speed up the simulation, our work still runs on a single processor with the aim to reduce the number of branches. Thus, they have different design choices.

First about the synchronization among multiple node simulations. Parallel simulation considers how to reduce the communication overhead of the synchronization among different processors. Our decomposition technique considers how to eliminate unnecessary comparisons of independent events in the synchronization, but not about communication overhead (none as it runs on a single processor).

Second about the lookahead that is the minimum latency for an event on a node to have an impact on another node and is usually the propagation delay from the first node to the second one. Lookahead is widely used in many parallel simulation methods to improve the parallelism of different node simulations. For example, event e_i on a node can be executed before e_j on a different node, if $e_i.t < e_j.t + lookahead$ (i.e., e_j has no impact on e_i). However, two non-overlapping timestamps $e_i.t$ and $e_j.t$ might become overlap, due to the lookahead. Thus lookahead is not always helpful, and is not used in our decomposition technique.

4) Our technique: Pseudocode 6 shows our decomposed simulator corresponding to the original NS-3 simulator illustrated in Pseudocode 1. By comparing the first three lines of these two simulators, we can see that we still keep the original network state variables, but we change the one-dimensional array $list$ to an two-dimensional array $local\_list$ and change the variable $clock$ to an one-dimensional array $local\_clock$ so that each node i has its own event list $local\_list[i]$ and its own clock $local\_clock[i]$. Below we use $local\_list$ to refer to the set of all the events in a network, and $local\_list[i]$ to refer to the sorted list of all the events at node i. The two simulators have the same $Main$ function, but different $FindAnEvent$ and $ExecuteAnEvent$ functions, which are explained below.

Function $FindAnEvent$ needs to find an event e that is safe to execute, in order to avoid causal violations. An event e in $local\_list$ is *safe*, if there does not exist any event e' in $local\_list$ such that $e' \to e$. Because relation $\to$ is a strict partial order, there may exist multiple safe events. But for a node i, its local earliest event $local\_list[i][0]$ may not be safe. There are two general ways to determine whether $local\_list[i][0]$ is safe: global synchronization using the global time information of all the nodes, and local synchronization using only the local time information of the neighbors of node i. In order to reduce the unnecessarily timestamp comparisons among different nodes, we choose local synchronization.

Function $LocalSynchronization$ implements our local synchronization method, which is motivated by the local causal constraint [9] in the traditional parallel simulation. The basic idea is that the local earliest event $local\_list[i][0]$ at node i is safe, if $local\_list[i]$ contains at least one packet arrival event from each neighbor and the nondecreasing arrival condition is met. The *nondecreasing arrival condition* requires that the packet arrival events from a source node j to a destination node i must be added into $local\_list[i]$ in the nondecreasing order of their timestamps. Note that, because of the uncertain delay, the timestamp order of packet arrival events to node i may not be the same as the order that they are generated at source node j. To achieve the nondecreasing arrival condition, a newly generated packet arrival event is first inserted into $local\_list[j]$ of source node j (line 40). When this event becomes the local earliest event in $local\_list[j]$, it is moved to $local\_list[i]$ of destination node i (line 21).

However, deadlock may occur in $LocalSynchronization$, which happens when each node is waiting for a packet arrival event from one or more of its neighbors. In this case, $LocalSynchronization$ could not find any safe event and returns $null$. The deadlock can be recovered in two general

Pseudocode 6 Decomposition to multiple node simulations

```
1:  variables for network state
2:  array: $local\_list[node][\,]$                    ▷ local event lists
3:  array: $local\_clock[node]$                       ▷ local clocks
4:  function $Main$
5:      repeat
6:          $e \leftarrow FindAnEvent()$
7:          $ExecuteAnEvent(e)$
8:      until $list$ is empty or $e$ is a simulation end event
9:  function $FindAnEvent$
10:     $e \leftarrow LocalSynchronization()$
11:     if $e = null$ then
12:         $e \leftarrow GlobalDeadlockRecovery()$
13:     return $e$
14: function $LocalSynchronization$
15:     repeat
16:         for each node $i$ do
17:             while $local\_list[i]$ contains at least one arrival event from each neighbor do
18:                 $e \leftarrow local\_list[i][0]$
19:                 Remove $local\_list[i][0]$ from $local\_list[i]$
20:                 if ($e$ is arrival event) and ($i \neq e.dst$) then
21:                     Insert $e$ to $local\_list[e.dst]$
22:                 else
23:                     return $e$
24:     until no more moving of arrival events
25:     return $null$
26: function $GlobalDeadlockRecovery(\,)$
27:     for each node $i$ do
28:         for each node $j \neq i$ do
29:             $safe \leftarrow True$
30:             if not $Before(local\_list[i][0], local\_list[j][0])$ then
31:                 $safe \leftarrow False$
32:                 break;
33:         if $safe$ then
34:             $e \leftarrow local\_list[i][0]$
35:             Remove $local\_list[i][0]$ from $local\_list[i]$
36:             return $e$
37: function $ExecuteAnEvent(e)$
38:     $local\_clock[e.node] \leftarrow t(e)$
39:     Update related network state variables
40:     Insert newly generated events to $local\_list[e.node]$
```

ways: global recovery using global time information of all the nodes, and local recovery using the local time information of the neighboring nodes. However, a limitation of local recovery (such as the null message method [9]) is the time-creeping problem, where the local clock of each node advances iteratively but slowly when comparing with the timestamps of its events, and leads to multiple unnecessary timestamp comparisons. Thus, we choose global recovery.

Function $GlobalDeadlockRecovery$ implements our global recovery method. If the local earliest event $local\_list[i][0]$ of node i happens before the local earliest event at every other node, it is a safe event. As explained before, we do not use the lookahead information when determining whether $local\_list[i][0]$ is safe or not.

Finally, function $ExecuteAnEvent$ updates the local clock of node $e.node$, updates related state variables, and inserts any newly generated events to its local event list.

5) Correctness: We prove the correctness of the proposed decomposition technique by proving that the events returned by $LocalSynchronization$ and $DeadlockRecovery$ are safe. That is, they do not violate the causal constraints and thus do not change the simulation results.

Theorem 3: The event e returned by function $LocalSynchronization$ is a safe event.

Proof: We prove that there does not exist any event e' in $local\_list$ such that $e' \rightarrow e$. Let i denote the node of event e. That is, $e = local\_list[i][0]$.

First, consider all the events at node i. Because $e = local\_list[i][0]$, e has the earliest timestamp among all the events in $local\_list[i]$. Thus, there does not exist any event e' in $local\_list[i]$ such that $e' \rightarrow e$.

Second, consider all the events on other node $j \neq i$. Because $local\_list[i]$ contains at least one arrival event from each neighbor and the nondecreasing arrival condition is met, there does not exist any event e' in $local\_list[j]$ such that $e' \rightarrow e$. ■

Theorem 4: The event e returned by function $DeadlockRecovery$ is a safe event.

Proof: Let i denote the node of event e. That is, $e = local\_list[i][0]$. Because e happens before the local earliest event at every other node j (line 30), e is the globally earliest event in $local\_list$ and is safe. ■

6) Efficiency: We consider two extreme cases of the proposed emulated parallel simulation. First, in the best case when $LocalSynchronization$ never returns null. That is, deadlock never occurs. Second, in the worst case when $LocalSynchronization$ always returns null. That is, deadlock always occurs. We prove that in both cases, Pseudocode 6 generates no more branches than Pseudocode 1.

Theorem 5: In the best case, Pseudocode 6 is more efficient than Pseudocode 1.

Proof: In the best case, Pseudocode 6 only compares an event e with other events at the same node (line 40), or if e is a packet arrival event, compares it with other events at the destination node (line 21). Thus Pseudocode 6 does not have any unnecessary comparisons of events on different nodes as in Pseudocode 1. ■

Theorem 6: In the worst case, Pseudocode 6 has the same efficiency as Pseudocode 1.

Proof: In the worst case, Pseudocode 6 compares the events on different nodes (line 30) or same node (line 40) using relation $Before$. As a result, Pseudocode 6 and 1 might have different total numbers of comparisons, but they have the same number of comparisons of overlapping timestamps. ■

Overall, the number of branches generated by Pseudocode 6 is in the order of the number of different partial orders of the events with respect to relation $\rightarrow$ in the best case, and is in the order of the number of different total orders of the events with respect to relation $Before$ in the worst case which is the same as Pseudocode 1.

V. Evaluation

We evaluate the efficiency of our proposed techniques using NS3 with various protocols and network topologies.

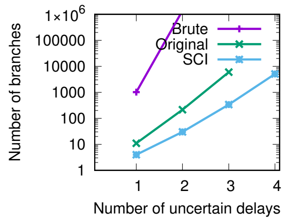


Fig. 4. Number of branches in the UDP experiments.

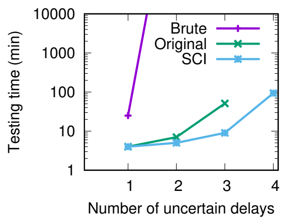


Fig. 5. Testing time of the UDP experiments

TABLE I
NUMBER OF BRANCHES OF THE UDP EXPERIMENTS

	$n = 1$	$n = 2$	$n = 3$	$n = 4$
Original	11	215	6013	⥁
S	8	109	2098	⥁
C	8	130	3184	⥁
I	5	41	677	13591
SC	5	53	858	18677
SCI	4	30	337	5065
SCP$_g$	15	310	9179	⥁
SCP$_n$	260	⥁	⥁	⥁

A. Simulation setup

We evaluate the following systematic testing methods. 1) We directly run the original NS-3 for each delay vector in the delay space (referred to as *Brute*). 2) We use S^{2}E to execute the original NS-3 (referred to as *Original*). 3) We use S^{2}E to execute modified NS-3 by eliminating unnecessary comparisons due to Simultaneous events (Referred to as *technique S*). 4) We use S^{2}E to execute modified NS-3 by eliminating unnecessary comparisons due to Conditional ineffective events (Referred to as *technique C*). 5) We use S^{2}E to execute modified NS-3 by eliminating unnecessary comparisons due to Independent events (Referred to as *technique I*). 6) Different combinations of techniques S, C, and I. For example, SCI means that all three techniques are used. 7) We use S^{2}E to execute NS-3 using parallel simulation methods. NS3 itself supports both sequential and parallel simulation methods. However, we find that the parallel simulation methods of NS-3 do not work under S^{2}E, because the communication messages sent by their synchronization mechanisms do not support symbolic variables. Therefore, we have implemented two popular parallel simulation methods using shared variables instead of communication messages for synchronization: the global safe window method (referred to as *technique P_g*) and the null message method (referred to as *technique P_n*) [9].

We run each testing method for each experiment for at most one day on virtual machines configured with a 2.3GHz 4-Core processor, 64 GByte RAM, and Ubuntu 14.04. The simulation scripts used in the experiments are selected from the example scripts provided in the NS-3. We keep all the network topologies and parameter settings in the original simulation scripts, and we add uncertain packet delay to a group of selected packets.

B. UDP experiments: Multiple nodes

This group of experiments use the simple-error-model.cc script of NS-3. There are a total of four nodes generating a total of about 2000 packets in the simulation. We introduce uncertain delays for $n = 1, 2, 3, 4$ packets from node 0 to node 2. Each of these n packets has an uncertain delay in $D = [1, 1024]$ ms with a millisecond resolution, and all other packets still have their delays specified in the script.

Fig. 4 shows the number of branches generated by methods Brute, Original, and SCI, and Fig. 5 shows their total testing times. For Brute, the number of branches is the number of individual NS-3 simulations. For example, with $n = 1$, Brute runs 1024 simulations, and takes 25 minutes. With $n = 2$, Brute needs to run about 10^6 simulations, and takes about 17 days. We can see that *Original has several orders of magnitude less numbers of branches and shorter testing times than Brute, and SCI has even several orders of magnitude less numbers of branches and shorter testing times than Original.* For example, with $n = 4$, Original takes several days (thus not shown in the figures), and SCI takes only 94 minutes.

To understand the efficiency of each technique, Table I shows the number of branches generated by each technique and different combinations. Symbol ⥁ means that the test could not finish in one day. We can see that each of our techniques is more efficient than Original. Especially, technique I is more efficient than techniques S and C in the UDP experiments, because there are four nodes and then many independent events on different nodes.

By comparing the results of SCI with SCP$_g$ and SCP$_n$ in Table I, we can also see that our technique I has several orders of magnitude less number of branches than the two popular parallel simulation methods P$_g$ and P$_n$. This is because they are designed for speeding up parallel simulation and have a large number of comparisons of timestamps.

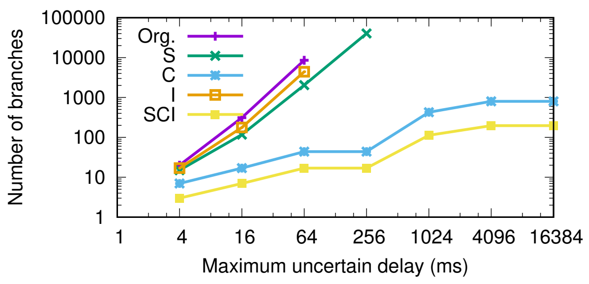


Fig. 6. Number of branches in the TCP experiments.

C. TCP experiments: Timeout events

This group of experiments use the tcp-bulk-send.cc script of NS-3. There are two nodes connected over a link. There is a TCP connection between two nodes, and a total of about 1500 packets are generated in the simulation. We introduce uncertain delays for three packets. Each of these three packets has an uncertain delay in $D = [1, d^{max}]$ ms with a millisecond resolution, and all other packets still have the delays specified in the script. We vary the maximum uncertain delay d^{max} from 4 ms to 16,384 ms.

Fig. 6 shows the number of branches generated by each testing method. We can see that each of our techniques is more efficient than Original. But technique C is more efficient than techniques S and I. This is because there are a large number

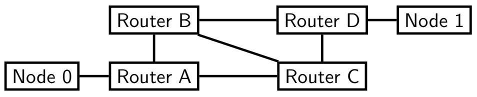


Fig. 7. Network topology of the IP routing experiments.

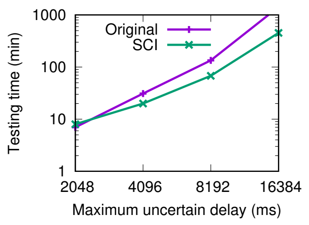


Fig. 8. Testing time of the IP routing experiments.

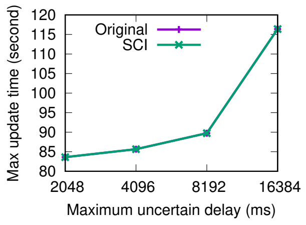


Fig. 9. Longest update time of the routing tables in the IP experiments.

of TCP retransmission timeout events, which are canceled by ACK and become ineffective events. Again we see that SCI is several orders of magnitude more efficient than Original.

D. IP routing experiments: A use case

This group of experiments demonstrates correctness testing and worst-case performance evaluation of a routing protocol under uncertain events. We use the rip-simple-network.cc script of NS-3. There are six nodes including four routers interconnected over a network shown in Fig. 7. The routers communicate with one another to run the RIP routing protocols. A total of about 100 IP packets are generated in the simulation. The link between routers B and D is broken at 40 seconds. All the packets from routers C to A after 40 seconds have uncertain delays in $D = [1, d^{max}]$ ms, and all other packets still have the delays specified in the script. We vary the maximum uncertain delay d^{max} from 2,048 to 16,384 ms.

First, if the routing protocol works correctly, all the routing tables will be correctly updated. Fig. 8 shows the testing times for Original and SCI. After the test, every Original's branch and every SCI's branch report that all the routing tables have been correctly updated. That is, the routing protocol works correctly under all possible cases of these uncertain events.

Second, for each d^{max}, we measure the longest time for correctly updating all routing tables among all possible cases of these uncertain events. Fig. 9 shows the results of Original and SCI. We can see that Original and SCI report the same result. This is expected and implies that SCI generates the same simulation result as Original. Overall, we can see that SCI can be used to test the correctness and evaluate the worst-case performance of a network protocol, while it takes significantly shorter time than Original as shown in Fig. 8.

VI. Related work

The related work on simulator-based testing methods has been discussed in the Introduction section. In addition to those, we note that implementation-level model checking of a network protocol [11] recursively explores the protocol states by attempting all possible events at each state. The efficiency of SEIB can be further improved by combining [5] with model checking. Existing work on SEIB, such as SPD [17], [18] and SymTime [8], [14] also consider the efficiency of SEIB. But SPD focuses on prioritizing different branches, and SymTime focuses on compressed representation of node states among different branches. In contrast, we focus on reducing the total number of branches. There are also various techniques [1], [16], [2] to improve the efficiency by modifying symbolic execution engines (e.g., S^2E). These techniques are complementary to our work that modifies the network simulators.

VII. Conclusions

In this paper, we propose three general techniques to modify a discrete-event network simulator, which can significantly improve the efficiency of SEIB for testing network protocol implementations.

ACKNOWLEDGMENT

The work presented in this paper was supported in part by NSF CNS-1526253 and NSF SHF-1718040.

References

[1] T. Avgerinos, A. Rebert, S. Cha, and D. Brumley. Enhancing symbolic execution with veritesting. In *Proceedings of International Conference on Software Engineering*, Hyderabad, India, June 2014.
[2] S. Bugrara and D. Engler. Redundant state detection for dynamic symbolic execution. In *Proceedings of USENIX ATC*, San Jose, CA, June 2013.
[3] C. Cadar, D. Dunbar, and D. Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of USENIX OSDI*, San Diego, CA, December 2008.
[4] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, February 2013.
[5] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford. A NICE way to test OpenFlow applications. In *Proceedings of USENIX NSDI*, San Jose, CA, April 2012.
[6] N. Cardwell, Y. Cheng, L. Brakmo, M. Mathis, B. Raghavan, N. Dukkipati, H. Chu, A. Terzis, and T. Herbert. PacketDrill: Scriptable network stack testing, from sockets to packets. In *Proceedings of USENIX ATC*, San Jose, CA, June 2013.
[7] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: design, implementation, and applications. *ACM Transactions on Computer Systems*, 30(1), February 2012.
[8] O. Dustmann. Symbolic execution of discrete event systems with uncertain time. *Lecture Notes in Informatics*, S-12:19–22, 2013.
[9] R. Fujimoto. Parallel discrete event simulation. *Communications of the ACM*, 33(10):30–53, October 1990.
[10] P. Li and J. Regehr. T-Check: Bug finding for sensor networks. In *Proceedings of ACM/IEEE IPSN*, Stockholm, Sweden, April 2010.
[11] M. Musuvathi and D. Engler. Model checking large network protocol implementations. In *Proceedings of USENIX NSDI*, San Francisco, CA, March 2004.
[12] Network Simulator 3. https://www.nsnam.org/.
[13] P. Peschlow, P. Martini, and J. Liu. Interval branching. In *Proceedings of ACM Workshops on Principles of Advanced and Distributed Simulation*, Rome, Italy, June 2008.
[14] R. Sasnauskas, O. Dustmann, B. Kaminski, K. Wehrle, C. Weise, and S. Kowalewski. Scalable symbolic execution of distributed systems. In *Proceedings of ICDCS*, Minneapolis, MN, June 2011.
[15] R. Sasnauskas, O. Landsiedel, M. Alizai, C. Weise, S. Kowalewski, and K. Wehrle. KleeNet: discovering insidious interaction bugs in wireless sensor networks before deployment. In *Proceedings of ACM/IEEE IPSN*, Stockholm, Sweden, April 2010.
[16] K. Sen, G. Necula, L. Gong, and W. Choi. MultiSE: Multi-path symbolic execution using value summaries. In *Proceedings of ESEC/FSE*, Italy, October 2015.
[17] W. Sun, L. Xu, and S. Elbaum. SPD: Automatically test unmodified network programs with symbolic packet dynamics. In *Proceedings of IEEE Globecom*, San Diego, CA, December 2015.
[18] W. Sun, L. Xu, and S. Elbaum. Improving the cost-effectiveness of symbolic testing techniques for transport protocol implementations under packet dynamics. In *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, Santa Barbara, CA, July 2017.