Spring 5-2022

Symbolic NS-3 for Efficient Exhaustive Testing

Jianfei Shao
University of Nebraska-Lincoln, jianfei.shao@huskers.unl.edu

SYMBOLIC NS-3 FOR EFFICIENT EXHAUSTIVE TESTING

by

Jianfei Shao

A THESIS

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Master of Science

Major: Computer Science

Under the Supervision of Professor Lisong Xu

Lincoln, Nebraska

May, 2022

SYMBOLIC NS-3 FOR EFFICIENT EXHAUSTIVE TESTING

Jianfei Shao, M.S.

University of Nebraska, 2022

Adviser: Lisong Xu

Exhaustive testing is an important type of simulation, where a user exhaustively simulates a protocol for all possible cases with respect to some uncertain factors, such as all possible packet delays or packet headers. It is useful for completely evaluating the protocol performance, finding the worst-case performance, and detecting possible design or implementation bugs of a protocol. It is, however, time consuming to use the brute force method with current NS-3, a widely used network simulator, for exhaustive testing. In this paper, we present our work on Sym-NS-3 for more efficient exhaustive testing, which leverages a powerful program analysis technique called symbolic execution. Intuitively, Sym-NS-3 groups all the cases leading to the same simulator execution path together as an equivalence class, and simulates a protocol only once for each equivalence class. We present our design choices and implementation details on how we extend current NS-3 to support symbolic execution, and also present several exhaustive testing results to demonstrate the significantly improved testing speeds of Sym-NS-3 over current NS-3.

DEDICATION

First and foremost, I would like to thank my advisor Dr. Lisong Xu. He paid a lot of time and patience during my master's program. Especially during the pandemic, he not only took care of my research, but also my life. It would be impossible for me to complete this thesis without Dr. Xu.

Next, I would like to show my gratitude to my committee members, Dr. Byrav Ramamurthy and Dr. Hamid Bagheri. They have provided very important guidance and feedback on my thesis.

In addition, I am indebted to all my lab members, Nan Jiang, Tianqi Fang, Minh Vu, Phuong Ha, and Mingrui Zhang. Thank you all for the accompany and support.

Besides, I also want to thank all my friends I met in UNL these years.

Most importantly, I want to thank my parents, Hong Shao and Minzhu Tang for their love and support.

ACKNOWLEDGMENTS

The work presented in this thesis was supported in part by NSF CCF-1918204.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

NS-3 is a popular network simulator that has been widely used in the networking community. It is usually used to evaluate the normal-case or special-case performance of a network protocol, where a user simulates the protocol for some normal or special cases. In this thesis, we consider an important type of simulation (referred to as *exhaustive testing* hereinafter), where a user exhaustively simulates a protocol for all possible cases with respect to some uncertain factors, such as all possible packet delays or headers. Exhaustive testing is useful for completely evaluating the performance of a protocol for all possible cases, finding the worst-case performance of a protocol among all possible cases, and for detecting possible design or implementation bugs of a protocol as many bugs happen only in corner cases.

It is, however, time consuming to use the current NS-3 for exhaustive testing, because a user needs to enumerate and simulate a protocol in each possible case (referred to as the *brute force* method). Let's consider a simple motivating example, where we need to exhaustively test a protocol in the network shown in Fig. 1.1, where the propagation delay d_i of link $i = 0, 1$ could be any value of between 1 ms and 1000 ms with a resolution of 1 ms. Thus, the test space (d_0, d_1) contains a total of $10^3 \times 10^3 = 10^6$ possible testing cases.

The brute force method runs NS-3 to enumerate and individually simulate each of the 10^6 cases in the test space, and thus takes a long time. The details can be found in Chapter 3.

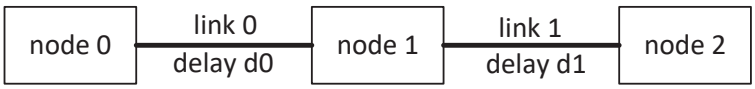

Figure 1.1: Network topology of the motivating example.

In this thesis, we present our work on *Symbolic NS-3* (Sym-NS-3 for short), which extends NS-3 to support the *symbolic execution method* [20, 8] that is a powerful and popular program analysis technique widely used in the software testing and verification community. Intuitively, the symbolic execution method divides the test space (d_0, d_1) into equivalence classes, each equivalence class containing all the cases leading to the same simulator execution path. The symbolic execution method simulates all the cases in an equivalence class together instead of individually as the brute force method. By doing so, the symbolic execution method can more efficiently exhaustively test the same test space (d_0, d_1) than the brute force method. The details can also be found in Chapter 3.

We make the following contributions in this thesis:

- *Symbolic variable management*: We present our design choices and implementation of the symbolic variables in Sym-NS-3 that are the foundation of symbolic execution. A symbolic variable takes a set of values instead of a single value as a normal variable. We have explored multiple different ways to introduce symbolic variables to Sym-NS-3 so that Sym-NS-3 users can easily manage symbolic variables and Sym-NS-3 developers can easily maintain and upgrade Sym-NS-3.

- *More efficient packet semantics testing*: There are two types of exhaustive testing: 1) packet dynamic testing: checking a protocol with all possible packet dynamic (e.g., delays), 2) packet semantics testing: checking a protocol for all possible packet header

and payload semantics. We have already presented several techniques to further improve the testing speed of Sym-NS-3 for packet dynamic testing in our previous work [33], and we present several techniques to further improve the testing speed of Sym-NS-3 for packet semantics testing in this thesis.

- *Simulations*: We present several exhaustive testing simulations to demonstrate how to use Sym-NS-3 and the significantly improved testing speeds of the symbolic execution method using Sym-NS-3 compared to the brute force method using current NS-3.

Chapter 2

Related work

Symbolic execution has been used to test network protocols. KleeNET [27], SymTime [14], SPD [32], and Chiron [18] test network protocols with symbolic packet delays or loss. DiCE [9], SymbexNet [29], NICE [10], SOFT [23], Chiron [18], BUZZ [15], SymNet [31], PIC [24], and MAX [21] test network protocols with symbolic packet headers. Different from these works that consider only simple and specific network environments, Sym-NS-3 attempts to support general network environments by leveraging NS-3.

S^2E [11] is a powerful, modular and flexible symbolic execution platform, which uses KLEE and QEMU to serve both Windows and Linux. Since S^2E is written in C, it is convenient to combine with NS-3 written in C++ as Sym-NS-3.

There is little work on extending NS-3 for exhaustive testing. VeriSim [4] extends NS-2 for formal trace analysis. To the best our knowledge, Sym-NS-3 is the only one to extend NS-3 by leveraging symbolic execution.

There is a large body of work on improving the efficiency of symbolic execution engines, such as compositional symbolic execution [16, 13], redundant path elimination [5], path prioritization using static analysis information [6, 2], path merging [22, 1, 28], state mapping methods [25, 26], and combination with random testing [17, 30, 12]. Sym-NS-3 is

complementary to and can be used together with these techniques.

Chapter 3

Motivating Example

In this chapter, we present a simple exhaustive testing example to illustrate the difference between the brute force method of current NS-3 and the symbolic execution method of our proposed Sym-NS-3. All the code is available at https://github.com/JeffShao96/Symbolic-NS3.

3.1 An exhaustive testing problem

Let's consider a network shown in Fig. 1.1, where three nodes are connected by two point-to-point links. Nodes 0 and 2 each simultaneously sends a UDP packet to node 1. The propagation delay d_i of link $i = 0, 1$ could be any value between 1 ms and 1000 ms. An exhaustive testing problem is to find the maximum and minimum of *diff* among a total of 10^6 testing cases (i.e., combinations) of d_0 and d_1, where *diff* is the arrival time difference at node 1 between the packets from node 0 and node 2.

3.2 Brute force using current NS-3

To find the range of *diff* using the brute force method with the current NS-3, we write the shell script `repeatCurrentDemo.sh` as shown in Code 3.1 to enumerate all possible 10^6 cases of d_0 and d_1, and run an NS-3 simulation for each case.

Code 3.1: Shell script repeatCurrentDemo.sh to enumerate all possible cases

```
...
for delay0 in {1..1000}
do
  for delay1 in {1..1000}
  do
    ./waf --run "currentDemo --d0=$delay0 --d1=$delay1"
  done
done
...
```

We also write the NS-3 script `currentDemo.cc` as shown in Code 3.2 to simulate the network according to the link delays specified in the arguments. To simplify the example, we calculate *diff* directly in this script instead of modifying file `udp-server.cc` to measure the packet arrival times and then calculate *diff* at the receiver.

Code 3.2: NS-3 script currentDemo.cc to simulate each case

```
...
p2p[0].SetChannelAttribute("Delay",TimeValue(Time(d0)));
p2p[1].SetChannelAttribute("Delay",TimeValue(Time(d1)));
...
...
Time Diff = Time(d0)-Time(d1);
std::cout<<"Diff is "<<Diff<<std::endl;
...
```

Overall, it takes a total of about six days to run the code, and the total reported range of *diff* is [-999, 999] ms.

3.3 Symbolic execution using Sym-NS-3

To find the range of *diff* using the symbolic execution method with our Sym-NS-3, we write only one script `symDemo.cc` as shown in Code 3.3 to simulate the network with two symbolic link delays, each in the range of [1, 1000] ms. To simplify the example, we calculate *diff* directly in the script instead of modifying `udp-server.cc`.

Code 3.3: Sym-NS-3 script symDemo.cc

```
1  symDemo.cc},label={lst:sym_ns3}]
2  ...
3  Ptr<Symbolic> sym0 = CreateObject<Symbolic>();
4  sym0->SetMinMax(1, 1000);
5  uint32_t d0 = sym0->GetSymbolicUintValue();
6  Ptr<Symbolic> sym1 = CreateObject<Symbolic>();
7  sym1->SetMinMax(1, 1000);
8  uint32_t d1 = sym1->GetSymbolicUintValue();
9  ...
10 p2p[0].SetChannelAttribute("Delay",TimeValue(Time(d0)));
11 p2p[1].SetChannelAttribute("Delay",TimeValue(Time(d1)));
12 ...
13 ...
14 Symbolic Diff=sym0-sym1;
15 Diff.PrintRange("Diff");
```

It takes a total of about a minute to execute the code using a symbolic execution engine, which is several orders of magnitude faster than the brute force method. The total reported range of *diff* is also [-999, 999] ms, the same as the range reported by the brute force method.

Chapter 4

Sym-NS-3 Overview

4.1 Architecture of Sym-NS-3

Fig. 4.1 illustrates the different architectures of the brute force method with current NS-3 and the symbolic execution method with Sym-NS-3. Both current NS-3 and Sym-NS-3 use NS-3 style script to execute in the experiment. Different from the brute force method that directly executes NS-3, the symbolic execution method uses symbolic execution platform S^2E [11] to symbolically execute Sym-NS-3 in virtual machines. S^2E emulates the virtual machines using the QEMU machine emulator [3] and conducts symbolic execution using the KLEE symbolic execution engine [7]. Instead of execute directly, S^2E passes NS-3 style script from Host operating system to the guest operating system on the virtual machine. Different from the brute force method where each variable can take only a value at a time, the symbolic execution method introduces symbolic variables, each of which can take a set of values described by a group of constraints.

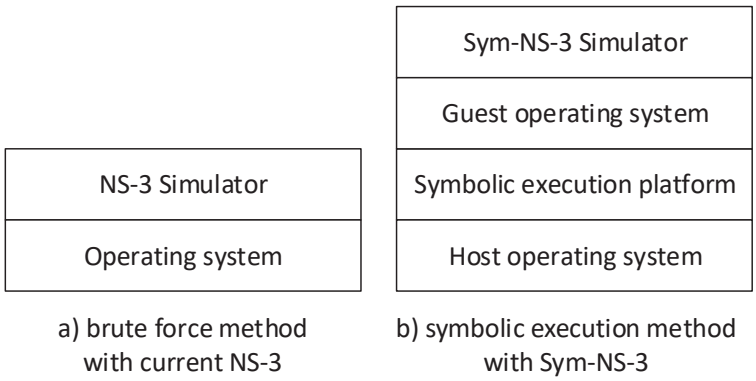

Figure 4.1: Architectures of the brute force and symbolic execution methods.

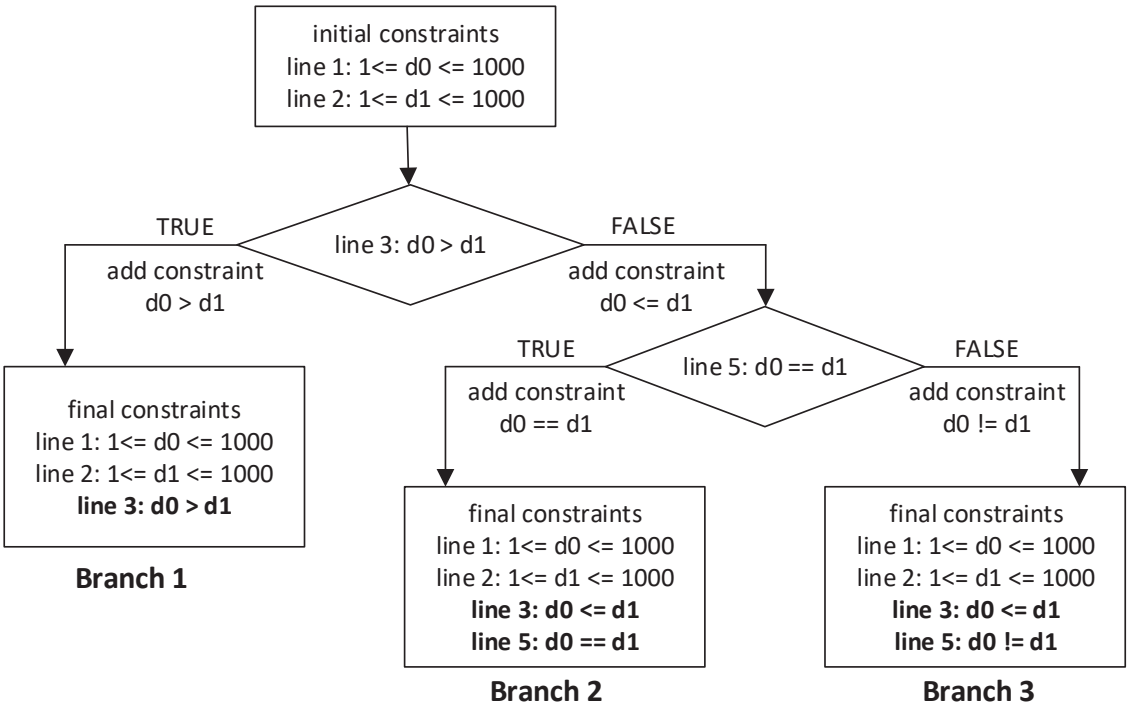

Figure 4.2: Three branches are generated during the symbolic execution of Code 4.1.

4.2 Symbolic execution

We use the C-like pseudocode shown in Code 4.1 as an example to explain how S^{2}E works. S^{2}E initially runs the code on a single virtual machine. Lines 1 and 2 define two symbolic variables d_0 and d_1 with the same initial constraints, and thus each of them initially takes

a set of values in the range of 1 and 1000. When S^{2}E reaches an `if` statement involving symbolic variables, such as lines 3 and 5, it checks both possibilities by forking the current virtual machine into two virtual machines (called *branches*). For example, when the `if` statement at line 3 is executed, S^{2}E forks the current virtual machine into two virtual machines, where the true branch continues to line 4 with additional constraint $d_0 > d_1$ and the false branch continues to line 5 with additional constraint $d_0 <= d_1$. Similarly for the `if` statement at line 5.

Finally, S^{2}E stops with three branches as illustrated in Fig. 4.2, where the final constraints for each branch are also listed. Using these final constraints, S^{2}E can then calculate the possible range of variable *diff* defined in line 10. Specifically, the range of *diff* is [1, 999] ms for branch 1, [0, 0] ms for branch 2, and [-999, -1] ms for branch 3. The total range of *diff* is the union of these ranges and thus is [-999, 999] ms.

Code 4.1: An example for symbolic execution

```
1  sym 1<= d0 <= 1000
2  sym 1<= d1 <= 1000
3  if (d0 > d1){
4    //simulate accordingly
5  }else if (d0==d1){
6    //simulate accordingly
7  }else{
8    //simulate accordingly
9  }
10 diff = d0 - d1;
```

4.3 Design goals

Sym-NS-3 is designed with the following design goals.

4.3.1 Easy to use

It is easy for current NS-3 users to use Sym-NS-3 for exhaustive testing. Specifically, a Sym-NS-3 user writes a Sym-NS-3 testing script in a way very similar to a NS-3 user writing an NS-3 testing script.

4.3.2 Easy to develop

It is easy for Sym-NS-3 developers to maintain and upgrade Sym-NS-3. Specifically, Sym-NS-3 makes as little change as possible to current NS-3, especially, existing NS-3 modules.

4.3.3 Efficient

It is more efficient to conduct exhaustive testing using Sym-NS-3 than current NS-3. Although symbolic execution already makes Sym-NS-3 more efficient for exhaustive testing than brute force with current NS-3, we propose several techniques to further improve the efficiency of Sym-NS-3.

Chapter 5

Symbolic Variable Management

In this chapter, we describe how to design Sym-NS-3 so that a Sym-NS-3 user can easily use symbolic variables (i.e., the first design goal) and a Sym-NS-3 developer can easily develop Sym-NS-3 (i.e., the second design goal).

5.1 Managing symbolic variables

An exhaustive testing simulates a network in all possible cases with respect to some uncertain factors, which can be tested using symbolic variables in Sym-NS-3. For example, the motivating example simulates a network for all possible link delays d_0 and d_1, and they are tested using symbolic variables in Chapter 3.3.

There are two different ways to change a normal variable to a symbolic variable in Sym-NS-3.

- *Direct Symbolization*: We can use S^{2}E functions to make a normal variable symbolic. For example, function `s2e_make_symbolic(&x,sizeof(x),"x")` makes variable `x` a symbolic variable by marking `sizeof(x)` number of bytes at address `&x` symbolic.

- *Assignment Symbolization*: If a normal variable `y` is set to the value of an expression involving a symbolic variable `x`, variable `y` also becomes a symbolic variable. For example, if symbolic variable `x` has a symbolic value between 1 and 1000, variable `y` will have a symbolic value between 2 and 1001 after executing assignment `y = x+1`.

We need to provide users with the following types of functions to manage the symbolic variables.

- *Initialization functions*: We need to provide users with functions to set the initial constraint of a symbolic variable. For example, d_0 in the motivating example should be defined as a symbolic variable with an initial constraint of between 1 and 1000 ms.

- *Operation functions*: We need to provide users with functions to operate on symbol variables, such as functions to perform math operations (e.g., addition, subtraction) of symbolic variables, and functions to change the type of a symbolic variable (e.g., change from `unsigned` to `time`).

- *Inquiry functions*: Different from a normal variable that takes a single value, a symbolic variable takes a set of values described by a group of constraints. Furthermore, the constraints of a symbolic variable may change as the simulation continues. For example, the three branches in Fig. 4.2 each has a different group of constraints for symbolic variables d_0 and d_1. Thus, we need to provide functions for users to inquire and print out the current range of a symbolic variable, such as the max, min, and sample values.

We have explored three different methods to manage the symbolic variables in Sym-NS-3. Below we explain these methods using the propagation delay of link 0 in the motivating example, which is a point to point channel. The propagation delay of `PointToPointChannel`

in NS-3 is defined as a `Time` variable named `m_delay`. We have explored three different methods to make `m_delay` symbolic in Sym-NS-3.

5.2 Method 1: In-module direct symbolization

This method directly modifies existing NS-3 modules that are involved in an exhaustive test. For the motivating example, this method directly modifies `PointToPointChannel` by adding new channel attributes and modifying its code accordingly. For example, Code 5.1 shows a different script `symDemo.cc` implemented using this method. Specifically, we add three new channel attributes `SymbolicMode`, `DelayMin`, and `DelayMax`. If attribute `SymbolicMode` is true, function `s2e_make_symbolic(&m_delay,sizeof(m_delay),"m_delay")` is called to make `m_delay` symbolic, and then its minimum and maximum values are set to `DelayMin`, and `DelayMax`, respectively.

Code 5.1: Sym-NS-3 script symDemo.cc using Method 1

```
1 ...
2 p2p[0].SetChannelAttribute("SymbolicMode", BooleanValue(true));
3 p2p[0].SetChannelAttribute("DelayMin", TimeValue(Time("1ms")));
4 p2p[0].SetChannelAttribute("DelayMax", TimeValue(Time("1000ms")));
5 ...
```

We first adopted this method at the beginning of our project, because the advantage of this method is that it can flexibly implement more module-specific functionalities. For example, instead of all the packets on the link experiencing the same symbolic delay `m_delay`, we can introduce a new attribute to specify only a certain type of packets experiencing the symbolic delay (e.g., only data packets of a specific TCP flow), and a new attribute to specific that different packets experiencing different symbolic delays (e.g., to introduce packet reordering).

We later explored other methods, because this method has two disadvantages that make it hard to develop (i.e., the second design goal). First, this method makes a big change to an NS-3 module, because we need to modify and add many functions to the NS-3 module in order to add and implement all the new channel attributes and to implement the initialization, operation, and inquiry functions to manage the symbolic variables. Second, this method makes big changes to many NS-3 modules, because we have to modify each NS-3 module for which we need to add symbolic variables. For example, if we want to exhaustively test other channels, such as wireless channels, we need to modify all these channels.

5.3 Method 2: Assignment symbolization using new attributes

Method 2 makes less change to existing NS-3 modules than method 1 by defining and managing symbolic objects using a new class called `Symbolic`, which we have developed for Sym-NS-3. For example, Code 5.2 shows a different script `symDemo.cc` implemented using this method. It first creates a symbolic variable `symObj0` which is a `Symbolic` class variable. The `Symbolic` class implements all the functions to manage symbolic variables, such as initialization, operation, and inquiry functions, so that existing NS-3 modules do not need to implement these functions as in Method 1. For example, line 3 sets the initial range of `symObj0` using function `SetMinMax`.

Method 2 still changes `PointToPointChannel` by adding a new attribute `SymbolicDelay`, which takes a pointer value of `symObj0` whose value is then assigned to variable `m_delay`.

The advantage of this method is that it can flexibly implement the same module-specific functionalities as Method 1 while making less changes to existing NS-3 modules

than Method 1. For example, similar to Method 1, instead of all the packets on the link experiencing the same symbolic delay `m_delay`, Method 2 can also introduce a new attribute to specify only a certain type of packets experiencing the symbolic delay (e.g., only data packets of a specific TCP flow). But different from Method 1, Method 2 does not need to change `PointToPointChannel` to implement the initialization, operation, and inquiry functions to manage symbolic variables.

Code 5.2: Sym-NS-3 script symDemo.cc using Method 2

```
1 ...
2 Ptr<Symbolic> symObj0 = CreateObject<Symbolic>();
3 symObj0->SetMinMax(1, 1000);
4 ...
5 p2p[0].SetChannelAttribute ("SymbolicDelay", PointerValue (symObj0));
6 ...
```

The disadvantage of this method is that it still makes changes (although just adding new attributes) to many NS-3 modules, because we have to modify each NS-3 module for which we need to add symbolic variables.

5.4 Method 3: Assignment symbolization using existing attributes (as illustrated in the motivating example)

This method does not modify existing NS-3 modules at all. Because it does not add any new attributes to an NS-3 module, we have to use the existing attributes. For example, Code 5.3 shows script `symDemo.cc` implemented using this method, which is just the one illustrated in the motivating example in Chapter 3.3. It first creates the same symbolic variable `symObj0` as Method 2. But it gets the corresponding symbolic unsigned integer `d0` from `symObj0`, and then passes `d0` to `p2p[0]` (i.e., link 0) using the exist-

ing `PointToPointChannel` attribute `Delay`. Note that variable `d0` is also a symbolic variable with the same set of values as `symObj0`.

The advantage of this method is that it does not make any changes to the existing NS-3 modules. As a result, is very easy for Sym-NS-3 developers to maintain and upgrade Sym-NS-3 (i.e., the second design goal), and it is also easy to apply this method to any NS-3 modules and any module attributes, in addition to `PointToPointChannel` and its `Delay` attribute used in the example.

Code 5.3: Sym-NS-3 script symDemo.cc using Method 3

```
1 ...
2 Ptr<Symbolic> symObj0 = CreateObject<Symbolic>();
3 symObj0->SetMinMax(1, 1000);
4 uint32_t d0 = symObj0->GetSymbolicUintValue();
5 ...
6 p2p[0].SetChannelAttribute("Delay",TimeValue(Time(d0)));
7 ...
```

The disadvantage of this method is that it uses only the existing attributes of NS-3 modules, and thus supports only limited module-specific functionalities. For example, all the packets on the link have to experience the same symbolic delay, and we cannot specify only a certain type of packets experiencing the symbolic delay.

5.5 Comparison of the three methods

Fig. 5.1 compares the above three methods to manage symbolic variables. Method 1 makes the biggest changes to existing NS-3 modules, whereas Method 3 does not make any changes. On the other side, Methods 1 and 2 support more module-specific functionalities than Method 3 that supports only the basic module-specific functionalities.

While all three methods make Sym-NS-3 easy to use for current NS-3 users (i.e., the first design goal), Method 3 is the easiest for the Sym-NS-3 developers to develop (i.e., the

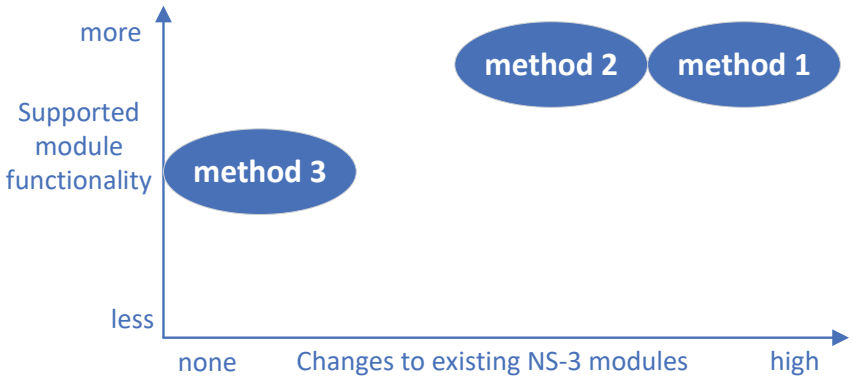

Figure 5.1: Different methods to manage symbolic variables in Sym-NS-3.

second design goal). Therefore, for the current release of Sym-NS-3, we choose Method 3 so that symbolic variables can be used with all current NS-3 modules.

Chapter 6

Making Sym-NS-3 More Efficient

Although the methods proposed in the previous chapter already make Sym-NS-3 more efficient for exhaustive testing than current NS-3, we have noticed that we can make Sym-NS-3 even more efficient by redesigning some of NS-3 modules (i.e., the third design goal). Intuitively, this is because NS-3 was not originally designed and implemented for symbolic execution, and thus we have proposed some techniques to redesign and make it symbolic execution friendly.

We have proposed two types of techniques for two general types of exhaustive testing using Sym-NS-3. 1) *Exhaustive packet dynamic testing*: It tests a network protocol in a network with all possible packet dynamic, such as all possible packet delays in the motivating example. For this type of testing, Sym-NS-3 changes some time-related variables to symbolic variables, such as d_0 in the motivating example. As a result, the timestamps of events also become symbolic variables. In our previous work [33], we have proposed several techniques to redesign the event schedulers of Sym-NS-3 so that it can more efficiently compare the symbolic timestamps of the events.

2) *Exhaustive packet semantics testing*: It tests a network protocol for packets with all possible header and payload semantics, such as all possible destination IP addresses. For

this type of testing, Sym-NS-3 changes the packet header fields or packet payload to symbolic variables. In this thesis, we consider the destination IP address field of a packet, which is one of the most important fields of a packet header. Below, we propose two techniques to redesign the IP routing protocol of Sym-NS-3 so that it can more efficiently handle packets with symbolic destination IP addresses.

6.1 Symbolic IP address

A symbolic IP address can be used for exhaustive testing of a packet with a set of destination IP addresses. For example, a reachability test [19, 31] checks whether a packet from a node can reach another node. However, it is time consuming to find all possible nodes that can be reached by a packet from a node, if we exhaustively try all possible destination IP addresses for the packet. With the help of a symbolic destination IP address, we can more efficiently find all possible nodes that can be reached from a node. Code 6.1 shows how a symbolic IP address can be defined in Sym-NS-3.

Code 6.1: Defining a symbolic IP address in Sym-NS-3

```
1 Ptr<Symbolic> symObj0 = CreateObject<Symbolic>();
2 Ipv4Address symIP0 = symObj0->GetSymbolicIpv4Add();
```

6.2 How current NS-3 simulates IP routing?

We describe how current NS-3 simulates IP routing table in this section, and then explain why it is not friendly to symbolic execution and how we modify it in the following sections. Below we briefly demonstrate how current NS-3 maintains a routing table, checks a table entry for possible match, and finds the best match for the whole table, as each of them will be redesigned in Sym-NS-3.

NS-3 maintains an unsorted routing table, where a new table entry is added to the end of the table. For example, Table 6.1 shows a possible routing table at a node.

Table 6.1: Routing table example of current NS-3

Destination	Mask	Interface	Metric
127.0.0.0	255.0.0.0	0	1
0.0.0.0	0.0.0.0	1	1
10.1.0.0	255.255.0.0	2	10
10.2.0.0	255.255.0.0	2	10

For a table entry `entry` with network destination `entry.ip` and mask `entry.mask`, NS-3 checks whether the destination IP address `dst` of a packet matches the entry using function `IsMatch` as illustrated in Pseudocode 6.2.

Code 6.2: Pseudocode of function IsMatch in NS-3

```
1 IsMatch (IP Address dst, Table Entry entry)
2   if ((dst & entry.mask) == (entry.ip & entry.mask))
3       return true;
4   else
5       return false;
```

Code 6.3: Pseudocode of function Lookup in NS-3

```
1 Lookup (IP Address dst, IP Table table)
2   for each entry in the table
3     if IsMatch(dst, entry)
4         if (entry.masklen > bestmatch.masklen)
5             bestmatch = entry;
6         else if ((entry.masklen = bestmatch.masklen) and
7                 (entry.metric < bestmatch.metric))
8             bestmatch = entry;
9   return bestmatch;
```

NS-3 checks every table entry to find the best match, which is the entry with the longest mask among all matching entries. If there are multiple matching entries with the same

longest length of masks, the one with the shortest metric is the best match. The code is illustrated as function `Lookup` in Pseudocode 6.3. For example, if `dst`=10.1.0.1, there are two matching entries in Table 6.1: the entry for 10.1.0.0 and the entry for 0.0.0.0. Because the former has a longer mask than the latter, the best match is the former.

6.3 Why current NS-3 is not symbolic execution friendly?

The efficiency of symbolic execution mainly depends on the number of *symbolic comparisons* that are the conditional statements involving symbolic variables, such as lines 3 and 5 in Code 4.1. There are two reasons. First, each symbolic comparison takes a non-trivial amount of time for the constraint solver of symbolic execution to determine whether the symbolic comparison is true or false or both with the current branch constraints. Second, if the symbolic comparison could be both true and false, symbolic execution forks the current branch (i.e., virtual machine) into two branches, a true branch and a false branch with correspondingly updated constraints. However, branch forking (i..e, virtual machine forking) takes a significant amount of time and space.

The IP routing simulation of current NS-3 is not friendly to symbolic execution, because it compares a symbolic destination IP address with each entry of a routing table. As an example, if we symbolically run Pseudocode 6.3 on Table 6.2 with a symbolic destination IP address `dst`, then there are 4 symbolic comparisons in Pseudocode 6.3 because it calls line 2 of Pseudocode 6.2 for each of 4 table entries. If the range of `dst` is from 10.1.0.0 to 10.2.255.255, there are two best matches: entry 10.1.0.0 and entry 10.2.0.0. As a result, there are finally two branches:

- one branch returns entry 10.1.0.0 (interface 2) for `dst` between 10.1.0.0 and 10.1.255.255,
- the other branch returns entry 10.2.0.0 (interface 2) for `dst` between 10.2.0.0 and

10.2.255.255.

6.4 Proposed techniques for more efficient IP simulations in Sym-NS-3

We redesign the simulation of IP routing in Sym-NS-3 to make it more friendly to symbolic execution and thus more efficient. Specifically, we propose two techniques: 1) the table sorting technique that reduces the number of symbolic comparisons and thus reduces the number of times to call the constraint solver, and 2) the group comparison technique that reduces the number of branches (i.e., virtual machines).

The *table sorting technique* sorts a routing table according to the priority of each table entry. An entry with a longer mask is given a higher priority. If tie occurs, a shorter metric is given a higher priority. If tie still occurs, the interface is used to break the tie. All the entries with the same length of masks, same metric, and same interface belong to the same priority group. For example, Table 6.2 shows the sorted result of Table 6.1. With a sorted routing table, Sym-NS-3 only needs to find the first matching priority group, but does not need to check all the remaining entries. By doing so, Sym-NS-3 can reduce the number of symbolic comparisons.

Table 6.2: Sorted routing table in Sym-NS-3

Destination	Mask	Interface	Metric	Priority group
10.1.0.0	255.255.0.0	2	10	1
10.2.0.0	255.255.0.0	2	10	1
127.0.0.0	255.0.0.0	0	1	2
0.0.0.0	0.0.0.0	1	1	3

The *group comparison technique* checks all the table entries within the same priority group

together using only one symbolic comparison. Specifically, it replaces function `IsMatch` in Pseudocode 6.2 with Pseudocode 6.4, which returns the same result but without using a symbolic comparison. The actual code implements the logical not operator `!` at line 2 in the pseudocode using a sequence of bit-wise operations.

Code 6.4: Pseudocode of function IsMatch in Sym-NS-3

```
1 IsMatch (IP Address dst, Table Entry entry)
2   return !((dst&entry.mask)^(entry.ip&entry.mask));
```

It also replaces function `Lookup` in Pseudocode 6.3 with Pseudocode 6.5, which checks whether there is at least one matching entry in a priority group using only one symbolic comparison (i.e., line 6). Thus it generates at most one new branch for all the entries in a priority group.

Code 6.5: Pseudocode of function Lookup in Sym-NS-3

```
1 Lookup (IP Address dst, IP Table table)
2   for each priority group in the table
3     flag = false;
4     for each entry in the priority group
5       flag = flag | IsMatch(dst, entry);
6     if (flag)
7       return the group;
```

Our techniques are inspired by SymNet [31], which proposes a new symbolic execution friendly language to model computer networks including routing tables, whereas our techniques modify NS-3 code to be friendly to symbolic execution.

In the following two sections, we will prove that these two techniques are correct and efficient. Specifically, we will prove that IP-Efficient Sym-NS-3 is correct and more efficient with respect to Basic Sym-NS-3.

- Basic Sym-NS-3 refers to Pseudocode 6.3 and Pseudocode 6.2 with the symbolic variable management proposed in Chapter 5.

- IP-Efficient Sym-NS-3 refers to Pseudocode 6.5 and Pseudocode 6.4 (i.e., the techniques proposed in this chapter) with the symbolic variable management proposed in Chapter 5.

6.5 Is IP-Efficient Sym-NS-3 correct?

In this section, we prove that IP-Efficient Sym-NS-3 is correct. That is, we prove that Basic Sym-NS-3 (i.e., Pseudocode 6.3, Pseudocode 6.2) with an unsorted table (e.g., unsorted Table 6.1) generates the same simulation result as IP-Efficient Sym-NS-3 (i.e., Pseudocode 6.5, Pseudocode 6.4) with the corresponding sorted table (e.g., sorted Table 6.2). Specifically, we prove the correctness using two steps.

- Step 1: Basic Sym-NS-3 with an unsorted table generates the same simulation result as Basic Sym-NS-3 with the corresponding sorted table.
- Step 2: IP-Efficient Sym-NS-3 with a sorted table generates the same simulation result as Basic Sym-NS-3 with the same sorted table.

6.5.1 Step 1: Basic Sym-NS-3 with an unsorted table generates the same simulation result as Basic Sym-NS-3 with the corresponding sorted table

We prove that Basic Sym-NS-3 with an unsorted table (e.g., unsorted Table 6.1) generates the same simulation result as Basic Sym-NS-3 with the corresponding sorted table (e.g., sorted Table 6.2) using the following theorem. For example, if `dst`=10.1.0.1, Basic Sym-NS-3 with both unsorted Table 6.1 and sorted Table 6.2 returns the same best matching entry 10.1.0.0 with interface 2.

Theorem 1 *Basic Sym-NS-3 (i.e., Pseudocode 6.3 and 6.2) with both an unsorted table and the corresponding sorted table finds the same best matching entry, assuming that both tables have a unique best matching entry.*

Proof: The `for` loop between line 2 to line 8 in Pseudocode 6.3 checks each entry of a table (unsorted or sorted). At the beginning of the loop, the `IsMatch` function at line 3 checks whether the `dst` matches the current `entry`. If it does not match, the `for` loop then checks the next`entry`. If the `dst` matches the current `entry`, the `for` loop checks whether the current `entry` is the new best matching entry. Recall that the best matching entry is the entry with the longest mask among all matching entries. If there are multiple matching entries with the same longest length of masks, the entry with the shortest metric is the best matching entry. Specifically, line 4 checks whether the mask length of the current `entry` is longer than the current best matching entry `bestmatch`. Lines 6 and 7 check whether `entry` has the same mask length as `bestmatch` but with a shorter metric.

Because there is no `break`, `continues` or `return` in the `for` loop between line 2 to line 8 in Pseudocode 6.3, all entries in the routing table are checked. Therefore, the returned `bestmatch` is the best matching entry among all entries of the table. Under the assumption that the table has a unique best matching entry, `bestmatch` is the same independent of the order of the table entries. As a result, Pseudocode 6.3 always returns the same best matching entry independent of the order of the table entries. ■

To have a better understanding of the above theorem, let's consider an example using Pseudocode 6.3 with unsorted Table 6.1 and sorted Table 6.2 for `dst`=10.1.0.1.

- For unsorted Table 6.1, the second entry (i.e., the one with destination 0.0.0.0) is the first matching entry for `dst`=10.1.0.1, and then `bestmatch` is set to the second entry. Next, the third entry (i.e., the one with destination 10.1.0.0) is the next matching entry.

Compared with the second entry, the third entry has a longer mask length. Therefore, `bestmatch` is changed to the third entry. Since the last entry does not match the destination, the return value of Pseudocode 6.3 is the third entry with interface 2 and destination 10.1.0.0.

- For sorted Table 6.2, the first entry (i.e., the one with destination 10.1.0.0 is the first matching entry for `dst`=10.1.0.1, and then `bestmatch` is set to the first entry. Next, the last entry (i.e., the one with destination 0.0.0.0) is the next matching entry. Compared with the first entry, the last entry has a shorter mask length. Therefore, `bestmatch` is not changed. Finally, the return value of Pseudocode 6.3 is the first entry with interface 2 and destination 10.1.0.0.

We can see that Pseudocode 6.3 with unsorted Table 6.1 and sorted Table 6.2 returns the same best matching entry with interface 2 for `dst`=10.1.0.1.

6.5.2 Step 2: IP-Efficient Sym-NS-3 with a sorted table generates the same simulation result as Basic Sym-NS-3 with the same sorted table

We prove that IP-Efficient Sym-NS-3 with a sorted table (e.g., sorted Table 6.2) generates the same simulation result as Basic Sym-NS-3 with the same sorted table using the following theorem. For example, if `dst`=10.1.0.1, both IP-Efficient Sym-NS-3 and Basic Sym-NS-3 with sorted Table 6.2 return the same interface 2.

Theorem 2 *Both IP-Efficient Sym-NS-3 (i.e., Pseudocode 6.5 and 6.4) and Basic Sym-NS-3 (i.e., Pseudocode 6.3 and 6.2) with the same sorted table return the same interface.*

Proof: Theorem 1 proves that Basic Sym-NS-3 returns the best matching entry, and below we prove that IP-Efficient Sym-NS-3 returns the priority group containing the best matching entry.

As shown in Pseudocode 6.5, IP-Efficient Sym-NS-3 processes all the entries in the same priority group together, which are sorted and arranged when each entry is inserted to the routing table. Specifically, a Boolean variable `flag` is initialized at the beginning of the group loop (Line 3). The `flag` is used to indicate whether this group contains at least one matching entry (Lines 4 and 5). If `flag` becomes true, there is at least one matching entry in the current priority group. Note that, all priority groups are sorted in the table, and thus the first priority group with a true `flag` is the group containing the best matching entry and is returned by Pseudocode 6.5 (Lines 6 and 7).

Because all the entries in a priority group has the same interface, Pseudocode 6.5 (i.e., IP-Efficient Sym-NS-3) returns the interface of the priority group, which is the same as the interface of the best matching entry returned by Pseudocode 6.3 (i.e., current NS-3). ■

To have a better understanding of the above theorem, let's consider an example using IP-Efficient Sym-NS-3 with Pseudocode 6.5 and Basic Sym-NS-3 with Pseudocode 6.3 with sorted Table 6.2 for `dst`=10.1.0.1.

- For Basic Sym-NS-3, the return value of Pseudocode 6.3 is the first entry with interface 2 and destination 10.1.0.0.
- For IP-Efficient Sym-NS-3, the return value of Pseudocode 6.5 is the first priority group with interface 2. This is because the first entry of the group matches `dst` and thus `flag` of the group becomes true.

We can see that both Basic Sym-NS-3 and IP-Efficient Sym-NS-3 with sorted Table 6.2 returns the same interface 2 for `dst`=10.1.0.1.

6.6 Is IP-Efficient Sym-NS-3 efficient?

In this section, we prove that IP-Efficient Sym-NS-3 is more efficient that Basic Sym-NS-3. Note that although Sym-NS-3 needs additional time to sort a routing table, the time to call constraint solvers and fork virtual machines is significantly more time consuming than the time to sort a routing table. The time to call constraint solvers is the number of symbolic comparisons, and the time of virtual machines is the number of branches. Therefore, we prove the efficiency in two parts.

- Part 1: IP-Efficient Sym-NS-3 with a sorted table has less symbolic comparisons than Basic Sym-NS-3 with the corresponding unsorted table.
- Part 2: IP-Efficient Sym-NS-3 with a sorted table generates less branches than Basic Sym-NS-3 with the corresponding unsorted table.

6.6.1 Part 1: IP-Efficient Sym-NS-3 has less symbolic comparisons than Basic Sym-NS-3

We prove that IP-Efficient Sym-NS-3 with a sorted table (e.g., sorted Table 6.2) has less symbolic comparisons than Basic Sym-NS-3 with the unsorted table (e.g., unsorted Table 6.1) using the following two theorems. For example, if we search for a symbolic `dst` in the range of 10.1.0.0 to 10.2.255.255, Basic Sym-NS-3 with unsorted Table 6.1 has 4 symbolic comparisons, because the table has 4 entries. In contrast, IP-Efficient Sym-NS-3 with sorted Table 6.2 has only 1 symbolic comparison.

Theorem 3 *If an unsorted table has e entries, Basic Sym-NS-3 (i.e., Pseudocode 6.3 and 6.2) has e symbolic comparisons for a symbolic* `dst`.

Proof: Function `Lookup` defined in Pseudocode 6.3 does not have any `break`, `continues` or `return` in the `for` loop between line 2 to line 8. Therefore, all e entries in the routing table are checked by function `IsMatch` defined in Pseudocode 6.2. Because `dst` is symbolic, each time function `IsMatch` is called, there is a symbolic comparison. Therefore, there is a total of e symbolic comparisons. ■

Theorem 4 *If a sorted table has g priority groups (i.e., 1, 2, ..., g) and the first matching group for a symbolic* `dst` *is m, IP-Efficient Sym-NS-3 (i.e., Pseudocode 6.5, Pseudocode 6.4) has m symbolic comparisons for* `dst`*. We have $m \leq g \leq e$.*

Proof: First, we can see that IP-Efficient Sym-NS-3 has only one symbolic comparison for each priority group. This is because function `IsMatch` defined in Pseudocode 6.4 does not have any comparison for each entry of a priority group, and function `Lookup` defined in Pseudocode 6.5 has only one symbolic comparison at line 6 of the `for` loop for each priority group. Second, we can see that function `Lookup` defined in Pseudocode 6.5 stops as soon as it find the first matching priority group at lines 6 and 7.

Therefore, if the first matching group for `dst` is m, IP-Efficient Sym-NS-3 has m symbolic comparisons.

Because m is a number between 1 and g, and because the upper bound of g is e (i.e., when each priority group has only one entry), we have $m \leq g \leq e$. ■

To have a better understanding of the above theorems, let's consider the example using Basic Sym-NS-3 with unsorted Table 6.1 and IP-Efficient Sym-NS-3 with sorted Table 6.2 for the symbolic `dst` in the range of 10.1.0.0 to 10.2.255.255.

- Basic Sym-NS-3 checks all the 4 entries in the routing table. Thus, there are a total of 4 symbolic comparisons.

- IP-Efficient Sym-NS-3 has only 1 symbolic comparison. Because Pseudocode 6.5 calls line 6 only once for the two entries in priority group 1 and then returns priority group 1 (i.e., interface 2). Also note that the remaining priority groups (i.e., last two entries) of Table 6.2 are not checked. Therefore, there is only 1 symbolic comparison.

6.6.2 Part 2: IP-Efficient Sym-NS-3 generates less branches than Basic Sym-NS-3

We prove that IP-Efficient Sym-NS-3 generates less branches than Basic Sym-NS-3 using the following theorem. For example, if we search for a symbolic `dst` in the range of 10.1.0.0 to 10.2.255.255, Basic Sym-NS-3 generates 2 branches whereas IP-Efficient Sym-NS-3 generates only 1 branch.

Theorem 5 *IP-Efficient Sym-NS-3 generates no more branches than Basic Sym-NS-3 for a symbolic* `dst`.

Proof: For each best matching entry in a table for a symbolic `dst`, Basic Sym-NS-3 generates a branch. Note that because a symbolic `dst` covers a range of IP addresses, there might be multiple best matching entries. Thus, the number of branches generated by Basic Sym-NS-3 is the number of best matching entries in a table.

The number of branches generated by IP-Efficient Sym-NS-3 is the number of corresponding priority groups containing these best matching entries. Because each priority group contains at least one entry, the number of corresponding priority groups is no more than the number of best matching entries. In the best case, the number of corresponding priority group is 1 if it contains all the best matching entries. In the worse case, the number of corresponding priority groups is equal to the number of best matching entries if each priority group contains only one entry.

Therefore, the number of branches generated by IP-Efficient Sym-NS-3 is no more than that of Basic Sym-NS-3. ■

To have a better understanding of the above theorem, let's consider an example using Basic Sym-NS-3 with unsorted Table 6.1 and IP-Efficient Sym-NS-3 with sorted Table 6.2 for the symbolic `dst` in the range of 10.1.0.0 to 10.2.255.255.

- Basic Sym-NS-3 finds two best matching entries, which are the third entry with interface 2 for IP 10.1.0.0 10.1.255.255 and fourth entry with interface 2 for IP 10.2.0.0 10.2.255.255. Therefore, it generates 2 branches.
- IP-Efficient Sym-NS-3 finds only the first matching priority group, which contains two entries with IP 10.1.0.0 10.1.255.255 and IP 10.2.0.0 10.2.255.255, both with interface 2. Therefore, it generates only 1 branch.

Chapter 7

Experiments

7.1 Simulation setup

We evaluate the following three methods.

1. Brute force using current NS-3, which is referred to as *Brute Force*.
2. Symbolic execution using Sym-NS-3 with only the symbolic variable management proposed in Chapter 5, which is referred to as *Basic SymEx*.
3. Symbolic execution using Sym-NS-3 with both the symbolic variable management proposed in Chapter 5 and the techniques proposed in Chapter 6 for more efficient IP simulations, which is referred to as *IP-Efficient SymEx*.

7.2 Exhaustive testing on TCP performance

In this group of simulations, we exhaustively test the TCP performance in a network shown in Fig. 7.1, which has different and independent delays in different directions between nodes 0 and 1. Delay d_0 from node 0 to node 1 is in the range of [1, 1000] ms, and delay

d_1 from node 1 to node 0 is also in the range of [1, 1000] ms. Node 0 starts to establish a TCP connection to node 1 at time 0 with an initial congestion window of 1 segment, and then sends a total of 2 data segments to node 1. The TCP performance is measured by the number of segments received by node 1 within 2000 ms.

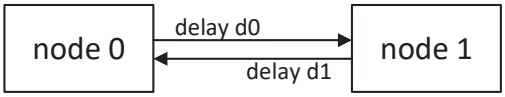

Figure 7.1: Network topology of the TCP performance testing.

Brute Force runs a total of $1000 \times 1000 = 10^6$ NS-3 simulations for all possible combinations of d_0 and d_1, and is estimated to take about 6 days. Basic SymEx uses two symbolic variables, one for d_0 and the other for d_1, and takes about 3 hours. Basic SymEx finally generates about 140 branches (i.e., equivalence classes of d_0 and d_1 values leading to the same simulator execution paths). To help us verify the correctness of the reported TCP performance, we also print out the ranges of $2d_0 + d1$ (i.e., the time for the three-way handshake) and $3d_0 + 2d1$ (i.e., one round-trip time after the three-way handshake) for each branch. The result of all the branches is summarized and aggregated in Table 7.1. Note that the link data rate is 5 Mbps, and the transmission time of a segment is slightly less than 1 ms. We can see that Basic SymEx exhaustively reports all possible TCP performance for all possible combinations of d_0 and d_1 in the specified ranges, and such information is very time consuming to obtain using Brute Force with NS-3.

Table 7.1: Exhaustive TCP performance testing by SymEx

$2d_0 + d_1$ (ms)	$3d_0 + 2d_1$ (ms)	Number of received segments
[**1999**, 3000]	[2999, 5000]	0
[1000, **1998**]	[**1999**, 3497]	1
[3, 1331]	[5,**1998**]	2

7.3 Exhaustive testing on reachability

In this group of simulations, we exhaustively test the reachability from node 0 to all other nodes in a network shown in Fig. 7.2. Specifically, node 0 sends a ping packet with a destination IP in the range of 10.0.0.0 and 10.255.255.255, and reports the round-trip time (RTT) if it receives a reply. The routing table of each node is automatically created by NS-3 function `PopulateRoutingTables`.

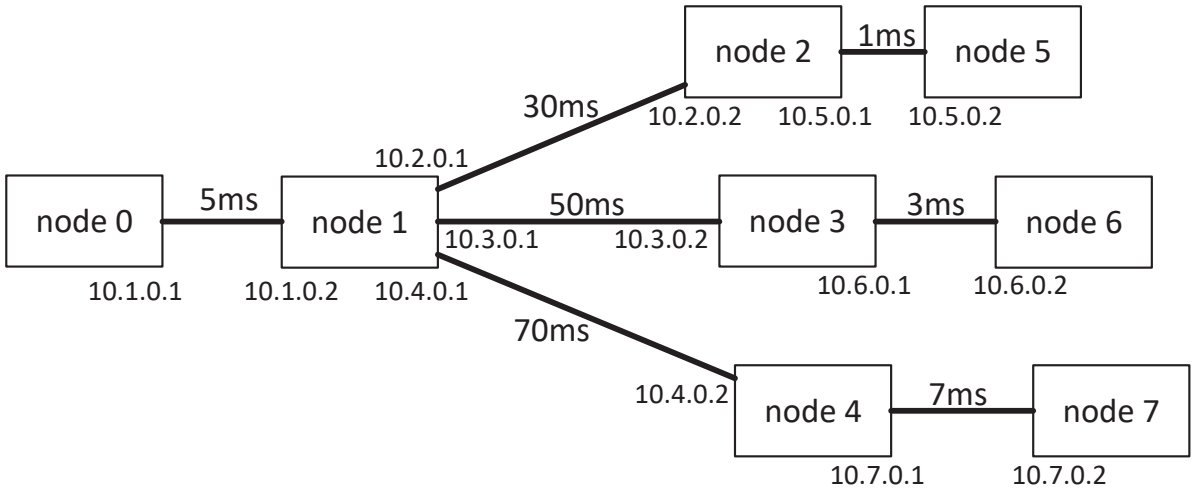

Figure 7.2: Network topology of the reachability testing.

Brute Force runs a total of 256^3 NS-3 simulations for each possible destination IP address, and is estimated to take about 100 days. Basic SymEx uses one symbolic variable for all the destination IP addresses, and takes about 15 minutes. Basic SymEx finally generates about 30 branches. The result of all branches is summarized in Table 7.2. We can see that Basic SymEx exhaustively reports all possible ping RTTs for all possible destination IP addresses in the specified range.

Table 7.2: Exhaustive reachability testing by SymEx

Destination IP	Ping RTT (ms)
10.1.0.1	0
10.1.0.2, 10.1.255.255, 10.2.0.1 10.2.255.255, 10.3.0.1, 10.3.255.255 10.4.0.1, 10.4.255.255	10
10.2.0.2, 10.5.0.1, 10.5.255.255	70
10.3.0.2, 10.6.0.1, 10.6.255.255	110
10.4.0.2, 10.7.0.1, 10.7.255.255	150
10.5.0.2	72
10.6.0.2	116
10.7.0.2	164
All other IP addresses	No reply

7.4 Evaluating IP-Efficient SymEx

The previous two groups of simulations have relatively small routing tables, so there is not much difference between Basic SymEx and IP-Efficient SymEx. In this group of simulations, we use a big routing table to demonstrate the different performance of Basic and IP-Efficient SymEx. Specifically, we manually add the following n additional entries to the routing table of node 2 in Fig. 7.2 for the reachability simulations.

The simulation results shown in Fig. 7.3 indicate that the number of branches generated by Basic SymEx increases proportionally as the number n of additional table entries increases, whereas that of IP-Efficient SymEx remains unchanged. Accordingly, the testing

Table 7.3: Additional table entries for node 2

Destination	Mask	Interface	Metric
10.5.1.0	255.255.255.0	2	default
...	...	...	...
10.5.*n*.0	255.255.255.0	2	default

time of Basic SymEx increases proportionally as n increases, whereas that of IP-Efficient SymEx increases only slightly.

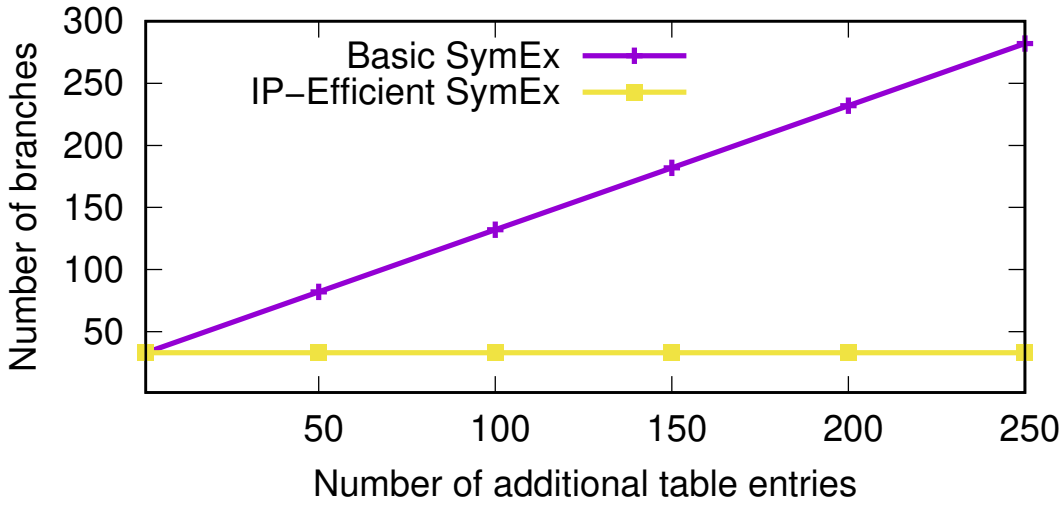

Figure 7.3: IP-Efficient SymEx is more efficient than Basic SymEx.

Chapter 8

Conclusions and Future work

8.1 Conclusions

In this thesis, we present our current progress on Sym-NS-3 for more efficient exhaustive testing. Specifically, we present our design choices and implementation on how we extend current NS-3 to support symbolic execution, and also the significantly improved testing speeds of Sym-NS-3 over current NS-3.

8.2 Shortcomings

One big problem with symbol execution is the path explosion. Therefore, Sym-NS-3 cannot deal with too many symbolic variables at the same time.

8.3 Future work

In the future, we plan to study and improve the performance of Sym-NS-3 for other types of symbolic variables, such as symbolic data rates of channels.

Bibliography

[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

[2] D. Babic, L. Martignoni, S. McCamant, and D. Song. Statically-directed dynamic automated test generation. In *Proceedings of ACM ISSTA*, Toronto, Canada, July 2011. 2

[3] F. Bellard. QEMU, a fast and portable dynamic translator. In *Proceedings of USENIX ATC*, Anaheim, CA, April 2005. 4.1

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

[5] S. Bugrara and D. Engler. Redundant state detection for dynamic symbolic execution. In *Proceedings of USENIX ATC*, San Jose, CA, June 2013. 2

[6] J. Burnim and K. Sen. Heuristics for scalable dynamic test generation. In *Proceedings of IEEE/ACM Conference on Automated Software Engineering*, L'Aquaila, Italy, September 2008. 2

[7] 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. 4.1

[8] C. Cadar and K. Sen. Symbolic execution for software testing: three decades later. *Communications of the ACM*, 56(2):82–90, February 2013. 1

[9] M. Canini, V. Jovanovic, D. Venzano, B. Spasojevic, and O. Crameri. Toward online testing of federated and heterogeneous distributed systems. In *Proceedings of USENIX ATC*, Portland, OR, June 2011. 2

[10] 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. 2

[11] V. Chipounov, V. Kuznetsov, and G. Candea. The S2E platform: design, implementation, and applications. *ACM Transactions on Computer Systems*, 30(1), February 2012. 2, 4.1

[12] C. Cho, D. Babic, P. Poosankam, K. Chen, E. Wu, and D. Song. MACE: model-inference-assisted concolic exploration for protocol and vulnerability discovery. In *Proceedings of USENIX Conference on Security (SEC)*, San Francisco, CA, August 2011. 2

[13] M. Dobrescu and K. Argyraki. Software dataplane verification. In *Proceedings of USENIX NSDI*, Seattle, WA, April 2014. 2

[14] O. Dustmann. Symbolic execution of discrete event systems with uncertain time. *Lecture Notes in Informatics*, S-12:19–22, 2013. 2

[15] S. Fayaz, T. Yu, Y. Tobioka, S. Chaki, and V. Sekar. BUZZ: Testing context-dependent policies in stateful networks. In *Proceedings of USENIX NSDI*, Santa Clara, CA, March 2016. 2

[16] P. Godefroid. Compositional dynamic test generation. In *Proceedings of POPL*, Nice, France, January 2007. 2

[17] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of ACM Programming Language Design and Implementation*, Chicagi, IL, June 2005. 2

[18] E. Hoque, O. Chowdhury, S. Chau, C. Nita-Rotaru, and N. Li. Analyzing operational behavior of stateful protocol implementations for detecting semantic bugs. In *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2017. 2

[19] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *Proceedings of USENIX NSDI*, San Jose, CA, April 2012. 6.1

[20] J. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976. 1

[21] N. Kothari, R. Mahajan, T. Millstein, R. Govindan, and M. Musuvathi. Finding protocol manipulation attacks. In *Proceedings of ACM SIGCOMM*, Toronto, Canada, August 2011. 2

[22] V. Kuznetsov, J. Kinder, S. Bucur, and G. Candea. Efficient state merging in symbolic execution. In *Proceedings of ACM Programming Language Design and Implementation*, Beijing, China, June 2012. 2

[23] M. Kuzniar, P. Peresini, M. Canini, D. Venzano, and D. Kostic. A SOFT way for OpenFlow switch interoperability testing. In *Proceedings of ACM Conference on emerging Networking EXperiments and Technologies (CoNEXT)*, Nice, France, December 2012. 2

[24] L. Pedrosa, A. Fogel, N. Kothari, R. Govindan, R. Mahajan, and T. Millstein. Analyzing protocol implementations for interoperability. In *Proceedings of USENIX NSDI*, Oakland, CA, May 2015. 2

[25] 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. 2

[26] R. Sasnauskas, P. Kaiser, R. Jukic, and K. Wehrle. Integration testing of protocol implementations using symbolic distributed execution. In *Proceedings of IEEE ICNP*, Austin, TX, October 2012. 2

[27] 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. 2

[28] 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. 2

[29] J. Song, C. Cadar, and P. Pietzuch. SymbexNet: Testing network protocol implementations with symbolic execution and rule-based specifications. *IEEE Transactions on Software Engineering*, 40(7):695–709, July 2014. 2

[30] N. Stephens, J. Grosen, C. Salls, A. Dutcher, R. Wang, J. Corbette, Y. Shoshitaishvili, C. Kruegel, and G. Vigna. Driller: augmenting fuzzing through selective symbolic execution. In *Proceedings of NDSS*, San Diego, CA, Feburary 2016. 2

[31] R. Stoenescu, M. Popovici, L. Negreanu, and C. Raiciu. SymNet: Scalable symbolic execution for modern networks. In *Proceedings of ACM SIGCOMM*, Brazil, August 2016. 2, 6.1, 6.4

[32] 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. 2

[33] M. Vu, L. Xu, S. Elbaum, W. Sun, and K. Qiao. Efficient systematic testing of network protocols with temporal uncertain events. In *Proceedings of IEEE INFOCOM*, Paris, France, April 2019. 1, 6

Appendix A

Source Code of Basic Sym-NS-3 in Chapter 5

Code A.1: Sym-NS-3 SymboClass.h

```
/* -*- Mode:C++; c-file-style:"gnu"; indent-tabs-mode:nil; -*- */
#ifndef SYMBOLIC_H
#define SYMBOLIC_H
#include "ns3/object.h"
#include "ns3/nstime.h"
#include "ns3/ipv4-address.h"

namespace ns3 {

/* ... */
class Symbolic: public Object
{

public:

  static TypeId GetTypeId (void);

  Symbolic ();
  Symbolic (uintptr_t v);
```

```
20    virtual ~Symbolic (){};
21
22    void SetMax(uintptr_t v);
23    void SetMin(uintptr_t v);
24    void SetMinMax(uintptr_t min, uintptr_t max);
25    void SetMax(Time v);
26    void SetMin(Time v);
27    void GetSymbolic();
28    uintptr_t GetSymbolicUintValue();
29    Ipv4Address GetSymbolicIpv4Add();
30    Time GetSymbolicTime();
31    void PrintRange();
32    static void PrintRange(char const* name, uintptr_t v);
33    static void PrintRangeTime(char const* name, Time t);
34    void PrintRange(char const* name);
35    void PrintSignedRange();
36    void PrintSignedRange(char const* name);
37    static void Stop(char const* name);
38    static void Print(const char * s);
39    static void Print(uintptr_t v);
40
41    uintptr_t GetUpperBound();
42    uintptr_t GetLowerBound();
43
44    uintptr_t m_symbolic;
45    uintptr_t m_min;
46    uintptr_t m_max;
47
48    friend Symbolic operator - (const Symbolic & l, const Symbolic & r);
49    friend Symbolic operator - (const Time & l, const Symbolic & r);
50    friend Symbolic operator - (const Symbolic &  l, const Time &r);
51    friend Symbolic operator + (const Symbolic & l, const Symbolic & r);
52    friend Symbolic operator + (const Time & l, const Symbolic & r);
53    friend Symbolic operator + (const Symbolic &  l, const Time &r);
54
55  };
56
57  inline Symbolic operator - (const Symbolic & l, const Symbolic & r)
58  {
```

```
59    return Symbolic (l.m_symbolic - r.m_symbolic);
60  }
61
62  inline Symbolic operator - (const Time & l, const Symbolic & r)
63  {
64    return Symbolic (l.GetTimeStep() + r.m_symbolic);
65  }
66
67  inline Symbolic operator - (const Symbolic &  l, const Time &r)
68  {
69    return Symbolic (l.m_symbolic - r.GetTimeStep());
70  }
71
72  inline Symbolic operator + (const Symbolic & l, const Symbolic & r)
73  {
74    return Symbolic (l.m_symbolic + r.m_symbolic);
75  }
76
77  inline Symbolic operator + (const Time & l, const Symbolic & r)
78  {
79    return Symbolic (l.GetTimeStep() + r.m_symbolic);
80  }
81
82  inline Symbolic operator + (const Symbolic &  l, const Time &r)
83  {
84    return Symbolic (l.m_symbolic + r.GetTimeStep());
85  }
86
87  }
88
89  #endif /* SYMBOLIC_H */
```

Code A.2: Sym-NS-3 SymboClass.cpp

```
1  #include "symbolic.h"
2
3
4  namespace ns3 {
5
```

```
6  /* ... */
7  NS_LOG_COMPONENT_DEFINE ("Symbolic");
8
9  NS_OBJECT_ENSURE_REGISTERED (Symbolic);
10
11 TypeId
12 Symbolic::GetTypeId (void)
13 {
14   static TypeId tid = TypeId ("ns3::Symbolic")
15     .SetParent<Object> ()
16     .SetGroupName ("Symbolic")
17     .AddConstructor<Symbolic> ()
18     .AddAttribute ("Max", "The Symbolic Max Value",
19                    UintegerValue (0xffff),
20                    MakeUintegerAccessor (&Symbolic::m_max),
21                    MakeUintegerChecker<uintptr_t> ())
22     .AddAttribute ("Min", "The Symbolic Min Value",
23                    UintegerValue (0),
24                    MakeUintegerAccessor (&Symbolic::m_min),
25                    MakeUintegerChecker<uintptr_t> ())
26   ;
27   return tid;
28 }
29
30 Symbolic::Symbolic()
31   :
32    m_symbolic(0),
33    m_min(0),
34    m_max(0xffffffff)
35
36 {
37   NS_LOG_FUNCTION_NOARGS ();
38 }
39
40 Symbolic::Symbolic(uintptr_t v)
41   :
42    m_symbolic(v),
43    m_min(0),
44    m_max(0xffffffff)
```

```
45
46  {
47    NS_LOG_FUNCTION_NOARGS ();
48  }
49
50
51  void
52  Symbolic::SetMax(uintptr_t v)
53  {
54      m_max=v;
55  }
56
57  void
58  Symbolic::SetMin(uintptr_t v)
59  {
60      m_min=v;
61  }
62
63  void
64  Symbolic::SetMinMax(uintptr_t min, uintptr_t max)
65  {
66      SetMin(min);
67      SetMax(max);
68  }
69
70  void
71  Symbolic::SetMax(Time v)
72  {
73      m_max=v.GetTimeStep();
74  }
75
76  void
77  Symbolic::SetMin(Time v)
78  {
79      m_min=v.GetTimeStep();
80  }
81
82  void
83  Symbolic::GetSymbolic()
```

```
84  {
85    if(s2e_is_symbolic(&m_symbolic,sizeof(m_symbolic)) != 1)
86    {
87      s2e_make_symbolic(&m_symbolic,sizeof(m_symbolic),"symbolic_value");
88      if(m_symbolic<m_min)
89      {
90        s2e_kill_state(0,"Out of Range, Lower");
91      }
92      else if(m_symbolic>m_max)
93      {
94        s2e_kill_state(0,"Out of Range, Upper");
95      }
96    }
97  }
98
99  uintptr_t
100 Symbolic::GetSymbolicUintValue()
101 {
102     GetSymbolic();
103     return m_symbolic;
104 }
105
106 Ipv4Address
107 Symbolic::GetSymbolicIpv4Add()
108 {
109     GetSymbolic();
110     Ipv4Address returnAddress;
111     returnAddress.Set(m_symbolic);
112     return returnAddress;
113 }
114
115 Time
116 Symbolic::GetSymbolicTime()
117 {
118     GetSymbolic();
119     return Time(m_symbolic);
120 }
121
122 void
```

```
123 Symbolic::PrintRange()
124 {
125   uintptr_t upper;
126   uintptr_t lower;
127   s2e_get_range(m_symbolic,&lower,&upper);
128   s2e_printf("The range of Symbolic Variables is %ld,%ld",lower,upper);
129 }
130
131 void
132 Symbolic::PrintRange(char const* name, uintptr_t v)
133 {
134   if(s2e_is_symbolic(&v,sizeof(v)) != 1)
135   {
136     s2e_printf("%s is not a symbolic variable.",name);
137   }else{
138     uintptr_t upper;
139     uintptr_t lower;
140     s2e_get_range(v,&lower,&upper);
141     s2e_printf("The range of %s Variables is %ld,%ld",name,lower,upper);
142     s2e_printf("The range of %s Variables is %lu,%lu",name,lower,upper);
143   }
144 }
145
146 void
147 Symbolic::Print(const char *s){
148   s2e_printf("%s",s);
149 }
150
151 void
152 Symbolic::Print(uintptr_t v){
153   s2e_printf("UInt Value %lu",v);
154   s2e_printf("Int Value %ld",v);
155 }
156
157 void
158 Symbolic::PrintRangeTime(char const* name, Time t)
159 {
160   if(s2e_is_symbolic(&t,sizeof(t)) != 1)
161   {
```

```
162       s2e_printf("%s is not a symbolic Time.",name);
163     }else{
164       PrintRange(name,t.GetTimeStep());
165     }
166 }
167 
168 void
169 Symbolic::PrintRange(char const* name)
170 {
171   uintptr_t upper;
172   uintptr_t lower;
173   s2e_get_range(m_symbolic,&lower,&upper);
174   s2e_printf("The range of %s Variables is %ld,%ld",name,lower,upper);
175   s2e_printf("The range of %s Variables is %lu,%lu",name,lower,upper);
176 }
177 
178 void
179 Symbolic::Stop(char const* name)
180 {
181   s2e_kill_state_printf(0,name);
182 }
183 
184 uintptr_t
185 Symbolic::GetUpperBound()
186 {
187   uintptr_t upper;
188   uintptr_t lower;
189   s2e_get_range(m_symbolic,&lower,&upper);
190   return upper;
191 }
192 
193 uintptr_t
194 Symbolic::GetLowerBound()
195 {
196   uintptr_t upper;
197   uintptr_t lower;
198   s2e_get_range(m_symbolic,&lower,&upper);
199   return lower;
200 }
```

```
201
202 }
```

Appendix B

Source Code of IP-Efficient Sym-NS-3 in Chapter 6

Code B.1: Sym-NS-3 Ipv4StaticRouting::LookupStatic

```
1  Ptr<Ipv4Route>
2  Ipv4StaticRouting::LookupStatic (Ipv4Address dest, Ptr<NetDevice> oif)
3  {
4    NS_LOG_FUNCTION (this << dest << " " << oif);
5    Ptr<Ipv4Route> rtentry = 0;
6    /* when sending on local multicast, there have to be interface specified */
7    if (dest.IsLocalMulticast ())
8      {
9        NS_ASSERT_MSG (oif, "Try to send on link-local multicast address, and no interface index is
             given!");
10
11       rtentry = Create<Ipv4Route> ();
12       rtentry->SetDestination (dest);
13       rtentry->SetGateway (Ipv4Address::GetZero ());
14       rtentry->SetOutputDevice (oif);
15       rtentry->SetSource (m_ipv4->GetAddress (m_ipv4->GetInterfaceForDevice (oif), 0).GetLocal ()
             );
16       return rtentry;
17     }
```

```
18
19    uint16_t masklenRec = 32;
20    uint32_t interfaceRec = 0;
21    uint32_t metricRec = 0xffffffff;
22    uint8_t matchRst = 1; //must be non-zero
23    Ipv4RoutingTableEntry *jRec = NULL;
24    bool sendFlag = false;
25    for (NetworkRoutesI i = m_networkRoutes.begin ();
26         i != m_networkRoutes.end ();
27         i++)
28      {
29        Ipv4RoutingTableEntry *j=i->first;
30        uint32_t metric =i->second;
31        Ipv4Mask mask = (j)->GetDestNetworkMask ();
32        uint16_t masklen = mask.GetPrefixLength ();
33        Ipv4Address entry = (j)->GetDestNetwork ();
34        uint32_t interfaceIdx = (j)->GetInterface ();
35        // NOT SUPPORT send through the requested interface, ignored
36        NS_LOG_LOGIC ("Searching for route to " << dest << ", checking against route to " << entry
              << "/" << masklen << ", Metric is "<< metric << ", Interface is " << j->GetInterface ()
              );
37        if(interfaceRec != interfaceIdx){
38          NS_LOG_LOGIC("Changed Interface");
39          if(matchRst == 0){
40            sendFlag = true;
41            break;
42          }
43          else{
44            masklenRec = masklen;
45            metricRec = metric;
46            interfaceRec = interfaceIdx;
47            matchRst = mask.MatchCalc(dest, entry);
48          }
49        }else if (metricRec != metric){
50          NS_LOG_LOGIC("Changed metric");
51          if(matchRst == 0){
52            sendFlag = true;
53            break;
54          }
```

```
55          else{
56            masklenRec = masklen;
57            metricRec = metric;
58            interfaceRec = interfaceIdx;
59            matchRst = mask.MatchCalc(dest, entry);
60          }
61        }else if(masklenRec != masklen){
62          NS_LOG_LOGIC("Changed masklen");
63          if(matchRst == 0){
64            sendFlag = true;
65            break;
66          }
67          else{
68            masklenRec = masklen;
69            metricRec = metric;
70            interfaceRec = interfaceIdx;
71            matchRst = mask.MatchCalc(dest, entry);
72          }
73        }else{
74          NS_LOG_LOGIC("In the same group");
75          matchRst = mask.MatchCalc(dest, entry, matchRst);
76        }
77        jRec = j;
78      }
79    if(matchRst == 0)
80      sendFlag = true;
81    if(sendFlag){
82      rtentry = Create<Ipv4Route> ();
83      rtentry->SetDestination ((jRec)->GetDest ());
84      rtentry->SetSource (m_ipv4->SourceAddressSelection (interfaceRec, (jRec)->GetDest ()));
85      rtentry->SetGateway ((jRec)->GetGateway ());
86      rtentry->SetOutputDevice (m_ipv4->GetNetDevice (interfaceRec));
87    }
88    if (rtentry != 0)
89      {
90        NS_LOG_LOGIC ("Matching route via " << rtentry->GetGateway () << " at the end");
91      }
92    else
93      {
```

```
94        NS_LOG_LOGIC ("No matching route to " << dest << " found");
95      }
96    return rtentry;
97  }
```

Code B.2: Sym-NS-3 Ipv4StaticRouting::AddNetworkRouteTo

```
1   void
2   Ipv4StaticRouting::AddNetworkRouteTo (Ipv4Address network,
3                                         Ipv4Mask networkMask,
4                                         Ipv4Address nextHop,
5                                         uint32_t interface,
6                                         uint32_t metric)
7   {
8     NS_LOG_FUNCTION (this << network << " " << networkMask << " " << nextHop << " " << interface <<
          " " << metric);
9
10    Ipv4RoutingTableEntry route = Ipv4RoutingTableEntry::CreateNetworkRouteTo (network,
11                                                                               networkMask,
12                                                                               nextHop,
13                                                                               interface);
14
15    Ipv4RoutingTableEntry *routePtr = new Ipv4RoutingTableEntry (route);
16    uint16_t masklen = networkMask.GetPrefixLength ();
17
18    for (NetworkRoutesI i = m_networkRoutes.begin ();
19      i != m_networkRoutes.end ();
20      i++)
21    {
22      Ipv4RoutingTableEntry *j=i->first;
23      uint32_t metricTmp =i->second;
24      Ipv4Mask maskTmp = (j)->GetDestNetworkMask ();
25      uint16_t masklenTmp = maskTmp.GetPrefixLength ();
26      uint32_t interfaceIdxTmp = (j)->GetInterface ();
27
28      if(masklen > masklenTmp){
29        m_networkRoutes.insert (i,make_pair (routePtr, metric));
30        return;
31      }else if(masklen < masklenTmp){
```

```
32        continue;
33      }else{
34        if(metric < metricTmp){
35          m_networkRoutes.insert (i,make_pair (routePtr, metric));
36          return;
37        }else if(metric > metricTmp){
38          continue;
39        }else{
40          if(interface <= interfaceIdxTmp){
41            m_networkRoutes.insert (i,make_pair (routePtr, metric));
42            return;
43          }else{
44            continue;
45          }
46        }
47      }
48    }
49    m_networkRoutes.push_back (make_pair (routePtr, metric));
50  }
51
52  void
53  Ipv4StaticRouting::AddNetworkRouteTo (Ipv4Address network,
54                                        Ipv4Mask networkMask,
55                                        uint32_t interface,
56                                        uint32_t metric)
57  {
58    NS_LOG_FUNCTION (this << network << " " << networkMask << " " << interface << " " << metric);
59
60    Ipv4RoutingTableEntry route = Ipv4RoutingTableEntry::CreateNetworkRouteTo (network,
61                                                                               networkMask,
62                                                                               interface);
63
64    Ipv4RoutingTableEntry *routePtr = new Ipv4RoutingTableEntry (route);
65    uint16_t masklen = networkMask.GetPrefixLength ();
66
67    for (NetworkRoutesI i = m_networkRoutes.begin ();
68      i != m_networkRoutes.end ();
69      i++)
70    {
```

```
71      Ipv4RoutingTableEntry *j=i->first;
72      uint32_t metricTmp =i->second;
73      Ipv4Mask maskTmp = (j)->GetDestNetworkMask ();
74      uint16_t masklenTmp = maskTmp.GetPrefixLength ();
75      uint32_t interfaceIdxTmp = (j)->GetInterface ();
76
77      if(masklen > masklenTmp){
78        m_networkRoutes.insert (i,make_pair (routePtr, metric));
79        return;
80      }else if(masklen < masklenTmp){
81        continue;
82      }else{
83        if(metric < metricTmp){
84          m_networkRoutes.insert (i,make_pair (routePtr, metric));
85          return;
86        }else if(metric > metricTmp){
87          continue;
88        }else{
89          if(interface <= interfaceIdxTmp){
90            m_networkRoutes.insert (i,make_pair (routePtr, metric));
91            return;
92          }else{
93            continue;
94          }
95        }
96      }
97    }
98    m_networkRoutes.push_back (make_pair (routePtr, metric));
99  }
```

Code B.3: Sym-NS-3 Ipv4Mask::MatchCalc

```
1  uint8_t
2  Ipv4Mask::MatchCalc (Ipv4Address a, Ipv4Address b, uint8_t record) const
3  {
4    NS_LOG_FUNCTION (this << a << b << record);
5    uint32_t temp = ((a.Get () & m_mask) ^ (b.Get () & m_mask));
6    uint8_t tempRec = 0;
7    for(int i = 0; i < 32; i++){
```

```
8       uint8_t index = temp & 1;
9       temp = temp >> 1;
10      tempRec = tempRec | index;
11    }
12    record = record & tempRec;
13    return record;
14  }
```

Appendix C

Source Code of Examples and Experiments

C.1 Brute force using current NS-3 for the motivating example in Chapter 3

Code C.1: repeatCurrentDemo.sh

```
#!/bin/bash
for delay0 in {1..1000}
do
  for delay1 in {1..1000}
  do
    ./waf --run "currentDemo --d0=$delay0 --d1=$delay1"
  done
done
```

Code C.2: currentDemo.cc

```
#include <iostream>
#include <fstream>
#include <string>
```

```
4  #include <cassert>
5
6  #include "ns3/core-module.h"
7  #include "ns3/network-module.h"
8  #include "ns3/internet-module.h"
9  #include "ns3/point-to-point-module.h"
10 #include "ns3/applications-module.h"
11
12 //Network Topology
13 //
14 // snda -------rcv------- sndb
15 //     point-to-point
16 //
17
18 using namespace ns3;
19
20 NS_LOG_COMPONENT_DEFINE ("CurrentNS3ScriptExample");
21
22 int
23 main (int argc, char *argv[])
24 {
25   Time::SetResolution (Time::MS);
26
27   uint32_t d0 = 0;
28   uint32_t d1 = 0;
29   CommandLine cmd (__FILE__);
30   cmd.AddValue ("d0", "The delay for link between snd1 and rcv.", d0);
31   cmd.AddValue ("d1", "The delay for link between snd2 and rcv.", d1);
32   cmd.Parse (argc, argv);
33
34
35   std::vector<PointToPointHelper> p2p (2);
36   p2p[0].SetChannelAttribute ("Delay", TimeValue (Time (d0)));
37   p2p[1].SetChannelAttribute ("Delay", TimeValue (Time (d1)));
38
39   NodeContainer nodes;
40   nodes.Create (3);
41
42   std::vector<NodeContainer> nodeAdjacencyList (2);
```

```
43  nodeAdjacencyList[0] = NodeContainer (nodes.Get (0), nodes.Get (2));
44  nodeAdjacencyList[1] = NodeContainer (nodes.Get (1), nodes.Get (2));
45
46  std::vector<NetDeviceContainer> devices (2);
47  devices[0] = p2p[0].Install (nodeAdjacencyList[0]);
48  devices[1] = p2p[1].Install (nodeAdjacencyList[1]);
49
50  InternetStackHelper stack;
51  stack.Install (nodes);
52
53  Ipv4AddressHelper address;
54  std::vector<Ipv4InterfaceContainer> interfaces (2);
55  for (uint32_t i = 0; i < 2; i++)
56    {
57      std::ostringstream subset;
58      subset << "10.1." << i + 1 << ".0";
59      address.SetBase (subset.str ().c_str (), "255.255.255.0");
60      interfaces[i] = address.Assign (devices[i]);
61    }
62
63  UdpServerHelper server (2333);
64
65  ApplicationContainer rcv = server.Install (nodes.Get (2));
66  rcv.Start (Seconds (1.0));
67  rcv.Stop (Seconds (10.0));
68
69  UdpClientHelper snd1 (interfaces[0].GetAddress (1), 2333);
70  snd1.SetAttribute ("MaxPackets", UintegerValue (1));
71
72  UdpClientHelper snd2 (interfaces[1].GetAddress (1), 2333);
73  snd2.SetAttribute ("MaxPackets", UintegerValue (1));
74
75  ApplicationContainer snd;
76  snd.Add(snd1.Install (nodes.Get (0)));
77  snd.Add(snd2.Install (nodes.Get (1)));
78  snd.Start (Seconds (1.0));
79  snd.Stop (Seconds (10.0));
80
81  Simulator::Run ();
```

```
82    Time Diff = Time(d0)-Time(d1);
83    std::cout<<"Diff is "<<Diff<<std::endl;
84    Simulator::Destroy ();
85    return 0;
86  }
```

C.2 Symbolic execution using Sym-NS-3 for the motivating example in Chapter 3

Code C.3: symDemo.cc

```
1   #include <iostream>
2   #include <fstream>
3   #include <string>
4   #include <cassert>
5
6   #include "ns3/core-module.h"
7   #include "ns3/network-module.h"
8   #include "ns3/internet-module.h"
9   #include "ns3/point-to-point-module.h"
10  #include "ns3/applications-module.h"
11  #include "ns3/symbolic-module.h"
12
13  //Network Topology
14  //
15  // snda -------rcv------- sndb
16  //     point-to-point
17  //
18
19  using namespace ns3;
20
21  NS_LOG_COMPONENT_DEFINE ("SymNS3ScriptExample");
22
23  int
24  main (int argc, char *argv[])
25  {
```

```
26  Time::SetResolution (Time::MS);
27
28  Ptr<Symbolic> sym0 = CreateObject<Symbolic>();
29  sym0->SetMinMax(1, 1000);
30  uint32_t d0 = sym0->GetSymbolicUintValue();
31  Ptr<Symbolic> sym1 = CreateObject<Symbolic>();
32  sym1->SetMinMax(1, 1000);
33  uint32_t d1 = sym1->GetSymbolicUintValue();
34
35  std::vector<PointToPointHelper> p2p (2);
36  p2p[0].SetChannelAttribute("Delay",TimeValue(Time(d0)));
37  p2p[1].SetChannelAttribute("Delay",TimeValue(Time(d1)));
38
39  NodeContainer nodes;
40  nodes.Create (3);
41
42  std::vector<NodeContainer> nodeAdjacencyList (2);
43  nodeAdjacencyList[0] = NodeContainer (nodes.Get (0), nodes.Get (2));
44  nodeAdjacencyList[1] = NodeContainer (nodes.Get (1), nodes.Get (2));
45
46  std::vector<NetDeviceContainer> devices (2);
47  devices[0] = p2p[0].Install (nodeAdjacencyList[0]);
48  devices[1] = p2p[1].Install (nodeAdjacencyList[1]);
49
50  InternetStackHelper stack;
51  stack.Install (nodes);
52
53  Ipv4AddressHelper address;
54  std::vector<Ipv4InterfaceContainer> interfaces (2);
55  for (uint32_t i = 0; i < 2; i++)
56    {
57      std::ostringstream subset;
58      subset << "10.1." << i + 1 << ".0";
59      address.SetBase (subset.str().c_str (), "255.255.255.0");
60      interfaces[i] =
61          address.Assign (devices[i]);
62    }
63
64  UdpServerHelper server (2333);
```

```
65
66    ApplicationContainer rcv = server.Install (nodes.Get (2));
67    rcv.Start (Seconds (1.0));
68    rcv.Stop (Seconds (10.0));
69
70    UdpClientHelper snd1 (interfaces[0].GetAddress (1), 2333);
71    snd1.SetAttribute ("MaxPackets", UintegerValue (1));
72
73    UdpClientHelper snd2 (interfaces[1].GetAddress (1), 2333);
74    snd2.SetAttribute ("MaxPackets", UintegerValue (1));
75
76    ApplicationContainer snd;
77    snd.Add(snd1.Install (nodes.Get (0)));
78    snd.Add(snd2.Install (nodes.Get (1)));
79    snd.Start (Seconds (1.0));
80    snd.Stop (Seconds (10.0));
81
82    Simulator::Run ();
83    int diff = d0 - d1;
84    Symbolic::PrintRange("Diff",diff);
85    Simulator::Destroy ();
86    Symbolic::Stop("Program terminated");
87    return 0;
88  }
```

C.3 Exhaustive testing on TCP performance in Chapter 7.2

Code C.4: tcp-demo.sh

```
1  // Network topology
2  //
3  //       n0 ----------- n1
4  //
5  // - Flow from n0 to n1 using BulkSendApplication.
6  // - Tracing of queues and packet receptions to file "tcp-bulk-send.tr"
7  //   and pcap tracing available when tracing is turned on.
8
```

```
9  #include <string>
10 #include <fstream>
11 #include "ns3/core-module.h"
12 #include "ns3/point-to-point-module.h"
13 #include "ns3/internet-module.h"
14 #include "ns3/applications-module.h"
15 #include "ns3/network-module.h"
16 #include "ns3/packet-sink.h"
17
18 using namespace ns3;
19
20 NS_LOG_COMPONENT_DEFINE ("TcpBulkSendExample");
21
22 int
23 main (int argc, char *argv[])
24 {
25   uint32_t maxBytes = 1000;
26   Config::SetDefault("ns3::TcpSocket::InitialCwnd",UintegerValue (1));
27
28 //
29 // Allow the user to override any of the defaults at
30 // run-time, via command-line arguments
31 //
32   CommandLine cmd (__FILE__);
33   cmd.Parse (argc, argv);
34
35 //
36 // Explicitly create the nodes required by the topology (shown above).
37 //
38
39   Time::SetResolution (Time::MS);
40
41   Ptr<Symbolic> symObj0 = CreateObject<Symbolic>();
42   symObj0->SetMinMax(1, 1000);
43   Time d0 = symObj0->GetSymbolicTime();
44   Ptr<Symbolic> symObj1 = CreateObject<Symbolic>();
45   symObj0->SetMinMax(1, 1000);
46   Time d1 = symObj1->GetSymbolicTime();
47
```

```
48 NodeContainer nodes;
49   nodes.Create (2);
50
51   uint32_t links = 2;
52   std::vector<NodeContainer> nodeAdjacencyList (links);
53   nodeAdjacencyList[0] = NodeContainer (nodes.Get (0), nodes.Get (1));
54   nodeAdjacencyList[1] = NodeContainer (nodes.Get (0), nodes.Get (1));
55   std::vector<PointToPointHelper> p2p (links);
56
57   std::vector<NetDeviceContainer> devices (links);
58
59   p2p[0].SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
60   p2p[1].SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
61   p2p[0].SetChannelAttribute ("Delay", TimeValue (d0));
62   p2p[1].SetChannelAttribute ("Delay", TimeValue (d1));
63
64
65   for(uint32_t i = 0; i < links; i++){
66     devices[i] = p2p[i].Install (nodeAdjacencyList[i]);
67   }
68
69   InternetStackHelper stack;
70   stack.Install (nodes);
71
72   Ipv4AddressHelper address;
73   std::vector<Ipv4InterfaceContainer> interfaces (links);
74   for (uint32_t i = 0; i < links; i++)
75     {
76       std::ostringstream subset;
77       subset << "10." << i + 1 << ".0.0";
78       address.SetBase (subset.str().c_str (), "255.255.0.0");
79       interfaces[i] =
80           address.Assign (devices[i]);
81     }
82
83
84   Ipv4StaticRoutingHelper ipv4RoutingHelper;
85   std::vector<Ptr<Ipv4StaticRouting>> remoteHostStaticRoutingNode (2);
86   for(uint32_t i = 0; i < 2; i++){
```

```
87      remoteHostStaticRoutingNode[i] = ipv4RoutingHelper.GetStaticRouting(nodes.Get (i)->GetObject<
            Ipv4>());
88    }
89    remoteHostStaticRoutingNode[0]->RemoveRoute(2);
90    remoteHostStaticRoutingNode[0]->RemoveRoute(1);
91    remoteHostStaticRoutingNode[1]->RemoveRoute(2);
92    remoteHostStaticRoutingNode[1]->RemoveRoute(1);
93    remoteHostStaticRoutingNode[0]->AddNetworkRouteTo(Ipv4Address("0.0.0.0"),Ipv4Mask("0.0.0.0"),
          Ipv4Address("10.1.0.2"),1);
94    remoteHostStaticRoutingNode[1]->AddNetworkRouteTo(Ipv4Address("0.0.0.0"),Ipv4Mask("0.0.0.0"),
          Ipv4Address("10.2.0.1"),2);
95  //
96  // Create a BulkSendApplication and install it on node 0
97  //
98    uint16_t port = 9;  // well-known echo port number
99
100
101   BulkSendHelper source ("ns3::TcpSocketFactory",
102                          InetSocketAddress (interfaces[0].GetAddress (1), port));
103   // Set the amount of data to send in bytes.  Zero is unlimited.
104   source.SetAttribute ("MaxBytes", UintegerValue (maxBytes));
105   ApplicationContainer sourceApps = source.Install (nodes.Get (0));
106   sourceApps.Start (Seconds (0.0));
107   sourceApps.Stop (Seconds (10.0));
108
109 //
110 // Create a PacketSinkApplication and install it on node 1
111 //
112   PacketSinkHelper sink ("ns3::TcpSocketFactory",
113                          InetSocketAddress (Ipv4Address::GetAny (), port));
114   ApplicationContainer sinkApps = sink.Install (nodes.Get (1));
115   sinkApps.Start (Seconds (0.0));
116   sinkApps.Stop (Seconds (10.0));
117
118 //
119 // Now, do the actual simulation.
120 //
121   NS_LOG_INFO ("Run Simulation.");
122   Simulator::Stop (Seconds (2));
```

```
123   Simulator::Run ();
124   Simulator::Destroy ();
125   NS_LOG_INFO ("Done.");
126   Time timeAdd0 = 2 * d0 + d1;
127   Time timeAdd1 = 3 * d0 + 2 * d1;
128   Symbolic::PrintRangeTime("2 d0 + 1 d1", timeAdd0);
129   Symbolic::PrintRangeTime("3 d0 + 2 d1", timeAdd1);
130   Ptr<PacketSink> sink1 = DynamicCast<PacketSink> (sinkApps.Get (0));
131   Symbolic::Print("Total Bytes Received:");
132   Symbolic::Print(sink1->GetTotalRx ());
133   Symbolic::Stop("Program terminated");
134 }
```

C.4 Exhaustive testing on reachability in Chapter 7.3

Code C.5: reachabilitySymEx.cc

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  #include <cassert>
5
6  #include "ns3/core-module.h"
7  #include "ns3/network-module.h"
8  #include "ns3/internet-module.h"
9  #include "ns3/point-to-point-module.h"
10 #include "ns3/applications-module.h"
11 #include "ns3/internet-apps-module.h"
12 #include "ctime"
13
14
15
16 using namespace ns3;
17
18 NS_LOG_COMPONENT_DEFINE ("ReachabilityTesting");
19
20 int
```

```
21 main (int argc, char *argv[])
22 {
23
24   Time::SetResolution (Time::MS);
25
26
27   CommandLine cmd;
28   cmd.Parse (argc, argv);
29
30   NodeContainer nodes;
31   nodes.Create (8);
32
33   uint32_t links = 7;
34   std::vector<NodeContainer> nodeAdjacencyList (links);
35   nodeAdjacencyList[0] = NodeContainer (nodes.Get (0), nodes.Get (1));
36   nodeAdjacencyList[1] = NodeContainer (nodes.Get (1), nodes.Get (2));
37   nodeAdjacencyList[2] = NodeContainer (nodes.Get (1), nodes.Get (3));
38   nodeAdjacencyList[3] = NodeContainer (nodes.Get (1), nodes.Get (4));
39   nodeAdjacencyList[4] = NodeContainer (nodes.Get (2), nodes.Get (5));
40   nodeAdjacencyList[5] = NodeContainer (nodes.Get (3), nodes.Get (6));
41   nodeAdjacencyList[6] = NodeContainer (nodes.Get (4), nodes.Get (7));
42   std::vector<PointToPointHelper> pointToPoint (links);
43
44   std::vector<NetDeviceContainer> devices (links);
45
46   pointToPoint[0].SetChannelAttribute ("Delay", StringValue ("5ms"));
47   pointToPoint[1].SetChannelAttribute ("Delay", StringValue ("30ms"));
48   pointToPoint[2].SetChannelAttribute ("Delay", StringValue ("50ms"));
49   pointToPoint[3].SetChannelAttribute ("Delay", StringValue ("70ms"));
50   pointToPoint[4].SetChannelAttribute ("Delay", StringValue ("1ms"));
51   pointToPoint[5].SetChannelAttribute ("Delay", StringValue ("3ms"));
52   pointToPoint[6].SetChannelAttribute ("Delay", StringValue ("7ms"));
53
54   for(uint32_t i = 0; i < links; i++){
55     pointToPoint[i].SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
56     devices[i] = pointToPoint[i].Install (nodeAdjacencyList[i]);
57   }
58
59   InternetStackHelper stack;
```

```
60    stack.Install (nodes);
61
62    Ipv4AddressHelper address;
63    std::vector<Ipv4InterfaceContainer> interfaces (links);
64    for (uint32_t i = 0; i < links; i++)
65      {
66        std::ostringstream subset;
67        subset << "10." << i + 1 << ".0.0";
68        address.SetBase (subset.str().c_str (), "255.255.0.0");
69        interfaces[i] =
70            address.Assign (devices[i]);
71      }
72
73    // We use the following code, it will auto create the routing table to connect to each other.
74    Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
75
76    //Symbol Address
77    Ptr<Symbolic> symObj0 = CreateObject<Symbolic>();
78    symObj0->SetMinMax(0xa000000, 0xaffffff);
79    Ipv4Address symIP0 = symObj0->GetSymbolicIpv4Add();
80
81    V4PingHelper ping(symIP0);
82    ApplicationContainer pingApp = ping.Install (nodes.Get(0));
83    pingApp.Start (Seconds (1.0));
84    pingApp.Stop (Seconds (1.7));
85
86
87    Simulator::Stop(Seconds (1.7));
88
89    Simulator::Run ();
90    Simulator::Destroy ();
91    symObj0->PrintRange("symIP0");
92    symObj0->Stop("Program terminated");
93    return 0;
94  }
```

C.5 Evaluating IP-Efficient SymEx in Chapter 7.4

Code C.6: IPEfficientSymEx.cc

```
1  #include <iostream>
2  #include <fstream>
3  #include <string>
4  #include <cassert>
5
6  #include "ns3/core-module.h"
7  #include "ns3/network-module.h"
8  #include "ns3/internet-module.h"
9  #include "ns3/point-to-point-module.h"
10 #include "ns3/applications-module.h"
11 #include "ns3/internet-apps-module.h"
12 #include "ctime"
13
14 //Network Topology
15 //
16 // snda -------rcv------- sndb
17 //     point-to-point
18 //
19
20 using namespace ns3;
21
22 NS_LOG_COMPONENT_DEFINE ("IP-EfficientSymEx");
23
24 int
25 main (int argc, char *argv[])
26 {
27
28   Time::SetResolution (Time::MS);
29   Config::SetDefault("ns3::Ipv4L3Protocol::DefaultTtl",UintegerValue (3));
30
31   CommandLine cmd;
32   cmd.Parse (argc, argv);
33
34   uint32_t nodesNum = 8;
35   NodeContainer nodes;
```

```
36  nodes.Create (nodesNum);
37
38  uint32_t links = 7;
39  std::vector<NodeContainer> nodeAdjacencyList (links);
40  nodeAdjacencyList[0] = NodeContainer (nodes.Get (0), nodes.Get (1));
41  nodeAdjacencyList[1] = NodeContainer (nodes.Get (1), nodes.Get (2));
42  nodeAdjacencyList[2] = NodeContainer (nodes.Get (1), nodes.Get (3));
43  nodeAdjacencyList[3] = NodeContainer (nodes.Get (1), nodes.Get (4));
44  nodeAdjacencyList[4] = NodeContainer (nodes.Get (2), nodes.Get (5));
45  nodeAdjacencyList[5] = NodeContainer (nodes.Get (3), nodes.Get (6));
46  nodeAdjacencyList[6] = NodeContainer (nodes.Get (4), nodes.Get (7));
47  std::vector<PointToPointHelper> pointToPoint (links);
48
49  std::vector<NetDeviceContainer> devices (links);
50
51  pointToPoint[0].SetChannelAttribute ("Delay", StringValue ("5ms"));
52  pointToPoint[1].SetChannelAttribute ("Delay", StringValue ("30ms"));
53  pointToPoint[2].SetChannelAttribute ("Delay", StringValue ("50ms"));
54  pointToPoint[3].SetChannelAttribute ("Delay", StringValue ("70ms"));
55  pointToPoint[4].SetChannelAttribute ("Delay", StringValue ("1ms"));
56  pointToPoint[5].SetChannelAttribute ("Delay", StringValue ("3ms"));
57  pointToPoint[6].SetChannelAttribute ("Delay", StringValue ("7ms"));
58
59  for(uint32_t i = 0; i < links; i++){
60    pointToPoint[i].SetDeviceAttribute ("DataRate", StringValue ("5Mbps"));
61    devices[i] = pointToPoint[i].Install (nodeAdjacencyList[i]);
62  }
63
64  InternetStackHelper stack;
65  stack.Install (nodes);
66
67  Ipv4AddressHelper address;
68  std::vector<Ipv4InterfaceContainer> interfaces (links);
69  for (uint32_t i = 0; i < links; i++)
70    {
71      std::ostringstream subset;
72      subset << "10." << i + 1 << ".0.0";
73      address.SetBase (subset.str().c_str (), "255.255.0.0");
74      interfaces[i] =
```

```
75            address.Assign (devices[i]);
76      }
77
78    // We use following code to create the routing table.
79    Ipv4StaticRoutingHelper ipv4RoutingHelper;
80    std::vector<Ptr<Ipv4StaticRouting>> remoteHostStaticRoutingNode (nodesNum);
81    for(uint32_t i = 0; i < nodesNum; i++){
82      remoteHostStaticRoutingNode[i] = ipv4RoutingHelper.GetStaticRouting(nodes.Get (i)->GetObject<
          Ipv4>());
83    }
84
85    uint32_t entryNum = 255;
86    for(uint32_t i = 1;i <= entryNum; i++){
87        std::ostringstream subset;
88        subset << "10.5." << i << ".0";
89      remoteHostStaticRoutingNode[2]->AddNetworkRouteTo(Ipv4Address(subset.str().c_str ()),Ipv4Mask
          ("255.255.255.0"),Ipv4Address("10.5.0.2"),2);
90    }
91
92    // We use the following code, it will auto create the routing table to connect to each other.
93    Ipv4GlobalRoutingHelper::PopulateRoutingTables ();
94
95    //Symbol Address
96    Ptr<Symbolic> symObj0 = CreateObject<Symbolic>();
97    symObj0->SetMinMax(0xa000000, 0xaffffff);
98    Ipv4Address symIP0 = symObj0->GetSymbolicIpv4Add();
99
100   V4PingHelper ping(symIP0);
101   ApplicationContainer pingApp = ping.Install (nodes.Get(0));
102   pingApp.Start (Seconds (1.0));
103   pingApp.Stop (Seconds (1.5));
104
105   Simulator::Stop(Seconds (1.6));
106
107   Simulator::Run ();
108   Simulator::Destroy ();
109
110   symObj0->Stop("Program terminated");
111   return 0;
```

```
112 }
```