Symbolic NS-3 for exhaustive testing

Jianfei Shao, Minh Vu, Lisong Xu
Department of Computer Science and Engineering
University of Nebraska-Lincoln
Lincoln, NE 68588-0115
Email: {jshao, mvu, xu}@cse.unl.edu

***Abstract*—In this project, we propose to extend NS-3 by leveraging symbolic execution so that it can be easily and efficiently used to simulate a network protocol under all possible network environments with respect to some uncertain factors. This short paper describes the current progress and future plans.**

I. 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 in some normal or special network environments. In this project, we consider a special type of simulations (referred to as *exhaustive testing* hereinafter), where a user simulates a protocol in all possible network environments with respect to some uncertain factors. This type of simulations is useful for evaluating the worse-case performance of a protocol where a user is not sure exactly what kind of network environments lead to the worst case, and for detecting possible design or implementation bugs of a protocol as many bugs happen only in corner cases.

In order to conduct exhaustive testing with the current NS-3, a user needs to enumerate and simulate a protocol in each possible network environment (referred to as the brute force method), which is time consuming. In this project, we propose to extend NS-3 by leveraging symbolic execution so that it can be easily and efficiently used for exhaustive testing. The extended NS-3 is referred to as *Symbolic NS-3*, and symbolic execution is a powerful and popular program analysis technique widely used in the software testing and verification community

II. Demo

In this section, we will demonstrate the difference between the current NS-3 and our proposed symbolic NS-3 for exhaustive testing using a toy example.

A. An exhaustive testing problem

Let's consider a network shown in Fig. 1, where two senders and one receiver are connected by two point-to-point links. Each sender $i = 0, 1$ simultaneously sends a UDP packet to the receiver. The capacity of each link is 5 Mbps. 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 how to determine *the range of the possible arrival time difference* $a_0 - a_1$ *considering all possible combinations of* d_0 *and* d_1, where a_i denotes the arrival time at the receiver of the packet from sender i.

Fig. 1. Network topology of the exhaustive testing problem.

B. Brute force using current NS-3

To find the range of $a_0 - a_1$ using the brute force method with the current NS-3, we write two scripts. The shell script `repeatCurrentDemo.sh` as shown in Code 1 enumerates all possible combinations of d_0 and d_1, and runs an NS-3 simulation for each combination. The NS-3 script `currentDemo.cc` as shown in Code 2 simulates the network according to the link delays specified in the arguments.

Code 1. Shell script repeatCurrentDemo.sh

```
1 ...
2 for d0 in {1..1000}
3 do
4   for d1 in {1..1000}
5   do
6     ./waf --run "currentDemo␣--delay0=$d0␣--delay1=$d1"
7   done
8 done
9 ...
```

Code 2. NS-3 script currentDemo.cc

```
1 ...
2 pointToPoint[0].SetDeviceAttribute("DataRate",
      StringValue("5Mbps"));
3 pointToPoint[0].SetChannelAttribute("Delay", TimeValue(
      Time(delay0)));
4 pointToPoint[1].SetDeviceAttribute("DataRate",
      StringValue("5Mbps"));
5 pointToPoint[1].SetChannelAttribute("Delay", TimeValue(
      Time(delay1)));
6 ...
```

In addition, we modify NS-3 file `udp-server.cc` to keep track of the packet arrival times and then calculate and print out the difference $a_0 - a_1$.

Code 3. Modified NS-3 file udp-server.cc

```
1 ...
2 int diff = a0 - a1;
3 std::cout << diff << std::endl;
4 ...
```

It takes a total of 521,900 seconds to run the code, and the total reported range of $a_0 - a_1$ is [-999, 999] ms. All the code is available at https://github.com/JeffShao96/Current-NS3.

C. Symbolic execution using Symbolic NS-3

To find the range of $a_0 - a_1$ using our proposed symbolic NS-3, we write only one script. The NS-3 script `symDemo.cc` as shown in Code 4 simulates the network with two symbolic link delays, each in the range of [1, 1000]ms.

Code 4. Symbolic NS-3 script symDemo.cc

```
1  ...
2  pointToPoint[0].SetDeviceAttribute("DataRate",
       StringValue("5Mbps"));
3  pointToPoint[0].SetChannelAttribute("SymbolicMode",
       BooleanValue(true));
4  pointToPoint[0].SetChannelAttribute("DelayMin",
       TimeValue(Time("1ms")));
5  pointToPoint[0].SetChannelAttribute("DelayMax",
       TimeValue(Time("1000ms")));
6  pointToPoint[1].SetDeviceAttribute("DataRate",
       StringValue("5Mbps"));
7  pointToPoint[1].SetChannelAttribute("SymbolicMode",
       BooleanValue(true));
8  pointToPoint[1].SetChannelAttribute("DelayMin",
       TimeValue(Time("1ms")));
9  pointToPoint[1].SetChannelAttribute("DelayMax",
       TimeValue(Time("1000ms")));
10 ...
```

In addition, we modify NS-3 file `udp-server.cc` to keep track of the packet arrival times, calculate the symbolic difference $a_0 - a_1$, and then calculate and print out the possible range of the symbolic difference.

Code 5. Modified NS-3 file udp-server.cc

```
1 ...
2 int diff = a0 - a1;
3 uintptr_t lower;
4 uintptr_t upper;
5 s2e_get_range(diff, &lower, &upper);
6 s2e_kill_state_printf(0,"The␣range␣of␣diff␣is␣%ld,␣%ld",
      lower,upper);
7 ...
```

It takes a total of only 33 seconds 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 $a_0 - a_1$ is also [-999, 999] ms. All the code is available at https://github.com/JeffShao96/Symbolic-NS3.

III. Symbolic Execution

This section explains the basic idea of symbolic execution [5], [3] that runs a program with symbolic variables using a symbolic execution engine. In this project, we use the powerful symbolic execution platform S^2E [4] to symbolically execute our proposed symbolic NS-3 in virtual machines. The virtual machines are emulated using the QEMU machine emulator [1] and symbolic execution is conducted using the KLEE symbolic execution engine [2]. Different from a normal variable that can take only a concrete value at a time, a symbolic variable can take all possible values satisfying the corresponding constraints,

Let's consider the C-like pseudocode shown in Code 6 as an example. Lines 1 and 2 define two symbolic variables d_0 and d_1 with the same initial constraints, and thus each of them initially could take any value in the range of 1 and 1000. Lines 3 and 4 define two new variables a_0 and a_1, which depend on variables d_0 and d_1 and thus are also treated as symbolic variables with the current constraints.

Code 6. An example for symbolic execution

```
1 sym 1<= d0 <= 1000
2 sym 1<= d1 <= 1000
```

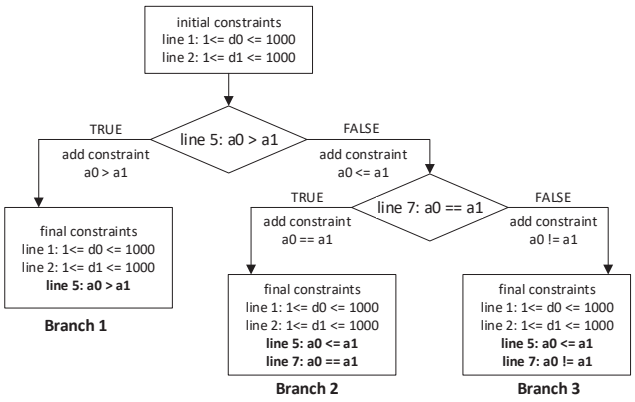

Fig. 2. Three branches are generated during the symbolic execution of Code 6.

```
3  a0 = txtime + d0  //txTime is a constant indicating the
       packet transmission time
4  a1 = txtime + d1
5  if (a0 > a1){
6    //do something
7  }else if (a0==a1){
8    //do something
9  }else{
10   //do something
11 }
12 diff = a0 - a1;
```

When symbolic execution reaches an `if` statement involving symbolic variables, such as lines 5 and 7, the symbolic execution engine checks both possibilities by forking the current execution into two branches. For example, when line 5 `if (a0>a1)` is executed, S^2E forks the current NS-3 virtual machine into two NS-3 virtual machines, called branches, where the true branch continues to line 6 with additional constraint $a_0 > a_1$ and the false branch continues to line 7 with additional constraint $a_0 <= a_1$. Similarly for line 7 `if (a0==a1)`.

Finally, symbolic execution stops with three branches illustrated in Fig. 2, where the final constraints for each branch are also listed. Using these final constraints, we can then calculate the possible range of symbolic variable `diff` defined in line 12. 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. Thus the total range of `diff` is the union of these ranges and is [-999, 999] ms. The result of the symbolic NS-3 mentioned in Section II is obtained in a similar way.

IV. API for Symbolic NS-3

Our current symbolic NS-3 introduces the following new attributes to `PointToPointChannel` so that a user can enable and specify symbolic delays for a point-to-point link.

New channel `bool` attribute `SymbolicMode`: A user sets the delay of a `PointToPointChannel` to a symbolic variable by setting this attribute to true. One example is line 3 of Code 4. By default, `SymbolicMode` is set to false.

New channel `Time` attributes `DelayMin` and `DelayMax`: These two attributes define the minimum and maximum values of the symbolic delay of a `PointToPointChannel`. One example is lines 4 and 5 of Code 4. By default, `DelayMin` is set to "0s" and `DelayMax` is set to "10s".

V. Implementation

We make the following two types of changes to implement the symbolic delay for a point-to-point link. The source code is available at https://github.com/JeffShao96/Symbolic-NS3

A. Channel Attributes

We change both `point-to-point-channel.h` as shown in Code 7 and `point-to-point-channel.cc` as shown in Code 8 to implement the new channel attributes described in the previous section.

Code 7. Modification in point-to-point-channel.h to add new attributes

```
1 private:
2 ...
3   bool m_symbolicDelay;  // Enable or disable the
        symbolic delay
4   bool m_isinitialized = false; // Check whether the
        symbolic delay variable has been initialized
5   Time m_delaymax;  // Maximum value of the symbolic
        delay
6   Time m_delaymin;  // Minimum value of the symbolic
        delay
7 ...
```

Code 8. Modification in point-to-point-channel.cc to add new attributes

```
1 TypeId PointToPointChannel::GetTypeId (void) {
2 ...
3   .AddAttribute("DelayMin", "Minimum␣value␣of␣the␣
        symbolic␣delay", TimeValue(Seconds (0)),
        MakeTimeAccessor (&PointToPointChannel::m_delaymin
        ), MakeTimeChecker ())
4   .AddAttribute("DelayMax", "Maximum␣value␣of␣the␣
        symbolic␣delay", TimeValue(Seconds (10)),
        MakeTimeAccessor (&PointToPointChannel::m_delaymax
        ), MakeTimeChecker ())
5   .AddAttribute("SymbolicMode", "Enable␣or␣disable␣the␣
        symbolic␣delay", BooleanValue (false),
        MakeBooleanAccessor (&PointToPointChannel::
        m_symbolicDelay), MakeBooleanChecker ())
6 ...
7 }
```

B. Symbolic Delay

We change `point-to-point-channel.cc` as shown in Code 9 to initialize and set the range of symbolic delay for a point-to-point link.

Code 9. An example for symbolic execution

```
1 bool PointToPointChannel::TransmitStart(Ptr<const Packet
      > p, Ptr<PointToPointNetDevice> src, Time txTime) {
2   ...
3   if(m_symbolicDelay && !m_isinitialized){
4     uintptr_t m_delayinit = 0;
5     s2e_make_symbolic(&m_delayinit, sizeof(m_delayinit),
          "m_delayinit");
6     m_delay = Time(m_delayinit);
7     if(m_delay<m_delaymin){
8       s2e_kill_state(0,"Out␣of␣Range,␣Lower");
9     }else if(m_delay>m_delaymax){
10      s2e_kill_state(0,"Out␣of␣Range,␣Upper");
11    }
12    m_isinitialized = true;
13  }
14  ...
15 }
```

VI. Future work

This short paper describes our current progress. In the future, we plan to work on the following improvements to symbolic NS-3.

- Symbolic packet delay: We plan to provide more APIs to support symbolic packet delays. For example, the current `PointToPointChannel` has the same symbolic delay for all the packets over the link. We plan to provide another type of symbolic delay so that different packets may have different symbolic delays over a link, such a link can be used to test network protocols under different packet delays, reordering, and jitters.
- Symbolic packet header: We plan to provide APIs so that a packet header may have symbolic fields, such as a symbolic destination IP address.
- Efficiency: We plan to incorporate the techniques proposed in our previous work [6] to address the path explosion problem of symbolic execution and improve the efficiency of symbolic NS-3.

Acknowledgment

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

References

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

[2] 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.

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

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

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

[6] 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.