Computational Verification of Network Programs for Several OpenFlow Switches in Coq

Hiroaki Date and Noriaki Yoshiura(✉)

Department of Information and Computer Sciences, Saitama University,
255, Shimo-ookubo, Sakura-ku, Saitama, Japan
yoshiura@fmx.ics.saitama-u.ac.jp

Abstract. OpenFlow is a network technology that enables to control network equipment centrally, to realize complicated forwarding of packets and to change network topologies flexibly. In OpenFlow networks, network equipment is separated into OpenFlow switches and OpenFlow controllers. OpenFlow switches do not have controllers that usual network equipment has. OpenFlow controllers control OpenFlow switches. OpenFlow controllers are configured by programs. Therefore, network configurations are realized by software. This kind of software can be created by several kinds of programming languages. NetCore is one of them. The verification method of NetCore programs has been introduced. This method uses Coq, which is a formal proof management system. This method, however, deals with only networks that consist of one OpenFlow switch. This paper proposes a methodology that verifies networks that consist of several OpenFlow switches.

1 Introduction

Large scale networks consist of many Layer 2 or Layer 3 network switches. Each of the network switches requires to be configured by network operators. Modification of network topologies or configurations requires modification of configurations of all network equipment. This modification takes much cost for network operations. Moreover, operations of different kinds of network equipment are harder than operations of the same kind of network equipment. One of the aims of OpenFlow is to simplify operations of much network equipment. OpenFlow is a network technology that enables to control network equipment centrally, to realize complicated forwarding of packets and to change network topologies flexibly [5]. In OpenFlow networks, network equipment is separated into OpenFlow switches and OpenFlow controllers. Exactly, OpenFlow is a protocol between OpenFlow switches and OpenFlow controllers. OpenFlow switches do not have controllers that usual network equipment has. OpenFlow controllers control OpenFlow switches. OpenFlow controllers are configured by programs. Therefore, network configurations are realized by software. OpenFlow is one of software defined networks (SDNs).

Network switches in OpenFlow forward packets like Layer 2 or Layer 3 switches, but the way of forwarding packets is not configured in network switches.

O. Gervasi et al. (Eds.): ICCSA 2016, Part II, LNCS 9787, pp. 223–238, 2016.
DOI: 10.1007/978-3-319-42108-7_17

The way of forwarding packets is decided by OpenFlow controllers as follows; OpenFlow switches send messages to OpenFlow controllers after receiving packets. The messages include the information of the packets that are received by the OpenFlow switches. The information includes source and destination IP addresses, source and destination port numbers, protocol number and so on After receiving the messages, OpenFlow controllers send messages to OpenFlow switches. The messages include the instructions of forwarding the packets. OpenFlow switches forward the packets according to the instructions in the messages. In OpenFlow, the instructions are called flow entries. After receiving the flow entries, OpenFlow switches keep the flow entries in itself. OpenFlow switches receive the same kind of packets (for example, the packets that have the same destination IP addresses) and forward the packets according to the flow entries without sending the messages to OpenFlow controllers. Therefore, OpenFlow switches do not have controllers in itself and the way of forwarding packets is not configured in OpenFlow switches. Flow entries are kept in a predefined time or as long as possible. After the time is expired or when cache memory for flow entries is full, the flow entries are erased. Modification of network configuration requires modification of the configuration of OpenFlow controllers without modification of OpenFlow switches. Even if a network consists of network switches that are manufactured in different companies, modification of configurations of OpenFlow controllers is enough to modify the network configuration in the case that the network switches can deal with OpenFlow protocol.

There is several hardware of OpenFlow controllers, but some software enables usual PCs to be used as OpenFlow controllers. The examples of such software are Trema [18], NetCore [6] and so on. NetCore is a programming language that can be used in functional programming language Haskell [17]. This feature is preferable for program verification and some researches use Coq for NetCore program verification. Coq is an interactive theorem prover and a formal proof management system [16]. If Coq proves that a program satisfies the properties that should be satisfied by the program, Coq reduces the cost of checking programs and guarantees the correctness of programs better than software testing. There are several researches that use Coq to verify software [2,10]; one is to use Coq to guarantee programs by typing [12]. In the classical program verification approach, a programs and their specifications are described separately and the verification procedure proves that the programs satisfy their specifications. Coq enables to create and verify programs simultaneously. In this case, specifications are expressed as types [9].

There are also several kinds of researches that verify OpenFlow programs: testing method [4,14,15], model checking [1,7,8], and proof system [11,13]. There are also several researches that use Coq to verify the programs in NetCore. Suppose that *pg* is a NetCore program, P is the precondition of *pg* and Q is the postcondition of *pg*. To verify the program *pg*, Coq proves a Hoare logic formula {P}*pg*{Q}, which represents that if P is satisfied before the program *pg* is executed, Q is satisfied after the program *pg* is executed. The previous research [11] verifies Hoare logic formulas by two steps; the first step is to calculate the minimum precondition that satisfies the postcondition Q after executing

the program *pg*. The second step is to check whether P implies the minimum precondition. The previous research verified NAT (Network Address Translation) in NetCore programs but dealt with only network topologies that consist of one network switch.

This paper proposes a methodology that verifies programs for network topologies that consist of several network switches. Especially, this paper verifies that NetCore programs do not generate looping packets in the networks. First, to realize this verification, this paper tries verifying a whole of a program for several network switches. However, this paper cannot verify a whole of a program because of shortage of memory and execution time. Therefore, this paper verifies a program for each network switch by Coq and manually checks that a whole program for a network does not generate looping packets by using the result of verification in Coq. As a result, this paper proposes the methodology of verification of programs and shows that the methodology is efficient.

This paper is organized as follows; Sect. 2 explains NetCore and Coq. Section 3 explains the methodology of verification. Section 4 shows the example of using the methodology that is proposed in this paper. Section 5 discusses the proposed methodology. Section 6 concludes this paper.

2 Preliminary

2.1 NetCore

This section explains NetCore programs. NetCore is a declarative programming language and the programmers describe only the way of forwarding packets. For example, let me set up the program for an OpenFlow controller so that an OpenFlow switch sends packets that are received from network interface "Port 2", to network interface "Port 1" and drops all other packets. First, the following program of NetCore is created.

```
Definition pg1 := WILD /=> FWD 1.
```

This program sends all packets to "Port 1". The following explains the syntax of Netcore briefly; "Definition pg1 :=" defines the program "pg1", the left side of "`/=>`" is a condition and the right side of "`/=>`" is an action. "WILD" represents true, that is to say "WILD" represents all packets. "FWD 1" represents sending packets to "Port 1". This program is not enough because the program does not specify the condition of forwarding packets. The following program restricts the packets that are forwarded.

```
Definition pg2 := RESTRICT pg1 BY PORT=2.
```

This program sends only packets that are received by "Port 2", to "Port 1". "RESTRICT x BY y" represents that the packets that satisfy the condition "y" is applied to program "x". "PORT=2" represents the condition that the packets are received by "Port 2". The program "pg2" realizes that an OpenFlow

switch sends packets that are received by network interface "Port 2", to network interface "Port 1" and drops all other packets. NetCore can represent many kinds of conditions and processes for packets. For example, NetCore can describe static NAT, firewalls and so on.

2.2 Coq

This subsection explains how NetCore programs are verified by Coq. This verification is based on Hoare logic [3]. A Hoare logic formula is the following description

$$\{\mathrm{P}\}pg\{\mathrm{Q}\}$$

where P is a precondition, pg is a NetCore program and Q is a postcondition. The Hoare logic formula represents that if the packets that arrive at an OpenFlow switch satisfy a precondition P, a post condition Q is satisfied after a program pg is applied to the packets. Verification of programs is to check whether Hoare logic formulas are satisfied. The previous researches proposed the method of checking a Hoare logic formula $\{\mathrm{P}\}pg\{\mathrm{Q}\}$. The method first deduces the weakest precondition so that a postcondition is satisfied after a program pg is applied to packets; in the following, $wp(pg, \mathrm{Q})$ is defined to be the weakest precondition. Next, the method checks whether P satisfies $wp(pg, \mathrm{Q})$ to check whether $\{\mathrm{P}\}pg\{\mathrm{Q}\}$. The following shows the verification of pg2 by using Coq.

```
Lemma verification: |-[PORT=2] pg2 [PORT=1].
Proof. checker. Qed.
```

In this description, "Lemma verification" means that Coq checks whether a proof "verification" holds or not. `|-[PORT=2] pg2 [PORT=1]` represents a Hoare logic formula {a packet arrives at Port 2}pg2{the packet departs from Port 1}. Concretely, "checker", which is a function for verification, verifies that this Hoare logic formula holds. The function "checker" calculates $wp(pg, \mathrm{Q})$ and checks whether P implies $wp(pg, \mathrm{Q})$. In this case, "Qed" shows that the Hoare logic formula holds. This paper uses the same method that is proposed by the previous research.

3 Methodology of Verification

This section explains the methodology that is proposed in this paper. For explanation, this section uses a network topology in Fig. 1. This section creates programs for the network topology and verifies that the programs do not generate looping packets. This section generates two programs: one program that does not generate looping packets and the other program that generates looping packets. Appendix A is a program that does not generate looping packets and Appendix B is a program that generates looping packets.

First, this paper tries to prove that the program in Appendix A does not generate looping packets; that is to say, this paper tries to prove a Hoare logic

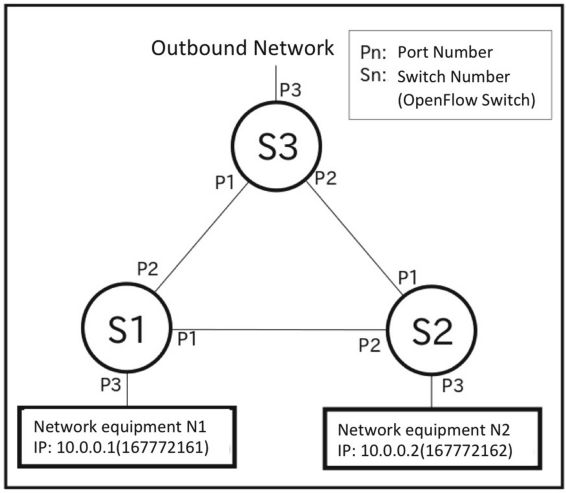

Fig. 1. Network topology

formula {P}*pg*{Q} where P is a precondition, Q is a postcondition and *pg* is the program in Appendix A. However, the program in Appendix A is too large to prove the Hoare logic formula. In fact, this paper cannot prove the Hoare logic formula by Coq because it lacks memories and takes much time.

Nowadays, CPUs become powerful and memory size of PCs become large, but the program in Appendix A is still large for verification by Coq. Those who have the ability of using Coq may prove the Hoare logic formula in the current PCs. However, many people cannot obtain this ability and it is important to prove the Hoare logic formula without high abilities of Coq and Hoare logic. Therefore, this paper proposes the methodology that enables those who are not familiar with Coq or Hoare logic to verify programs by Coq.

The overview of the methodology is as follows; suppose that the methodology checks that a program satisfies a property *Prop* at a network that consists of several network switches.

1. To create the properties that each switch should satisfy so that the program satisfies the property *Prop*.
2. To check that each switch satisfies the properties by Coq.
3. To prove that the program satisfies the property *Prop* by using the results that each switch satisfies the properties.

The point of this methodology is the first step, which is to create the properties for each switch. The first step separates the property *Prop* into several small properties for each switch. However, this separation cannot be accomplished automatically. In this paper, this separation is accomplished manually.

4 Verification

The section proves that the program in Appendix A does not generate looping packets by using the methodology that is proposed in this paper. The proof requires the description of network topologies in Coq. The Fig. 2 is a description of the network topology in Coq. There are several researches that deal with the descriptions of network topologies. The description in Fig. 2 is based on the previous research [11].

```
Definition def_topo: topo :=
{| ports := fun sw =>
            match sw with
              | 1 => [Word16.repr 1; Word16.repr 2; Word16.repr 3]
              | 2 => [Word16.repr 1; Word16.repr 2; Word16.repr 3]
              | 3 => [Word16.repr 1; Word16.repr 2; Word16.repr 3]
              | _ => nil
             end;
num_links := 3;
switch_topo := fun loc =>
            match loc with
              |Build_location 1 x =>
               if Word16.eq x (Word16.repr 1)
               then Some (Build_location 2 (Word16.repr 2))
               else if Word16.eq x (Word16.repr 2)
                    then Some (Build_location 3 (Word16.repr 1))
                    else None
              |Build_location 2 x =>
               if Word16.eq x (Word16.repr 1)
               then Some (Build_location 3 (Word16.repr 2))
               else None
              | _ => None
            end;
switch_topo' := fun loc =>
             match loc with
               |Build_location 2 x =>
                if Word16.eq x (Word16.repr 2)
                then Some (Build_location 1 (Word16.repr 1))
                else None
               |Build_location 3 x =>
                if Word16.eq x (Word16.repr 1)
                then Some (Build_location 1 (Word16.repr 2))
                else if Word16.eq x (Word16.repr 2)
                     then Some (Build_location 2 (Word16.repr 1))
                     else None
               | _ => None
             end |}.
```

Fig. 2. Network topology

```
Definition s_to_s' (ver_topo: topo)(loc_from loc_to: location) : bool :=
  match (ver_topo.(switch_topo) loc_from) with
    | Some loc' => loc_eq loc' loc_to
    | _ =>
       match (ver_topo.(switch_topo') loc_from) with
         |Some loc' => loc_eq loc' loc_to
         |_ => false
       end
  end.
```

Fig. 3. Function

In "`Def_topo`" in Fig. 2, "`ports`" represents the network interfaces for each network switch. "`num_links`" represents the number of connections. Each of the network switches in Fig. 1 has three network interfaces and the network topology has three connections. "`switch_topo`" and "`switch_topo'`" shows the connections among switches. Concretely, "`switch_topo`" and "`switch_topo'`" are functions that map a switch number and a port number to the other switch number and the other port that are connected with the switch number and the port number. The function "`s_to_s'`" in Fig. 3 shows connection between switches.

The function "`s_to_s'`" receives a network topology, two pairs of a switch number and a port number as the first, second and third arguments. If the two pairs of the switch number and the port number are connected in the network topology, the function outputs true. Otherwise, the function outputs false.

By using this description, this paper checks whether both programs of Appendices A and B satisfy the property, which is loop free. This paper uses a PC that has Ubuntu 14.04 LTS as OS, Core i5-6600K as CPU and 8 GB memory. This paper uses Coq several times for several proofs and to execute each proof takes less than five seconds.

4.1 Overview of Verification

This paper verifies the programs according to the methodology that is proposed in this paper. First, this paper tries to verify that the program in Appendix A does not generate looping packets. According to the methodology, the property that looping packets are not generated is separated into several properties that are satisfied by each switch. This paper focuses on three kinds of packets whose destinations are 10.0.0.1, 10.0.0.2 and the others. This paper uses Coq to check how each switch deals with these kinds of packets. The results of using Coq are used to verify that looping packets are not generated in the program in Appendix A.

Figure 4 shows the proofs of Coq for the program in Appendix A. Figure 4 includes the proof results of thirteen lemmas. Each lemma represents a property of each switch. For example, Lemma `tst1` represents that the packets that are received by Port 3 of Switch 1 and whose destination is 10.0.0.2 are forwarded to

```
Lemma tst1 : triple (SWITCH=1 AND PORT=3 AND NWDST =167772162)
             pg100 (PORT=1).
Proof. checker. Qed.
Eval compute in s_to_s' def_topo ( sw_pt 1 1) ( sw_pt 2 2).
Lemma tst1': triple (SWITCH=2 AND NOT PORT=3 AND (PORT=1 OR PORT =2) AND
             NWDST =167772162) pg200 (PORT =3).
Proof. checker. Qed.

Lemma tst2 : triple (SWITCH=1 AND PORT=3 AND NOT NWDST=167772162 AND
             NOT NWDST =167772161) pg100 (PORT =2).
Proof. checker. Qed.
Eval compute in s_to_s' def_topo ( sw_pt 1 2) ( sw_pt 3 1).
Lemma tst2': triple (SWITCH =3 AND (PORT =1 OR PORT =2) AND NOT
             NWDST =167772161 AND NOT NWDST =167772162) pg300 ( PORT =3).
Proof. checker. Qed.

Lemma tst3 : triple (SWITCH =2 AND PORT =3 AND NOT NWDST =167772161 AND
             NOT NWDST =167772162) pg200 ( PORT =1).
Proof. checker. Qed.
Eval compute in s_to_s' def_topo ( sw_pt 2 1) ( sw_pt 3 2).
Lemma tst3': triple (SWITCH=2 AND NOT PORT=3 AND (PORT=1 OR PORT=2) AND
             NWDST =167772162) pg200 ( PORT =3).
Proof. checker. Qed.

Lemma tst4 : triple (SWITCH=2 AND PORT=3 AND NDST=167772161) pg200
             (PORT=2).
Proof. checker. Qed.
Eval compute in s_to_s' def_topo ( sw_pt 2 2) ( sw_pt 1 1).
Lemma tst4': triple (SWITCH =1 AND NOT PORT =3 AND (PORT=1 OR  PORT=2)
             AND NWDST =167772161) pg100 ( PORT =3).
Proof. checker. Qed.

Lemma tst5 : triple (SWITCH=3 AND PORT=3 AND NWDST =167772161) pg300
             (PORT=1).
Proof. checker. Qed.
Eval compute in s_to_s' def_topo ( sw_pt 3 1) ( sw_pt 1 2).
Lemma tst5': triple (SWITCH =1 AND NOT PORT =3 AND (PORT =1 OR PORT =2)
             AND NWDST =167772162) pg100 (PORT =3).
Proof. checker. Qed.

Lemma tst6 : triple (SWITCH =3 AND PORT=3 AND NWDST =167772162) pg300
             ( PORT =2).
Proof. checker. Qed.
Eval compute in s_to_s' def_topo ( sw_pt 3 2) ( sw_pt 2 1).
Lemma tst6': triple ( SWITCH =2 AND NOT PORT =3 AND (PORT=1 OR  PORT=2)
             AND NWDST =167772162) pg200 ( PORT =3).
Proof. checker. Qed.

Lemma tst7 : triple (SWITCH =3 AND PORT =3 AND NOT NWDST =167772161 AND
             NOT NWDST =167772162) pg300 ( NOT WILD ).
Proof. checker. Qed.
```

Fig. 4. Proof of the program in Appendix A

Port 1 of Switch 1. Lemma `tst1'` represents that the packets that are received by Port 1 or Port 2 of Switch 2 and whose destination is 10.0.0.2 are forwarded to Port 3 of Switch 2, and Lemma `tst6` represents that the packets that are received by Port 3 of Switch 3 and whose destination is 10.0.0.2 are forwarded to Port 2 of Switch 3. These three lemmas, which are proved by Coq, are related with the packets whose destination is 10.0.0.2. These lemmas imply that the packets arrive at the network N2. As a result, the packets whose destinations are 10.0.0.2 do not loop in the network topology. Figure 4 includes several lemmas, which are proved by Coq, for the packets whose destination are 10.0.0.1 and the others. As the case of the packets whose destinations are 10.0.0.2, these lemmas imply that the packets whose destination are 10.0.0.1 and the others do not loop in the network topology.

Figure 5 shows the proofs of Coq for the program in Appendix B. Since this program generates looping packets, the lemmas in Fig. 5 show that the packet whose destination is 10.0.0.2 loop in the network topology.

```
Lemma tst1 : triple (SWITCH =3 AND PORT=3 AND NWSRC=167772164 AND
             NWDST=167772162) pg300 (PORT=1).
Proof. checker. Qed.

Eval compute in s_to_s' def_topo (sw_pt 3 1) ( sw_pt 1 2).

Lemma tst2 : triple (SWITCH =1 AND PORT=2 AND NWSRC =167772164 AND
             NWDST =167772162) pg100 ( PORT =1).
Proof. checker. Qed.

Eval compute in s_to_s' def_topo ( sw_pt 1 1) ( sw_pt 2 2).

Lemma tst3 : triple (SWITCH=2 AND PORT=2 AND NWSRC=167772164 AND
             NWDST=167772162) pg200 (PORT=1).
Proof. checker. Qed.

Eval compute in s_to_s' def_topo ( sw_pt 2 1) ( sw_pt 3 2).

Lemma tst4 : triple (SWITCH=3 AND PORT=2 AND NWSRC=167772164 AND
             NWDST =167772162) pg300 (PORT =1).
Proof. checker. Qed.
```

Fig. 5. Proof of the program in Appendix B

5 Discussion

The verification that is described in the previous section shows that one program creates looping packets and the other does not create looping packets. Actually

this paper would like to verify a whole of all programs. However, the function "`checker`" does not terminate to verify the whole of all programs because the verification is short of memory of the PC and takes much time. Therefore, this paper verifies a behavior of each switch and those who try to verify programs use the results of the verification for each switch to prove manually that the a program does not generate looping packets.

The network topology has three network switches and there are three destinations of packets; the first is N1, the second is N2 and the third is outbound. All switches may deal with all destination packets. Therefore, this paper requires three proofs for each of three switches and totally requires nine proofs to verify that a program does not generate looping packets. Since this paper deals with the simple network topology, nine proofs are enough for the verification. However, verifications in complicated network topologies require many proofs for many network switches. Moreover, those who try to verify programs must deal with many proofs. Therefore, the methodology that is proposed in this paper is hard to use for complicated network topologies.

6 Conclusion

This paper proposed the methodology that verifies the programs that are described in NetCore. In this methodology, manual proof and Coq verify the programs. The future works are to develop the method to verify the programs automatically and to improve the proposed methodology to deal with complicated network topologies.

A Appendix (Program that Does Not Generate Looping Packets)

```
(* s1 *)

(* s1_p1 *)

Definition pg101 := WILD /= > FWD 1.

Definition pg102 := RESTRICT pg101 BY (PORT=3 AND NWDST=167772162).

(* s1_p2 *)
Definition pg103 := WILD /= > FWD 2.

Definition pg104 := RESTRICT pg103 BY (PORT=3 AND (NOT NWDST=167772162)).

(* s1_p3 *)
Definition pg105 := PORT =1 /= > FWD 3.

Definition pg106 := PORT =2 /= > FWD 3.
```

```
Definition pg107 := RESTRICT (pg105 PAR pg106) BY
                 (NWDST=167772161 AND NOT PORT =3).

(* s1_behave *)
Definition pg100 := RESTRICT ( pg102 PAR pg104 PAR pg107 ) BY SWITCH =1.

(* s2_p1 *)
Definition pg201 := WILD /= > FWD 1.

Definition pg202 := RESTRICT pg201 BY (PORT=3 AND (NOT NWDST=167772161)).

(* s2_p2 *)

Definition pg203 := WILD /= > FWD 2.

Definition pg204 := RESTRICT pg203 BY ( PORT =3 AND NWDST=167772161).

(* s2_p3 *)
Definition pg205 := PORT =1 /= > FWD 3.

Definition pg206 := PORT =2 /= > FWD 3.

Definition pg207 := RESTRICT ( pg205 PAR pg206 ) BY
                 ( NWDST =167772162 AND NOT PORT =3).

(* s2_drop *)

(* s2_behave *)
Definition pg200 := RESTRICT (pg202 PAR pg204 PAR pg207 ) BY SWITCH =2.

(* s3_p1 *)
Definition pg301 := WILD /= > FWD 1.

Definition pg302 := RESTRICT pg301 BY (PORT =3 AND NWDST=167772161).

(* s3_p2 *)
Definition pg303 := WILD /= > FWD 2.

Definition pg304 := RESTRICT pg303 BY ( PORT =3 AND NWDST=167772162).

(* s3_p3 *)
Definition pg305 := WILD /= > FWD 3.

Definition pg306 := RESTRICT pg305 BY ((PORT =1 OR PORT=2) AND
                    NOT PORT =3 AND NOT NWDST =167772161 AND
                    NOT NWDST =167772162).

(* s3_behave *)
Definition pg300 := RESTRICT ( pg302 PAR pg304 PAR pg306 ) BY SWITCH =3.
```

```
Require   Import   List.
Require   Import   Bool.
Require   Import   Arith.
Require   Import   WP.

Record topo : Type :=
  {
     ports : nat -> list Word16.t ;
     num_links : nat ;
     switch_topo : location -> option location ;
     switch_topo ': location -> option location
  }.

Require Import List.
Import ListNotations.
Close Scope Z_scope.
Definition def_topo : topo :=
  {| ports := fun sw = >
           match sw with
            | 1 = > [ Word16.repr 1; Word16.repr 2; Word16.repr 3]
            | 2 = > [ Word16.repr 1; Word16.repr 2; Word16.repr 3]
            | 3 = > [ Word16.repr 1; Word16.repr 2; Word16.repr 3]
            | _ = > nil
          end ;
     num_links := 3;
     switch_topo := fun loc = >
                 match loc with
                   | Build_locatio n 1 x = >
                     if Word16.eq x ( Word16.repr 1)
                     then Some ( Build_location 2 ( Word16.repr 2))
                     else if Word16.eq x ( Word16.repr 2)
                          then Some ( Build_location 3 ( Word16.repr 1))
                          else None
                   | Build_location 2 x = >
                     if Word16.eq x ( Word16.repr 1)
                     then Some ( Build_location 3 ( Word16.repr 2))
                     else None
                   | _ = > None
                 end ;
     switch_topo ' := fun loc = >
                   match loc with
                     | Build_lo cation 2 x = >
                       if Word16.eq x ( Word16.repr 2)
                       then Some ( Build_location 1 ( Word16.repr 1))
                       else None
                     | Build_lo cation 3 x = >
                       if Word16.eq x ( Word16.repr 1)
                       then Some ( Build_location 1 ( Word16.repr 2))
                       else if Word16. eq x ( Word16.repr 2)
                            then Some (Build_location 2 (Word16.repr 1))
```

```
                        else None
                    | _ = > None
                  end |}.

Definition loc_eq ( loc1 loc2 : location ): bool :=
  match loc1 , loc2 with
    | Bu ild_location sw1 pt1 ,
      Build_location sw2 pt2 = > beq_nat sw1 sw2 && Word16.eq pt1 pt2
  end.


Definition s_to_s' (ver_topo:topo)(loc_from loc_to:location):bool:=
  match ( ver_topo.( switch_topo ) loc_from ) with
    | Some loc' => loc_eq loc' loc_to
    | _ =>
       match ( ver_topo.( switch_topo') loc_from) with
         | Some loc' = > loc_eq loc' loc_to
         | _ = > false
       end
  end.

Definition sw_pt ( n : nat )( m : BinNums.Z ): location :=
  Build_location n ( Word16.repr m ).
```

B Appendix (Program that Generates Looping Packets)

```
(* s1_p1 *)
Definition pg101 := WILD /= > FWD 1.

Definition pg102 := RESTRICT pg101 BY ((PORT=2 OR PORT=3) AND
                 NOT PORT=1 AND NWDST=167772162).

(* s1_p2 *)
Definition pg103 := WILD /= > FWD 2.

Definition pg104 := RESTRICT pg103 BY ( PORT =3 AND NOT NWDST=167772162).

(* s1_p3 *)
Definition pg105 := WILD /= > FWD 3.

Definition pg106 := RESTRICT pg105 BY ((PORT=1 OR PORT=2) AND NOT PORT=3
                 AND NWDST=167772161).

(* s1_behave *)
Definition pg100 := RESTRICT ( pg102 PAR pg104 PAR pg106 ) BY SWITCH =1.

(* s2_p1 *)
Definition pg201 := WILD /= > FWD 1.
```

```
Definition pg202 := RESTRICT pg201 BY ((PORT=2 OR PORT =3) AND
                NOT PORT=1).

(* s2_behave *)
Definition pg200 := RESTRICT pg202 BY SWITCH =2.

(* s3_p1 *)
Definition pg301 := WILD /= > FWD 1.

Definition pg302 := RESTRICT pg301 BY ((PORT =2 OR PORT=3) AND NOT PORT=1
                   AND (NWDST=167772161 OR NWDST=167772162)).

(* s3_p3 *)
Definition pg303 := WILD /= > FWD 3.

Definition pg304 := RESTRICT pg303 BY PORT =1.

(* s3_behave *)
Definition pg300 := RESTRICT (pg302 PAR pg304 ) BY SWITCH =3.

Record topo : Type :=
  {
     ports : nat -> list Word16.t ;
     num_links : nat ;
     switch_topo : location -> option location ;
     switch_topo ': location -> option location
  }.

Definition def_topo : topo :=
  {| ports := fun sw = >
           match sw with
             | 1 = > [ Word16.repr 1; Word16.repr 2; Word16.repr 3]
             | 2 = > [ Word16.repr 1; Word16.repr 2; Word16.repr 3]
             | 3 = > [ Word16.repr 1; Word16.repr 2; Word16.repr 3]
             | _ = > nil
            end ;
     num_links := 3;
     switch_topo := fun loc = >
                 match loc with
                   | Build_locatio n 1 x = >
                     if Word16.eq x ( Word16.repr 1)
                     then Some ( Build_location 2 ( Word16.repr 2))
                     else if Word16.eq x ( Word16.repr 2)
                          then Some ( Build_locati on 3 ( Word16.repr 1))
                          else None
                   | Build_locatio n 2 x = >
                     if Word16.eq x ( Word16.repr 1)
                     then Some ( Build_location 3 ( Word16.repr 2))
                     else None
                   | _ = > None
```

```
                end ;
    switch_topo ' := fun loc = >
                  match loc with
                    | Build_lo cation 2 x = >
                      if Word16.eq x ( Word16.repr 2)
                      then Some ( Bui ld_location 1 ( Word16.repr 1))
                      else None
                    | Build_lo cation 3 x = >
                      if Word16.eq x ( Word16.repr 1)
                      then Some ( Bui ld_location 1 ( Word16.repr 2))
                      else if Word16.eq x ( Word16.repr 2)
                           then Some (Build_location 2 ( Word16.repr 1))
                           else None
                           | _ = > None
                  end |}.

Definition loc_eq ( loc1 loc2 : location ): bool :=
  match loc1 , loc2 with
    | Bu ild_location sw1 pt1 ,
      Build_location sw2 pt2 = > beq_nat sw1 sw2 && Word16.eq pt1 pt2
  end.

Definition s_to_s' (ver_topo:topo)(loc_from loc_to:location ):bool :=
  match ( ver_topo.( switch_topo ) loc_from ) with
    | Some loc' = > loc_eq loc' loc_to
    | _ =>
  match (ver_topo.( switch_topo') loc_from ) with
              | Some loc' = > loc_eq loc' loc_to
              | _ = > false
            end
     end.

Open Scope Z_scope.
Definition sw_pt (n : nat)(m : BinNums.Z): location :=
  Build_location n ( Word16.repr m).
```

References

1. Canini, M., Venzano, D., Perešíni, P., Kostić, D., Rexford, J.: A NICE way to test openflow applications. In: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (2012)
2. Garcia, R., Tanter, É., Wolff, R., Aldrich, J.: Foundations of typestate-oriented programming. ACM Trans. Program. Lang. Syst. **36**(4), 1–44 (2014)
3. Hoare, C.A.R.: An axiomatic basis for computer programming. Commun. ACM **12**(19), 576–580 (1969)
4. Kuzniar, M., Peresini, P., Canini, M., Venzano, D., Kostic, D.: A SOFT way for openflow switch interoperability testing. In: Proceedings of the 8th International Conference on Emerging Networking Experiments and Technologies, pp. 265–276 (2012)

5. McKeown, N., Anderson, T., Balakrishnan, H., Parulker, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: OpenFlow: enabling innovation in campus networks. SIGCOMM Comput. Commun. Rev. **38**(2), 69–74 (2008)
6. Monsanto, C., Foster, N., Harrison, R., Walker, D.: A compiler and runtime system for network programming languages. In: Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pp. 217–230 (2012)
7. Majumdar, R., Tetali, S.D., Wang, Z.: Kuai: a model checker for software-defined networks. In: Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design, pp. 163–170 (2014)
8. Sethi, D., Narayana, S., Malik, S.: Abstractions for model checking SDN controllers. In: Formal Methods in Computer-Aided Design, pp. 145–148 (2013)
9. Sheard, T., Stump, A., Weirich, S.: Language-based verification will change the world. In: Proceedings of the FSE/SDP Workshop on the Future of Sofware Engineering Research, pp. 343–348 (2010)
10. Siek, J., Taha, W.: Gradual typing for functional languages. In: Proceedings of the Scheme and Functional Programming Workshop, pp. 81–92, September 2006
11. Stewart, G.: Computational verification of network programs in Coq. In: Gonthier, G., Norrish, M. (eds.) CPP 2013. LNCS, vol. 8307, pp. 33–49. Springer, Heidelberg (2013)
12. Tanter, É., Tabareau, N.: Gradual certified programming in Coq. In: Proceedings of the 11th Symposium on Dynamic Languages, pp. 26–40 (2015)
13. Ball, T., Bjørner, N., Gember, A., Itzhaky, S., Karbyshev, A., Sagiv, M., Schapira, M., Valadarsky, A.: VeriCon: towards verifying controller programs in software-defined networks. SIGPLAN Not. **49**(6), 282–293 (2014). PLDI 2014
14. Wu, Y., Haeberlen, A., Zhou, W., Loo, B.T.: Answering why-not queries in software-defined networks with negative provenance. In: Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks, pp. 1–7 (2013)
15. Wundsam, A., Levin, D., Seetharaman, S., Feldmann, A.: OFRewind: enabling record and replay troubleshooting for networks. In: Proceedings of the USENIX Annual Technical Conference (2011)
16. The Coq Proof Assistant. https://coq.inria.fr
17. Haskell. https://www.haskell.org
18. NEC: "Trema Openflow Controller". http://trema.github.com/trema/