Efficient Model Checking of OpenFlow Networks Using SDPOR-DS

Yutaka Yakuwa, Nobuyuki Tomizawa, Toshio Tonouchi
Knowledge Discovery Research Laboratories
NEC Corporation
Kawasaki, Kanagawa 211-8666, Japan
y-yakuwa@ap.jp.nec.com, n-tomizawa@ap.jp.nec.com, tonouchi@cw.jp.nec.com

***Abstract*—OpenFlow is one of the most popular protocols to realize Software-Defined Networking. OpenFlow has attracted a great deal of interest because of its wide utility and applicability for automation of network management. While OpenFlow provides the ability to control a network using software, there is the risk of bugs occurring in the software that could cause erroneous network behavior. Therefore, improving the reliability of the network is very important. Model checking is a well-known technique to verify the correctness of distributed systems such as OpenFlow networks. However, it is difficult to apply it to this problem because model checking takes an exponential amount of time in relation to the scale of its target. Naïve model checking may take too much time, even to verify a toy network. We introduce an effective method for model checking of the OpenFlow network. Our method reduces the state-explosion problem with dynamic partial-order reduction and with state transition based on symbolic execution. We implemented a prototype for our method to evaluate it. The results indicated that our method completed model checking in less than 10% of the execution time of naïve depth first search model checking and in 31% of the execution time of an existing state-of-the-art tool.**

Keywords—Software-Defined Networking; SDN; OpenFlow; model checking; formal methods

I. INTRODUCTION

A. OpenFlow

OpenFlow [1] is the first standard to be accepted widely by both academia and industry for Software-Defined Networking (SDN). By using OpenFlow, we can automate the operation and management of networks with one centralized OpenFlow controller. With programs installed in the controller, networks become programmable and configurable more dynamically. We expect that a huge variety of programs written on an OpenFlow framework such as Trema [2] or NOX [3] will be developed by vendors or network administrators for almost all network operation and management functions in the future.

B. Challenges of OpenFlow Verification

Although networks become more flexible with the programs installed in the OpenFlow controller, there is the risk of bugs in the programs having an adverse effect on the networks' behavior. Therefore, it is important to make such networks more reliable by identifying and fixing problems that occur in them. If OpenFlow programs developed by network administrators include bugs, ideally, those administrators should identify and fix the bugs themselves, although this is a difficult task. Therefore, some ways to automatically check the "correctness" of OpenFlow networks are necessary.

Generally, the correctness of networks is tested with naïve tools such as *ping* and *traceroute*. However, it is difficult to make sure a network is operating rightly because networks are inherently distributed and asynchronous. OpenFlow networks have the same problem. For example, a delay of OpenFlow messages may cause unexpected behavior. Fig. 1 illustrates an example in which a packet is received by *switch 2* before the installation of a new flow entry that should be installed in *switch 2* before the packet reaches it. In this case, *switch 2* applies an old flow entry to the packet or sends a *packet-in* message to the controller unexpectedly, and the packet may then be processed in the wrong manner (e.g., forwarded to the wrong destination or dropped erroneously). Such problems occur only under certain event orderings, so they are hard to detect by general testing methods using the naïve tools (*ping*, *traceroute*, etc.). Thus, more rigorous and systematic methods are needed to verify the correctness of networks.

Model checking [4] is a powerful technique for verifying distributed and asynchronous systems. It involves modeling a target system as a finite state machine and exploring it exhaustively to detect wrong states or paths. It enables all bugs in the model to be detected without any omissions.

However, it is difficult to apply it simply to this problem because model checking takes an exponential amount of time in relation to the scale of the target system. This is called the "*state-explosion problem.*" In model checking of OpenFlow networks, two types of state-explosion problems occur; the first is caused by the massive number of packet patterns, and the second is caused by the massive number of orders of network events. OpenFlow switches and controllers can react to arbitrary packets based on their headers, which have a huge number of bit patterns. Model checking should cover all of these patterns. This is the first state-explosion problem. In addition, network events such as receiving packets can occur in any order due to communication delays. Model checking should also cover all variations in orders. This is the second state-explosion problem. These problems need to be resolved in order to apply model checking to OpenFlow.

NICE [5] is an existing state-of-the-art tool for model checking of OpenFlow networks. When a tested OpenFlow model has bugs, NICE detects them quickly (in less than 1 min. in most examples in [5]). However, when the tested OpenFlow model has no bugs and an exhaustive search is

This work was partly supported by the Ministry of Internal Affairs and Communications, Japan.

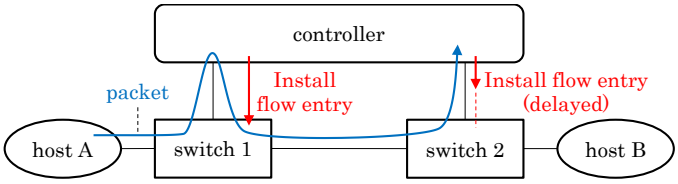

Fig. 1. Example of OpenFlow networks.

needed, NICE took a long time to complete the model checking in some cases (more than 4 days in the worst example in [5]). It is therefore necessary to reduce the time required for an exhaustive search.

C. Contributions

The contributions of our research are as follows.

- SDPOR-DS: We propose a novel method based on dynamic partial-order reduction (DPOR) [6] to reduce the amount of space to be explored for the verification of distributed systems such as OpenFlow networks.
- Symbolic state transition model of OpenFlow networks: We incorporate the concept of symbolic execution [7] into the state transition model of the OpenFlow network.

We evaluated our method with a prototype. As a result, we were able to verify its effectiveness as follows: our method completed the model checking for the exhaustive exploration (i) in less than 10% of the execution time of the naïve depth first search (DFS) model checking and (ii) in 31% of the execution time of the state-of-the-art tool NICE.

II. Background

A. Model Checking

Model checking [4] is a method of verifying software or systems. It is used to verify whether such a verification target satisfies functional specifications or has rare errors caused by timing issues. This is done by modeling the verification target as a state machine and exhaustively exploring it.

In the beginning of the process, *model checkers* (i.e., tools for model checking) execute a transition that is executable in an initial state, and obtain the next state. Model checkers iterate this execution of a transition until the state has no more transitions. If a state has multiple transitions, model checkers branch their exploration and execute all of them in order. Typically, model checkers explore the *state space* (i.e., the space of the state machine) by running a DFS algorithm.

B. Symbolic Execution

Symbolic execution [7] is a method that executes a program with symbolically expressed values for inputs, outputs, and variables instead of using concrete values. In symbolic execution, when a program reaches a branch (e.g., `if flag == true then ... else ...`), it executes each branch, memorizing the constraints of the branch (i.e. `flag == true` for `then` branch and `flag == false` for `else` branch). The conjunction of all memorized constraints by the end of the program path is just an *executional constraint* of it. If such input values that satisfy the executional constraint of a program path exist, it is possible to execute the program path. Constraint solvers, which are tools for quickly solving constraint expressions, have advanced remarkably in recent years, so the input values that satisfy the executional constraint can be calculated efficiently with them.

III. Related Work

NICE [5] is an state-of-the art model checking tool for OpenFlow networks. To reduce state space, NICE calculates packet content in a symbolic execution in which an OpenFlow network model executes transitions along each different path, and NICE explores the state space using concrete packets created based on the calculation. Also, NICE uses heuristic search strategies for OpenFlow networks. However, NICE does not use any technique to omit redundant explorations based on the equivalency of state space (except what is clearly specified as "experimental"). As a result, NICE can detect bugs quickly, but if it has to explore the entire state space, NICE may take a long time to complete the model checking.

From a verification point of view, some paths do not need to be explored. For example, when two different hosts both send packets to two different switches, exchanging the order of these two transitions (i.e., the events of a network modeled as a state machine such as sending or receiving packets) will not result in different network behavior. Even if such paths where only the order of these transitions is different are explored, the verification result will not change. Therefore, exploring only one of these paths is sufficient for the verification. We say "two transitions are independent" or "two transitions have no dependency" if exchanging the order of these transitions does not lead to another result. On the other hand, we say "two transitions are dependent" or "two transitions have dependency" if another result is produced. DPOR [6] analyzes dependencies between each two transitions on a path after the exploration of the path, and creates a backtrack point, which marks state space that should be explored, on a state where the first of the two transitions was executed only if they are dependent. When DPOR explores state space after backtrack points, it changes the order of the two dependent transitions from prior exploration. In this way, DPOR explores only state space that should be explored and omits exploration of paths where only the order of independent transitions is different. As a result, DPOR makes model checking more efficient.

Model checking with DPOR does not stop the exploration even if it reaches a "*visited state*" (i.e., a state reached in a past exploration), since it cannot rightly analyze dependencies between a transition in an omitted space of the exploration and another on an explored path if it simply prunes the exploration after the visited state. Therefore, some inefficient cases of model checking exist because of the repeated exploration of the same state space. To address this problem, SDPOR [8] constructs a graph of the transition history of past exploration. If SDPOR reaches a visited state, it stops the exploration after the state and analyzes dependencies with the graph. As a result, SDPOR rightly calculates state space that should be explored and prunes redundant exploration.

DPOR and SDPOR are used for model checking of multi-thread programs. In contrast, DPOR-DS [9] is used for model checking of distributed systems. The basic approach used to prune the exploration is the same as in DPOR. DPOR-DS

changes the way of creating backtrack points in order to adapt DPOR to verification models of distributed systems. DPOR-DS defines the "happens-before relation" for the distributed system model differently from DPOR, and uses the results of analysis of this happens-before relation to determine the need for backtracking. The happens-before relation is the relation between two transitions that occur in the same order in any case in the model. For example, a transition of sending packet "*p*" always happens before another transition of receiving packet "*p.*" The happens-before relation is the order relation that is always satisfied in the model by its causal association, as in this example. DPOR-DS analyzes the happens-before relation of any two transitions on a path in addition to the dependency relation, and does not create a backtrack point if a happens-before relation exists between two transitions, even if a dependency relation exists between them. However, DPOR-DS does not prune explorations after visited states, whereas SDPOR does.

Header Space Analysis (HSA) [10] is a static-analysis technique for networks. HSA deals with a *L*-bit packet header as *L*-dimensional space, and models all processes of routers and middleboxes as *box transfer function*s, which transform subspaces of the *L*-dimensional space to other subspaces. We can detect network problems such as reachability failure or forwarding loops with the box transfer functions. HSA cannot detect problems occurring in the network where configurations of network devices dynamically change, whereas our method can detect these problems.

IV. Our OpenFlow Verification Method

A. Features of Our Method

Our method uses model checking to verify the correctness of OpenFlow programs and has two features as follows: (i) a symbolic state transition model to reduce the state-explosion problem caused by massive numbers of packet patterns and (ii) SDPOR-DS to reduce the state-explosion problem caused by massive numbers of orders of network events. We discuss the details of these features in this section.

B. OpenFlow Network Model for Verification

We model an OpenFlow network composed of hosts, OpenFlow switches, and OpenFlow controllers. This network model executes a transition by sending and receiving packets and OpenFlow messages by these hosts, switches, and controllers (collectively called "node(s)"). We describe the modeling of each node below.

We create a host model that only sends and receives packets. Actually, hosts can behave almost arbitrarily in a network, but it is hard to define and verify arbitrary behaviors. Thus, we model hosts as a simple transition system. The host model can execute two kinds of transitions as follows: (i) "*sending a packet*" and (ii) "*receiving a packet.*"

We model switches as a transition system that processes packets based on the switch's flow table and carries out actions based on the contents of an OpenFlow message sent from the controller. The flow table of a switch represents a state of the switch, and so a change of a switch's flow table indicates a change of its state. The switch model can execute two kinds of transitions as follows: (i) "*processing a packet*": A switch receives a packet and then applies a flow entry to it or sends a *packet-in* message to the controller, and (ii) "*processing an OpenFlow message*": A switch receives an OpenFlow message sent from the controller and carries out actions based on its contents.

We model the controller as a transition system that processes *packet-in* messages sent from switches. When the controller receives a *packet-in* message, it executes a *packet-in* handler defined in a program installed in it. Global variables of the program represent a state of the controller, so a change of a controller's global variables indicates a change of the controller's state. This controller's transition system can execute one transition called "*processing packet-in.*"

C. Symbolic Execution of OpenFlow Network Model

In our method, the contents of packets (e.g., IP address, MAC address) are expressed symbolically to reduce the state-explosion problem occurring in model checking of OpenFlow networks, which are difficult to test with existing tools. We reduce the state space with the concept of symbolic execution, and stop exploring any non-existent executional path in the real network by analyzing constraints with solvers.

We explain further with the example shown in Fig. 1. We assume here that *host A* sends packet "*p*" to *host B*, and that *switch 1* had already installed flow entries to forward packets to *host A* or *switch 2* according to the packets' destination MAC address. Our method deals with packet contents symbolically with constraint expressions as in symbolic execution. The constraints limit the results of events in networks and also limit packet contents. In this example, because *host A* sends packet "*p*" to *host B*, the source MAC and IP address are those of *host A,* and the destination MAC and IP address are those of *host B*. Therefore, our method elicits the following four constraint expressions:

p.Src_MAC_Address == hostA.MAC_Address … (i)
p.Dst_MAC_Address == hostB.MAC_Address … (ii)
p.Src_IP_Address == hostA.IP_Address … (iii)
p.Dst_IP_Address == hostB.IP_Address … (iv)

When *switch 1* receives packet "*p*," it is handled according to the flow table of *switch 1*. However, our method does not specify which flow entry is applied because "*p*" does not have a concrete value. Thus, our method branches out taking into account all possibilities of actions of *switch 1*. In this example, *switch 1* installed two flow entries; one is to forward packets to *host A*, and the other is to forward packets to *switch 2*. Thus, the following three branches occur; the first one is where *switch 1* applies the former flow entry, the second one is where *switch 1* applies the latter flow entry, and the third one is where *switch 1* sends *packet-out* message to the controller without applying any flow entry. By dealing with packet contents symbolically and branching the exploration in this manner, we do not have to distinguish states of which the details are insignificantly different with such branches, so our method can reduce the number of states that are explored.

Furthermore, our method can omit the exploration of branches that do not actually exist by solving the constraints. In this example, the first and third branches do not actually

occur. For instance, our method elicits the following constraint expression from the applied flow entry in the first branch:

p.Dst_MAC_Address == hostA.MAC_Address … (v)

Constraints (ii) and (v) are not satisfied simultaneously (except in odd networks where some hosts may have the same MAC address). Constraint solvers can solve these expressions quickly. Our method omits needless exploration with them, so it exhaustively and efficiently explores paths that can be executed in a real network without any overlapping state space.

D. SDPOR-DS

We introduce SDPOR-DS, which is an approach to reduce the state-explosion problem through massive ordering of state transitions and pruning the exploration of state space.

1) Overview of SDPOR-DS

SDPOR-DS prunes redundant exploration of model checking of distributed systems. Plainly speaking, SDPOR-DS is "DPOR-DS that prunes exploration after visited states." It is based on DPOR-DS because it aims to verify distributed systems such as OpenFlow networks. Furthermore, SDPOR-DS stops exploring a path if it reaches a visited state, which is the same idea as SDPOR. However, since DPOR-DS uses happens-before relations in addition to dependency relations to determine whether to explore state space, the graph SDPOR uses is not sufficient because the graph only contains the history with which SDPOR can analyze dependency relations. In SDPOR-DS, the graph contains the history with which it can analyze happens-before relations also.

E. Details of SDPOR-DS

1) Definitions

Definition 1 (Dependency Relation). *A dependency relation exists between transitions t_1 and t_2 if and only if a processing node of t_1 is the same as t_2 and either of the following is satisfied: (i) the state of a processing node changed in either transition t_1 or t_2, or (ii) the destination of any packet or OpenFlow message sent in transition t_1 is the same as that in t_2.*

Definition 2 (Happens-Before Relation). *We denote $t_1 \rightarrow t_2$ when a happens-before relation exists between transitions t_1 and t_2 (t_1 always occurs before t_2). A happens-before relation exists between transitions t_1 and t_2 if and only if either of the following is satisfied: (i) a packet or OpenFlow message sent in transition t_1 is the same as one received in transition t_2, or (ii) $t_1 \rightarrow t_3$ and $t_3 \rightarrow t_2$.*

2) Algorithm of SDPOR-DS

We describe here the algorithm of SDPOR-DS (Fig. 2). First, SDPOR-DS explores an arbitrary path (Line 3). Then, SDPOR-DS analyzes the dependency and happens-before relations on the path and identifies states that the exploration should backtrack to (Line 5). Next, the exploration restarts from the deepest backtrack point (Line 8). SDPOR-DS iterates this process until there is no backtrack point (Lines 4, 7).

In `explore_path` (Fig. 2, Line 3), SDPOR-DS explores an arbitrary path of a verification model of an OpenFlow network. Transitions are executed from an initial state step-by-step until it reaches a state as follows: (i) it has no transition that can be executed, or (ii) it was already visited in a past exploration, or (iii) it does not actually exist. Since constraint expressions of a state that does not exist cannot be satisfied, we can easily check whether a state exists using any constraint solver. In a real OpenFlow network, it never goes into such a state, so SDPOR-DS does not explore the space after a state.

In the exploration process, a transition sequence of a path that is therein explored and a graph of the transition history of the entire exploration space are maintained and updated. They include what is necessary to analyze the dependency and happens-before relations such as packets that are sent and received, or the destination of the packets described in 1). After any transition, SDPOR-DS adds an element to the transition sequence and a node to the graph of the transition history. Adding an element to the transition sequence consists only of adding an element such as a list structure. Adding a node to the graph consists of adding a node n_1 representing the latest transition t_1 and connecting n_1 and another node n_2 representing a transition t_2 just before t_1 with a directed edge (n_2 to n_1). These nodes are maintained as being related to states where transitions that the nodes represent are executed. When SDPOR-DS reaches a visited state, it connects nodes n_i, representing transitions that can be executed from the visited state, and node n_1, representing the latest transition t_1 with directed edges (n_1 to each n_i). After the exploration of one path, SDPOR-DS analyzes the dependency and happens-before relations between each transition on the path with the transition sequence and the graph of the transition history.

The analysis of the dependency and happens-before relations in SDPOR-DS has two phases: a phase using only the transition sequence (Fig. 3, Line 3) and a phase using the transition sequence and the graph of the transition history (Fig. 3, Lines 4-6). In the former phase, SDPOR-DS takes any two transitions (Fig. 4, Lines 2, 4) and calculates whether the dependency and happens-before relations exist between the two transitions (Fig. 4, Line 6). If the dependency relation exists and the happens-before relation does not exist, SDPOR-DS makes a backtrack point (Fig. 4, Line 7) to explore another path where the order of the two transitions switches. In the latter phase, SDPOR-DS takes nodes representing transitions that were executed on previous paths after the visited state that SDPOR-DS reached in the latest exploration (Fig. 5, Lines 11-13), and calculates whether the dependency and happens-before relations exist between any transition in the transition sequence and another transition taken from the graph of the transition history (Fig. 5, Line 7). Just as in the former phase, if the dependency relation exists, and the happens-before relation does not exist, SDPOR-DS creates a backtrack point (Fig. 5, Line 8). If SDPOR-DS did not reach any visited state in the latest exploration, the latter phase is skipped since past transitions are not executed on the latest path.

In SDPOR-DS, the graph of the transition history stores all transitions that were executed in past explorations, so all transition sequences from any states can be obtained with this graph. Therefore, there is no need to explore the same space after any visited state while executing the same transitions again in order to analyze the dependency and happens-before relations. As a result, SDPOR-DS can omit explorations after visited states, which is a great advantage in terms of efficiency.

```
def sdpor_ds(initial_state)
  branch = -1
  path, history_node = explore_path(empty_path, initial_state)
  loop {
    path = analyze_relation(path, branch, history_node)
    branch = path.get_branch
    return true if last_branch < 0
    path, history_node = explore_branch(path, branch)
  }
end
```

Fig. 2. Pseudo code of top part of SDPOR-DS.

```
def analyze_relation(path, branch, history_node)
  path_len = path.length
  path = analyze_with_path(path, branch)
  history_node.children.each { |e|
    path = analyze_with_history(path_len, path[0...path_len], e)
  }
  return path[0...path_len]
end
```

Fig. 3. Pseudo code of start of relation analysis in SDPOR-DS.

```
def analyze_with_path(path, branch)
  path.each_with_index { |e1, i|
    s = max(i + 1, branch)
    path[s..-1].each { |e2|
      hb_list = e2.hb_list
      if e1 and e2 are dependent && hb_list not include e1
        e1.add_branch(e2.transition)
      end
    }
  }
  return path
end
```

Fig. 4. Initial phase of relation analysis in SDPOR-DS.

V. PERFORMANCE EVALUATION

We developed a prototype for the evaluation. It takes a network topology file of Trema [2] as input and automatically verifies whether a controller program causes forwarding loops and black holes of packets on the topology (These are built-in properties that our prototype verifies. It has the extendability for other properties that a user wants to verify additionally). We defined and implemented models of each network node in the prototype as built-in. The constraint solver we used was Yices [11]. All of our experiments were conducted on a machine set up as follows: OS: Ubuntu 12.04.3 LTS; CPU: Intel(R) Xeon(R) X5690 @ 3.47 GHz * 2; memory: 96 GB.

The controller program was "*MAC-learning switch*." With this program, the controller that receives a *packet-in* message registers the pair of a source MAC address and a receiving port number of a packet contained in the *packet-in* message. Then, if the pair of the destination MAC address and a port number of this packet is already registered in the controller, it sends *flow-mod* and *packet-out* messages to a switch that received the packet in order to prompt the switch to send the packet to the known destination. If not, the controller sends a *packet-out* message to the switch in order to make it flood the packet from all of its ports. The topology of this experimental example is shown in Fig. 1. We measured the prototype's performance in verifying whether any problem happens in this example when *host A* sent some packets to *host B* and *host B* replied to *host A*. The results are given in TABLE I and II.

```
def analyze_ with_history(path_length, path, history_node)
  path.add_path_element(history_node)
  path_len = path.length
  path[0...path_len].each_with_index { |e1, i|
    e2 = path[-1]
    hb_list = e2.hb_list
    if e1 and e2 are dependent && hb_list not include e1.tr
      e1.add_branche(e2.transition)
    end
  }
  history_node.children.each { |e|
    path = analyze_ with_history(path_len, path[0...path_len], e)
  }
  return path[0...path_length]
end
```

Fig. 5. Latter phase of relation analysis of SDPOR-DS.

First, we compared the results of SDPOR-DS and DFS. For the model where *host A* sends four packets, SDPOR-DS reduced the number of states and the execution time to 12% of that compared with DFS. Furthermore, the larger the scale of a model was, the greater the reduction rate was. This means that SDPOR-DS is scalable. With the model where *host A* sends five packets, we could not strictly compare SDPOR-DS and DFS because DFS did not finish its exploration within five days. However, we can estimate that SDPOR-DS takes less than 10% of the execution time compared with DFS.

Next, we compare the results of our method and NICE. For the model where *host A* sends five packets, our method reduced the execution time to 31% compared with NICE. In our method, states that do not actually exist are created once and checked whether they exist. If not, the remaining steps of the exploration process such as dumping and loading of states are skipped. Consequently, the cost of exploring a state that does not exist is low compared with existing ones. Therefore, although many states followed by space that is not explored are created in our method internally, we expect that the exploration cost of our method is correlated with the number of states that actually exist. The rate of existing states among the total states is around 25%. For example, the number of existing states in an example where *host A* sends five packets to *host B* is 2,674,776. This is much smaller than the number of states with NICE, so this is thought to be the main reason why our method takes less time to execute than NICE.

As a result, our method was very fast even when compared with NICE, so we made sure of its high efficiency.

VI. BUG DETECTION

In this section, we explain an example of bugs that our prototype can detect from OpenFlow networks.

TABLE I. COMPARISON OF NUMBER OF EXPLORED STATES.

# packets	SDPOR-DS	DFS	NICE
2	989	2,311	510
3	19,801	89,931	14,340
4	426,970	3,432,569	353,150
5	9,788,689	N/A	7,987,806

TABLE II. COMPARISON OF EXECUTION TIME [sec.].

# packets	SDPOR-DS	DFS	NICE
2	1.37	2.93	1.51
3	21.15	113.65	59.07
4	628.96	5528.53	2259.03
5	22093.44	N/A	71156.64

With the MAC-learning switch program described in section V, the controller sends a *packet-out* message that contains a *FLOOD* action operation in order to prompt a switch to forward the packet(s) using Spanning Tree Protocol (STP). We assume here that the controller program includes a bug that results in the controller sending *packet-out* that contains an *ALL* action operation instead of a *FLOOD* action operation. In *ALL* action, a switch forwards the packets from all of its ports except the port from which the switch received the packet. Therefore, if any cycle exists in the topology, a broadcast storm occurs. Our prototype can detect problems caused by such bugs. For example, if our prototype executed model checking of an OpenFlow network shown in Fig. 6 that was controlled by a program including the bug described above, our prototype would display the following message.

```
The forwarding loop below was detected.
  sw1@1 -> sw3@3 -> sw2@3 -> sw1@2 -> sw3@3
```

The packet forwarding route where a broadcast storm occurs is shown in the second line. Switch names come before the `@` symbol, and the physical port numbers come after `@`.

As described above, even simple controller programs such as MAC-learning switch can include bugs that cause problems that have an impact on the entire network. With our method, we can verify whether the possibility of such a problem exists in a network, and if there is such a possibility, we can get a concrete idea of where the problem might occur.

VII. CONCLUSION

We proposed a new model-checking method of OpenFlow networks that tackles the state-explosion problem with SDPOR-DS and a symbolic state transition model. Further, we developed a prototype of our method and estimated its performance. It required only 12% of the execution time of naïve model checking with DFS for the model where a host sends four packets and another host receives the packets and replies to them. Furthermore, we can estimate that our method takes only less than 10% of the execution time compared with DFS for the model where a host sends five packets and another host receives the packets and replies to them. We also compared our method with NICE, a state-of-the-art tool for model checking of OpenFlow networks, and we confirmed that our method took only 31% of the execution time of NICE. In conclusion, our method can efficiently execute model checking for exhaustive exploration. We note that our method is not sound or complete, but we believe that our method is useful to detect bugs in OpenFlow networks effectively.

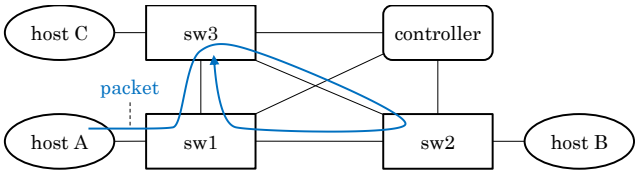

Fig. 6. A topology that has a cycle. The packet route between the controller and any switch is omitted.

In the future, we will improve the scalability of our method to apply it to actual networks. So we will introduce some heuristics based on domain knowledge of OpenFlow (e.g., FLOW-IR proposed in [5]) to the exploration of SDPOR-DS.

ACKNOWLEDGMENT

We would like to thank Prof. Hagiya at the Univ. of Tokyo, Prof. Tanabe at NII (National Institute of Informatics), and the members of Prof. Hagiya's lab for their helpful comments.

REFERENCES

[1] N. McKeown et al., "Openflow: enabling innovation in campus networks," SIGCOMM Comput. Commun. Rev., vol. 38, no. 2, pp. 69-74, Mar. 2008.

[2] H. Shimonishi et al., "Programmable Network Using OpenFlow for Network Researches and Experiments," in Proceedings of 6th International Conference on Mobile Computing and Ubiquitous Networking (ICMU 2012), 2012, pp. 164-171.

[3] N. Gude et al., "Nox: Towards an operating system for networks," SIGCOMM Comput. Commun. Rev., vol. 38, no. 3, pp. 105-110, Jul. 2008.

[4] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled, "Model checking." Cambridge, MA, USA: MIT Press, 1999.

[5] M. Canini, D. Venzano, P. Peresini, D. Kostic, and J. Rexford, "A nice way to test openflow applications," in Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 127-140.

[6] C. Flanagan and P. Godefroid, "Dynamic partial-order reduction for model checking software," SIGPLAN Not., vol. 40, no. 1, pp. 110-121, Jan. 2005.

[7] J. C. King, "Symbolic execution and program testing," Commun. ACM, vol. 19, no. 7, pp. 385-394, Jul. 1976.

[8] Y. Yang, X. Chen, G. Gopalakrishnan, and R. M. Kirby, "Efficient stateful dynamic partial order reduction," in Proceedings of the 15th International Workshop on Model Checking Software, ser. SPIN'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 288-305.

[9] M. Yabandeh and D. Kostic, "DPOR-DS: Dynamic Partial Order Reduction in Distributed Systems," Tech. Rep., 2009. [Online]. Available: http://infoscience.epfl.ch/record/139173/files/paper_2.pdf

[10] P. Kazemian, G. Varghese, and N. McKeown, "Headerspace analysis: Static checking for networks," in Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, ser. NSDI'12. Berkeley, CA, USA: USENIX Association, 2012, pp. 113-126.

[11] B. Dutertre and L. de Moura, "The Yices SMT solver," Aug. 2006. [Online]. Available: http://yices.csl.sri.com/tool-paper.pdf