Formal Modeling and Verification of Software-Defined Networking with Multiple Controllers

Miyoung Kang and Jin-Young Choi(✉)

Center for Information Security Technologies,
Korea University, Seoul 02841, Korea
{mykang, choi}@formal.korea.ac.kr

Abstract. Traditional SDN has one controller, but more recent SDN approaches use multiple controllers on one network. However, the multiple controllers need to be synchronized with each other in order to guarantee a consistent network view, and complicated control management and additional control overhead are required. To overcome these limitations, Kandoo [5] has been proposed in which a root controller manages multiple unsynchronized local controllers. However, in this approach, loops can form between the local controllers because they manage different topologies. We propose a method for modeling a hierarchical design to detect loops in the topology and prevent them from occurring using UPPAAL model checker. In addition, the properties of multiple controllers are defined and verified based UPPAAL framework. In particular, we verify the following properties in a multiple controller: (1) elephant flows go through the root controller, (2) all flows go through the switch that is required to maintain security, and (3) they avoid unnecessary switches for energy efficiency.

Keywords: SDN · Formal modeling · Formal verification · UPPAAL

1 Introduction

Software-defined networking (SDN) [1] and network function virtualization (NFV) are the core technologies for 5G [2]. SDN is used to connect networks of virtual machines (VMs) in 5G. The 5G core network is deployed in a distributed horizontal cloud form. Horizontal distributed cloud based on cloud infrastructure by core network functions to separate control and data transfer functions using SDN and NFV. Each network function can efficiently cope with explosive increases in traffic by appropriately distributing control functions to the central cloud and data transfer functions to the edge cloud. However, in a 5G network using SDN and NFV as core technologies, even if the independent VNFs do not cause any errors in the central cloud when various applications are run, collisions can occur due to rule conflict in the edge cloud. Because this can cause errors across the entire 5G network, verification is necessary.

Our goal is to suggest a formal verification method to ensure the safety and consistency of multiple controllers in SDN. SDN is a technology that separates the network device control component from the data transfer component using open interfaces, such as the OpenFlow protocol. Through the SDN controller, which deals with the control

H. Gao et al. (Eds.): TridentCom 2019, LNICST 309, pp. 81–94, 2020.
https://doi.org/10.1007/978-3-030-43215-7_6

component, forwarding and packet processing rules are determined, and forwarding rules are transferred to the lower SDN switches.

The controller plays an important role in the traffic transfer process. Unfortunately, when network traffic increases rapidly, a single controller cannot handle all of the flow requests due to a limited controller capacity. If a single controller fails, the switch will not be able to plan the routing of newly arrived packets, which will affect communication and applications on the network. As a result, a new modern controller design has been proposed based on multiple controllers.

A flat design [3, 4] for multiple controllers extends the functionality of the control plane, but it requires complex controller management and additional control overhead because the controllers must communicate with each other to ensure a consistent network. To solve this problem, a hierarchical design has been suggested. This typically uses a two-tiered controller system consisting of local controllers and a root controller. A local controller manages switches locally and runs local control applications, while the root controller manages the local controllers and maintains the global network. Kandoo [5] is a typical hierarchical controller structure, in which the root controller communicates with the local controllers to obtain domain information but the local controllers do not communicate with each other.

However, while the root controller manages each local controller, it does not allow communication between local controllers. Therefore, loops can form within a local controller. We propose a method for formal modeling a hierarchical design that identifies loops in the topology and detects them using simulation in the UPPAAL model checker [6]. The properties of the multiple controllers are also defined and verified based this design.

In this paper, we propose a formal modeling and verification framework of three properties:

- elephant flows going through the root controller that is necessary to verify whether the flow has reached the root controller.
- all flows passing the switch required to maintain security that the flows must also be routed to a switch that performs a firewall function for security reasons.
- all flows avoiding unnecessary switches in order to improve energy efficiency. Many users employ data centers during the day. However, the number of users decreases after 10 o'clock in the evening. Therefore, instead of using all of the switches when there are fewer users, the number of switches can be reduced by selecting an optimal path.

This paper is organized as follows. Section 2 introduces Software-Defined Networking, multiple controllers in SDN and UPPAAL Framework. Section 3 presents the formal modeling of a Hierarchical design for SDN controllers. Section 4 addresses the formal verification of three properties. Section 5 reviews related literature. We conclude the paper in Sect. 6.

2 Background

2.1 Software-Defined Networking

SDN originated with OpenFlow [7], which was developed as a protocol for future internet infrastructure control technology [8]. However, it evolved into the SDN concept centering on the Open Networking Foundation (ONF) [9], which was established in 2011. It is now used as the core technology for 5G networks.

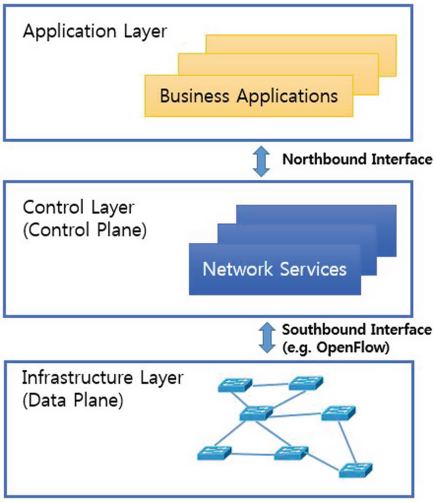

Fig. 1. The three-layered SDN architecture

Figure 1 shows the SDN framework, which consists of three layers: the application layer, the control layer, and the infrastructure layer. The application layer includes network applications that can introduce new network features such as security and manageability, provide forwarding schemes, or assist the control layer in the network configuration. The application layer can provide appropriate guidance for the control layer by obtaining an abstracted global view of the network from the controllers. Examples of network applications include network management and traffic engineering, load balancing for application servers, security and network access control, network testing, debugging and verification, inter-domain routing, and network virtualization. The interface between the application layer and the control layer is known as the northbound interface.

In the lower layer, the control plane is found. This is involved in the programming and management of the forwarding plane. In order to achieve this, it uses information provided by the forwarding plane and defines network operations and routing. It consists of one or more software controllers communicating with forwarding network elements through a standardized interface known as a southbound interface. OpenFlow, one of the most commonly used southbound interfaces, primarily considers switches, while other SDN approaches consider other network elements such as routers. The lowest layer is the infrastructure layer, which is also referred to as the data plane. It comprises forwarding network elements. The forwarding plane is primarily responsible for forwarding data, in addition to local information monitoring and statistics collection.

2.2 Multiple Controllers in SDN

In this subsection, we will introduce the origin of multiple controllers in SDN using two examples and then summarize two multiple controller architectures: flat and hierarchical.

In one of the earliest SDN designs, a single controller manages the entire network. This is illustrated in Fig. 2. In this design, packets can arrive at a switch where no corresponding rule is installed in the flow table; as a result, the switch cannot forward the packets on its own. The switch then notifies the controller about the packets. The controller identifies the path for the packets and installs appropriate rules in all of the switches along that path. The packets can thus be forwarded to their destination [10].

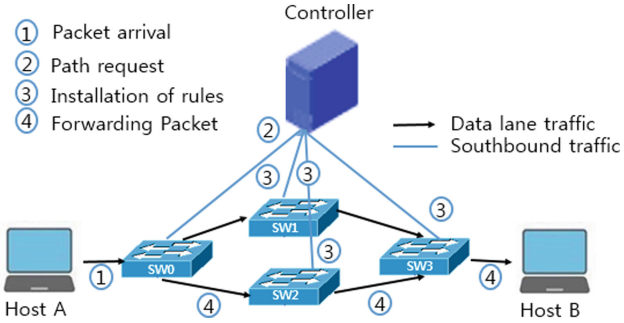

Fig. 2. Flow management of a single controller

The controller plays an important role in the traffic transfer process. Unfortunately, when the network traffic is high, a single controller cannot handle all of the flow requests due to limited controller capacity. If a single controller fails, switches will not be able to route newly arriving packets, affecting network communication and applications. As a response to this, the use of multiple controllers was introduced to SDN.

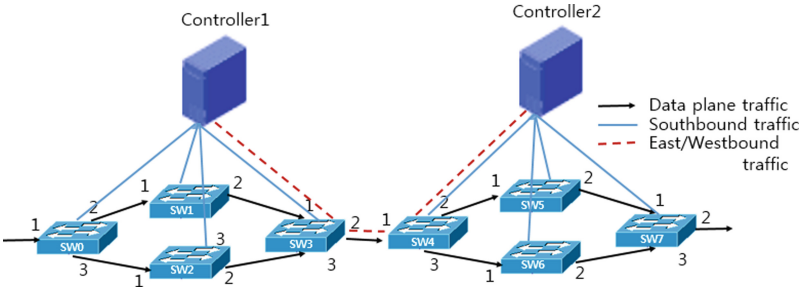

Fig. 3. Flat design for multiple controllers

As shown in Fig. 3, a flat design extends the functionality of the control plane but requires complex controller management and additional control overhead for east/westbound traffic. This is because controllers must communicate with each other to ensure a consistent network view. A hierarchical design has been proposed to solve this problem. It typically uses a two-tiered controller known as a root controller, which

manages the switches in the local domain, manages local controllers that run local control applications, and maintains a global network view. Kandoo [11] is a typical hierarchical controller structure. In Kandoo, the root controller communicates with the domain controller to obtain domain information, but the local controllers do not contact each other. Figure 4 shows the basic architecture of hierarchical design. In order to redistribute any overload of flow requests at one controller, which is a major issue for single-controller designs, the local controller sends the routing of elephant traffic to the root controller, and the root controller issues the forwarding rules.

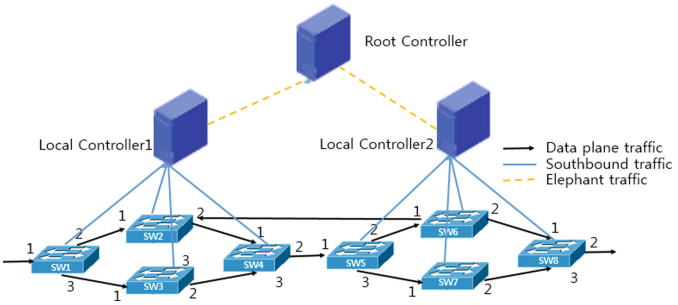

Fig. 4. Hierarchical design for multiple controllers

2.3 UPPAAL Framework

UPPAAL modeling for multiple controller consists of a root controller, local controllers, switches, and a host. Simulations can be used to confirm that the modeling operates as intended. In particular, the safety and reachability of the system can be verified. As a result of this verification process, users receive either a *satisfied* or *not satisfied* message (Fig. 5). In the paper, the model and its properties are modified through feedback and verification is run again. For details we refer the reader to [6].

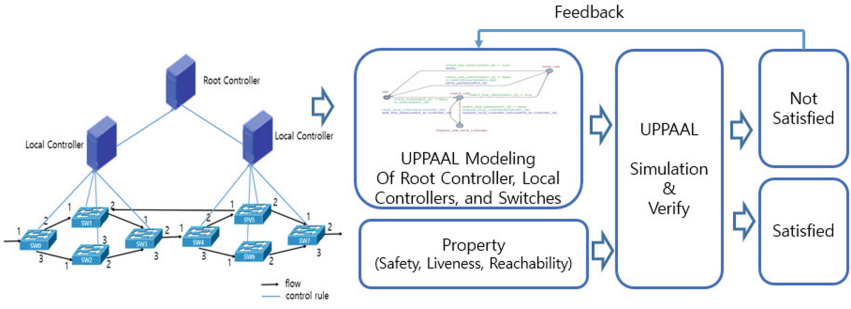

Fig. 5. UPPAAL framework

3 Modeling of a Hierarchical Design for SDN Controllers

The controllers within a flat design communicate with each other to ensure consistency, which can lead to overloads because of a lot of traffic. To improve this, a hierarchical design has been proposed, in which a logical centralized controller manages the network-wide state without the local controllers communicating with each other. Therefore, a hierarchical design requires considerably less control channel bandwidth compared with normal OpenFlow networks. However, loops can occur in a network-wide topology because the individual local controllers manage their own switches and do not share views with each other. Figure 4 shows that each local controller only manages switches that are linked to them and that only the root controller has the entire logical view. A loop can be created (e.g., Switch 5 -> Switch 1 -> Switch 3 -> Switch 4 -> Switch 5) if the local controller sends a rule that includes a route (Switch 5 -> Switch 1) at a certain time.

Our framework can detect the occurrence of a loop in advance and modify the topology to avoid it. Our framework consists of a host, switches, local controllers, and a root controller. Packets originate from the host. When a flow arrives at the switch, it looks for a matching entry in the flow table of the switch. If a matching entry is found, the actions associated with the specific flow entry are executed. If no match is found, the flow may be forwarded to the local controller. The local controller sends the rule to the switch according to the inquiry. If an elephant flow, which has 1 M pkt-in per second, enters a switch, the flow requests forwarding rules from the root controller via a local controller, and then the root controller determines the action for the elephant flow and sends it to the local controller. The local controller then issues a forwarding rule to the inquiring switch. This process decreases the load on the local and root controllers, preventing the overloading that occurs in a flat design in order to maintain consistency between the controllers.

3.1 Host Modeling

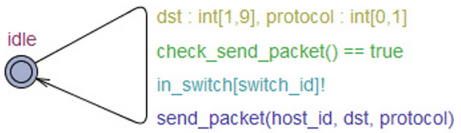

Fig. 6. Model of a host

The host is the starting point for a flow. Each packet in the flow starts with the address of the origin and the address of the destination. The origin is labeled 0 and the destination is randomly labeled between 1 and 9 (`dst: int [1,9]`). `Protocol: int[0,1]` sets a normal flow as 0 and an elephant flow as 1 (Fig. 6).

When the host has `check_send_packet () == true`, it sends a packet. When this packet leaves the topology, it sends the next packet. `in_switch [switch_id]!` is a channel that sends packets to a connected switch when the host creates it. Packets are sent to the switch that corresponds to the `switch_id` by synchronizing with the switch's `in_switch [switch_id]?`. Channel 0 is synchronized first, and the next switch is connected when the host is activated.

```
typedef struct {
 int id;
 int src;
 int dst;
 int protocol;
} PACKET;
```

```
typedef struct {
 int switch_id;
 int src;
 int dst;
 int protocol;
 int action;
}FLOWTABLE;
```

For example, "`host = host (0,0);`" shows that a host with the host ID 0 and the switch ID 0 has been created.

Packets are abstracted and contain information about the packet number, source address, destination address, and protocol. The flow table is abstracted to have a switch ID, a source address, destination address, protocol, and an action.

3.2 Switch Modeling

The switch begins in an idle location; after a packet arrives at the switch, it matches the rule to the flow entry of the flow table to determine the forwarding rule for the packet. When the channel is synchronized with `in_switch[switch_id]?`, it first inquires whether the rule is in the flow table. If there is no rule, it queries the local controller. The controller provides the rule by adding a new flow entry to the switch's flow table (Fig. 7).

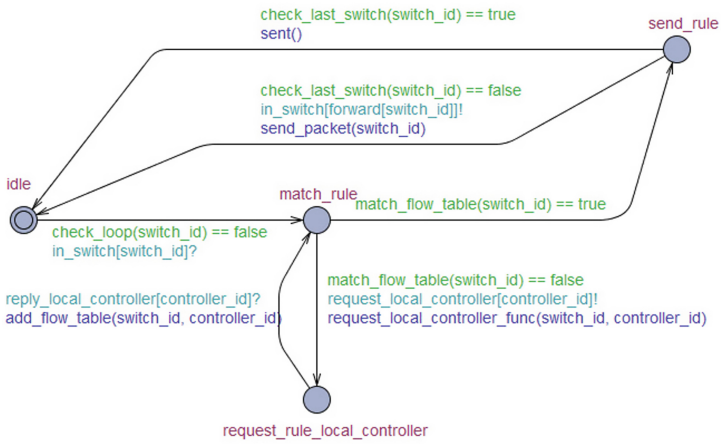

Fig. 7. Model of a switch

If the switch checks the flow table and finds a matching field, it looks at the matching flow entry's action and sends the packet to `send_packet(switch_id)` with the next switch. `check_last_switch(switch_id) == false` will send it to the next switch at random if it is not the last switch. For `check_last_switch (switch_id) == true`, no more switches are connected, and the packet leaves the topology.

`Match_flow_table(switch_id) == false` means that, if there is no matching flow entry in the flow table, `request_local_controller [controller_id]!` will synchronize with the local controller. `Request_local_controller_func(switch_id, controller_id)` will query the local controller connected to the switch via `switch_id` and `controller_id`. `add_flow_table(switch_id,controller_id)` adds a forwarding rule to the flow table of the switch through the channel of `reply_local_controller [controller_id]?` in `request_rule_local_controller` location. If `match_flow_table (switch_id) == true`, `send_packet (switch_id)` is sent according to the action of the rule in the flow table, and the switch becomes idle again.

3.3 Local Controller Modeling

If a packet does not match the flow entry of a switch's flow table, it queries the local controller for the rule. This controller has information about the switches. `sw0: int [0,1]` means that `switch 0` randomly selects `0` or `1`. `initialize_controller (sw0, sw1, sw2, sw3, sw4, sw5, sw6, sw7)` initializes the state of the switch in the controller. When a query for the forwarding rule comes through the switch channel `local_controller [controller_id]?`, whether it is an elephant flow or a normal flow is determined at `Identify_traffic` location (Fig. 8).

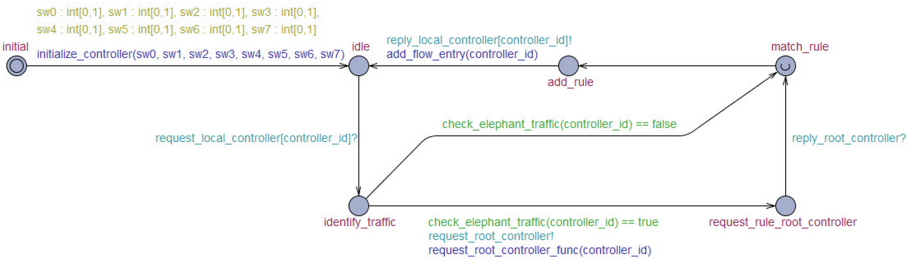

Fig. 8. Model of a local controller

If `check_elephant_traffic (controller_id) == true`, the local controller again queries the root controller for the rule because an elephant flow has arrived. Conversely, if `check_elephant_traffic (controller_id) == false`, the local controller makes an urgent transition from a `match_rule` location to an `add_rule` location, and than sands `reply_local_controller [controller_id]!` channel, and adds a rule to the flow table of the switch via

`add_flow_entry (controller_id)`. The `reply_local_controller [controller_id]?` in the switch and `reply_local_controller [controller_id]!` transition by synchronizing. Once the local controller has transferred the rule to the switch, it moves to an `idle` location.

3.4 Root Controller Modeling

The root controller handles the path of an elephant flow, so the overloading of the local controllers is reduced. As shown in Fig. 9, after the system starts, `initialize_controller (sw0, sw1, sw2, sw3, sw4, sw5, sw6, sw7)` initializes the state of the switch in the root controller.

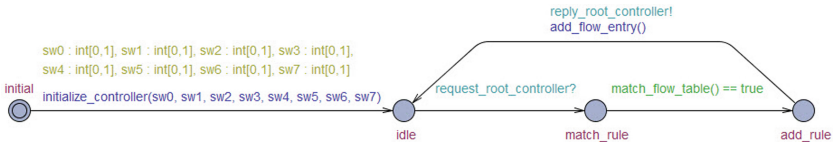

Fig. 9. Model of the root controller

When `request_root_controller?` occurs at an idle location, the root controller searches for the matching rule. If there is a matching rule, it transitions to `add_rule` location and then adds the rule to `add_flow_entry()`. In addition, `replay_root_controller!` synchronizes with the `replay_root_controller?` of the local controller and then becomes idle.

The root controller can handle elephant flows of more than 1 MB, which not only improves the performance of the system but also distributes a lot of traffic to the SDN controller.

3.5 Simulation of Hierarchical Design

After modeling the proposed hierarchical design, we run simulations. Analyzing the results, some problems in the model set-up can be observed. As can be seen in Fig. 10, a loop is created in the simulation and a deadlock occurs.

The destination of an elephant flow can be determined through the root controller. Compared to a normal controller, the root controller is 550 times more bandwidth efficient for control plane loads in an element flow detection scenario. The control plane loads are based on the number of nodes in the network [5]. Therefore, it is necessary to verify whether the flow has reached the root controller. In addition, flows must also be routed to a switch that performs a firewall function for security reasons. It is thus necessary to verify that the flow passes through this switch.

It is also important to verify that a flow does not pass through switches that it does not need to. Many users employ data centers during the day. However, the number of users decreases after 10 o'clock in the evening. Therefore, instead of using all of the switches when there are fewer users, the number of switches can be reduced by selecting an optimal path.

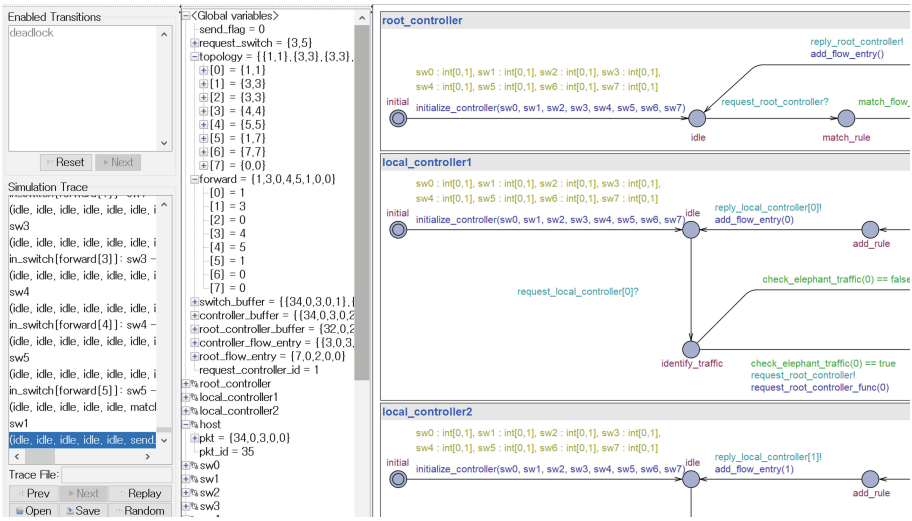

Fig. 10. Simulation of the model

The problems we found with the simulation were as follows. First, a loop occurs in the topology and this needs to be avoided in SDN topology. Second, a flow must pass though Switch 1, which functions as a firewall. However, there is a flow that does not pass Switch 1. Third, the fastest route needs to be implemented for energy efficiency. However, during modeling, even though a flow only needs to follow the route Switch 0 -> Switch 1 -> Switch 3 -> Switch 4 -> Switch 5 -> Switch 7, it also passes Switch 2 and Switch 6. We thus revise the topology of the verification framework to avoid these three problems.

```
bool check_loop(int switch_id)
{
   if (switch_buffer[switch_id].time > 10)
      return true;
   else
      return false;
}
```

A loop is found in the simulation and a deadlock occurs (Fig. 10). Therefore, we created a revised topology in which the loop does not occur. The check_loop () function determines whether a loop has occurred by checking `switch_buffer [switch_id] .time>10` and, in the improved model, no loop occurs.

In addition, the flow also passes through Switch 1, which it must do for security purposes. In addition, switches that do not need to be included in the route for energy efficiency purposes have been removed from the route.

4 Verification of the Hierarchical Design

In our study, the verification framework specifies and verifies each property with TCTL [6]. The query language in TCTL consists of path formulae and state formulae. State formulae describe individual states, whereas path formulae quantify the paths or traces of a model. Path formulae can be classified into reachability and safety [6]. UPPAAL framework that we propose verifies three properties in hierarchical controller: (1) elephant flows go through the root controller using reachability (2) all flows go through the switch that is required to maintain security using safety, and (3) they avoid unnecessary switches for energy efficiency which shows through safety.

4.1 Reachability

Reachability properties are often used when designing a model to perform sanity checks and validate the basic behavior of the model [6]. The following formulae verify that the reachability of the root controller is true:

```
E<> (switch_buffer[0].protocol == ELEPHANT_TRAFFIC) &&
        (root_controller.match_rule)
```

They ask whether a given state formula can *possibly* be satisfied by any reachable state, in particular, whether `ELEPHANT_TRAFFIC` reaches the `root_controller.match_rule` state. Figure 11 is verification results which is '`Property is satisfied`'. In other words, the elephant traffic reaches the root controller.

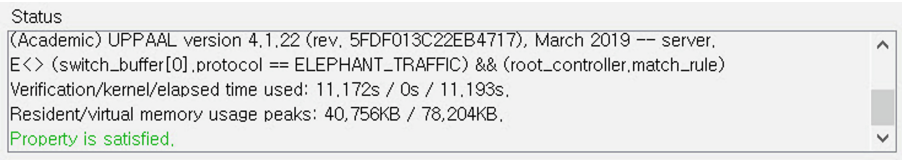

Fig. 11. Result of reachability verification

4.2 Safety

Safety is based on the concept that "something bad will never happen." In UPPAAL, this is formulated positively, e.g., something good is invariantly true. Let `P` be a state formula. We express that `P` should be true in all reachable states with the path formula `A[] p` whereas `E[] P` says that there should exist a maximal path such that `P` is always true [6].

Switch 7 must be passed through for security purposes:

```
E[] sw7.idle and sw7.match_rule and sw1.send_rule and sw1.request_rule_
local_controller
```

Packets do not pass through Switches 2 and 6 for energy efficiency purposes:

```
E[] not(sw2.idle and sw2.match_rule and sw2.send_rule and
      sw2.request_rule_local_controller and sw2.send_rule) and not
(sw6.idle and sw6.match_rule and sw6.request_rule_local_controller and
sw6.send_rule)
```

Figure 12 is verification results which is '`Property is satisfied`'. In other words, all flows avoid the Switches 2 and 6 in order to improve energy efficiency.

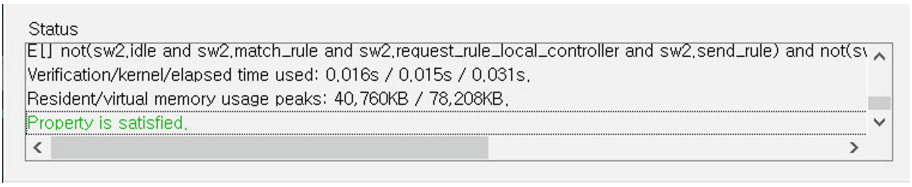

Fig. 12. Result of safety verification

5 Related Work

Other authors have employed models to verify SDN design. For example, NICE [11] is a model checking tool that uses the symbolic execution of event handlers to identify representation packets that exercise code paths on the controller. NICE detects programming errors such as no forwarding loops, no blackholes, direct paths, and no forgotten packets in testing unmodified controller programs in a NOX controller.

Frenetic [12] is a high-level programming language for OpenFlow applications running on top of NOX. Frenetic allows OpenFlow application developers to express packet processing policies at a higher level than the NOX API. Frenetic also has the network language define the formal semantics for OpenFlow rules and improves NetCore [13] by adding a compiler.

Kazemian *et al.* [14] allows the static checking of network specifications and configurations to identify important classes of failure, such as reachability failure, forwarding loops, and traffic isolation and leakage problems. A framework using formalism, referred to as header space analysis (HSA), looks at the entire packet header as a concatenation of bits. Hassel, which is a library of HSA tools, analyzes a variety of networks and protocols. The model developed by Kazemian *et al.* was the starting point for the Reitblatt *et al.* [15] model.

Reitblatt *et al.* [15] developed a formal model of OpenFlow networks and proved that consistent updates preserve a large class of properties. The formal model is made up of the notion of consistent network updates when transitioning between configurations. The model identified two distinct consistency levels, per-packet and per-flow, and presented general mechanisms for implementing the two levels in SDN using OpenFlow. The verification tool, which is an implemented prototype that reduces the overhead required to perform consistent updates, checks the correctness of the controller software.

Canini *et al.* [16] introduced a formal model describing the interaction between the data plane and a distributed control plane that consists of a collection of fault-prone controllers. In addition, Kang *et al.* [17] introduced a framework in which the consistency between a specification and its implementation is checked by dead-lock detection in the parallel composition of two different pACSR processes generated from two entities in different forms, one in the rule and the other in the OpenFlow table.

Xiao *et al.* [18] introduced the modeling and verification of SDN with multiple controllers to apply Communicating Sequential Processes (CSP). Using the model checker Process Analysis Toolkit (PAT), they verified that the models satisfied three properties: deadlock freeness, consistency, and fault tolerance. They plan to investigate other architectures, such as Kandoo.

Our study differs from the others mentioned above in that our verification and simulation identifies loops generated by the SDN in the hierarchical design of multiple controllers and receives feedback from the UPPAAL framework to remove these loops. It also verifies that elephant flows pass through the switch required to maintain security, that they do not pass through switches that are not required, and finally that they pass through the root controller.

6 Conclusion

SDN, a core technology of 5G communication, uses multiple controllers in a single network. However, synchronizing multiple controllers in order to ensure a consistent network view is problematic. Complex controller management and additional control overhead are also an issue. Therefore, a hierarchical controller structure has been proposed to manage asynchronous local controllers with many root controllers. However, when the root controller manages each local controller, loops can form between the local controllers because they manage different topologies.

We have modeled a hierarchical design that can extract loops that occur in these topologies. In addition, the three properties of the multiple controllers are defined and verified based on a simplified version of TCTL. The properties that must be validated in the multiple controllers are identified, including that elephant flows go through the root controller, pass through the security switch, and avoid unnecessary switches to improve energy efficiency. We also explained how reachability and safety can be verified.

Future research should focus on systems that automatically detect loops in a hierarchical design, and modeling and verification should be applied for the occurrence of collisions between rules in VNFs for the 5G core technologies SDN and NFV.

Acknowledgments. This research was supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2018R1A6A3A01012955) and supported by Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (2018R1A2B6009122).

References

1. KcKeown, N., et al.: OpenFlow: enabling innovation in campus networks. ACM SIGCOMM Comput. Commun. Rev. **28**, 69–74 (2008)
2. Hawilo, H., Shami, A., Mirahmadi, M., Asal, R.: NFV: state of the art, challenges and implementation in next generation mobile networks (vEPC). IEEE Netw. Mag. **28**, 18–26 (2014)
3. Dotan, D., Pinter, R.Y.: HyperFlow: an integrated visual query and dataflow language for end-user information analysis. In: IEEE Symposium Visual Languages and Human-Centric Computing (VL/HCC), pp. 27–34 (2005)
4. Shtykh, R.Y., Suzuki, T.: Distributed data stream processing with Onix. In: IEEE 4th International Conference on Big Data Cloud Computing, pp. 267–268 (2014)
5. Yegabeh, S.H., Ganjali, Y.: Kandoo: a framework for efficient and scalable offloading of control applications. In: HotSDN 2012, Helsinki, Finland (2012)
6. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-30080-9_7
7. OpenFlow Switch Specification. https://www.opennetworking.org/software-defined-standards/specifications/. Accessed 26 Aug 2019
8. SDN. https://www.opennetworking.org/sdn-definition/. Accessed 18 Jan 2019
9. ONF. https://www.opennetworking.org/. Accessed 25 July 2018
10. Braun, W., Menth, M.: Software-defined networking using OpenFlow: protocols, applications and architectural design choices. Future Internet **6**, 302–3363 (2014)
11. Canini, M., Venzano, D., Peresini, P., Kostic, D., Rexford, J.: A NICE way to test OpenFlow applications. In: NSDI (2012)
12. Foster, N., Harrison, R., Meola, M.L., Freedman, M.J., Rexford, M.J.J., Walker, D.: Frenetic: a high-level language for OpenFlow networks. In: Proceedings on PRESTO 2010 (2011)
13. Monsanto, C., Foster, N., Harrison, R., Walker, D.: A compiler and run-time system for network programming languages. In: POPL (2012)
14. Kazemian, P., Varghese, G., McKeown, N.: Header space analysis: static checking for networks. In: NSDI (2012)
15. Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., Walker, D.: Abstraction for network update. In: SIGCOMM (2012)
16. Canini, M., Kuznetsov, P., Levin, D., Schmid, S.: Distributed and robust SDN control plane for transactional network updates. In: INFOCOM 2015 (2015)
17. Kang, M., Choi, J., Kang, I., Kwak, H., Ahn, S., Shin, M.: A verification method of SDN firewall applications. IEICE Trans. Commun. **E99-B**(7), 1408–1415 (2016)
18. Xiao, L., Xiang, S., Zhu, H.: Modeling and verifying SDN with multiple controllers. In: Proceedings of SAC 2018: Symposium on Applied Computing (2018)