

A Case for Systematic Detection and Rigorous Location of SDN Control Conflicts

Lan Lin and Xin Sun

Department of Computer Science, Ball State University, Muncie, Indiana 47306

Email: {llin4, xsun6}@bsu.edu

Abstract—We take the position that the interaction of multiple software-defined networking (SDN) apps running in parallel must be thoroughly understood, and the potential conflict arising from it must be carefully prevented. We first motivate the problem with a concrete case study that shows that two SDN apps can cause a network to enter a permanent oscillation state. We then sketch a rigorous approach for systematically reasoning about the joint effect of multiple SDN apps, including whether conflicts may occur. The proposed approach models individual SDN apps as automata, and then uses standard automaton operations to compute their interactions. We further propose to apply sequence-based specification, a well-established rigorous method for software specification, to derive the automata. The key novelty and strength of our approach is its ability to reason about the joint effect of multiple SDN apps precisely (with the capability of identifying when and how conflicts may occur), proactively (prior to running the apps), and without the knowledge of the apps' implementation details.

I. INTRODUCTION

Software-defined networking (SDN) [2] is an exciting new paradigm in networking. Compared with the traditional networking infrastructure, SDN is designed to be much more open, agile, and innovation-friendly. This is achieved by the unique three-tier architecture as illustrated in Figure 1. The “controller” (middle layer), also known as network operating system, is a software program that centralizes a network's control plane. On the one hand, it manages the hardware boxes (bottom layer), known as SDN switches, by dynamically installing and updating configurations (also known as “flow rules”) to them, via the south-bound API. On the other hand, it supports and interacts with a set of SDN apps (top layer), by offering an abstraction of the network resources and states, via the north-bound API. An SDN app is a software program that runs on top of this abstraction and typically manages and optimizes a single aspect of the network, including link utilization, resiliency, energy efficiency, etc.

The SDN apps are the “brain” of an SDN network, as they implement sophisticated control logic, dictate the behavior of the network, and realize the promised benefits of SDN (such as ease of management, reduced cost, better agility, etc.). As such, over the past several years, the networking community's research and development effort has focused on building individual SDN apps, and as a result, today a myriad of SDN apps are readily available (a comprehensive review of them is presented in [9]), and marketplaces for selling and purchasing them have emerged as well (e.g., Hewlett-Packard

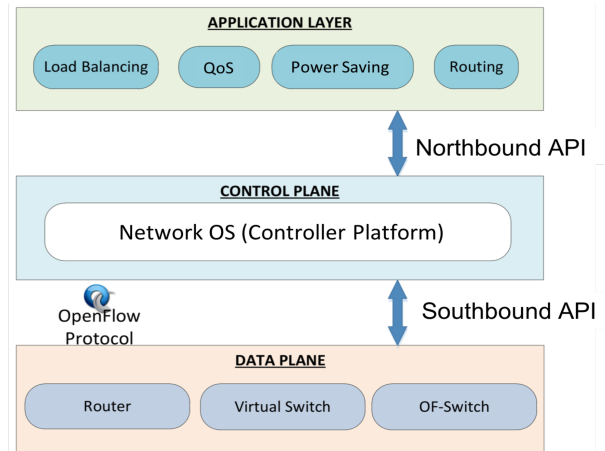


Fig. 1: The Three-Tier Architecture of SDN

App Store [1]). Now any organization that operates a software-defined network may purchase and run SDN apps based on their networking needs.

As virtually any organization has multiple networking needs, they will run multiple SDN apps, each optimizing one aspect of the network. However, relatively little research effort has been put into understanding and managing their joint interaction on top of the same network topology. In this paper, **we take the position that the interaction among multiple SDN apps must be thoroughly studied, and the potential control conflict arising from such interaction must be carefully prevented.** Control conflicts are inevitable, given each SDN app being developed by a different vendor, managing/optimizing a single aspect of the network, and assuming full control of the network. As we will illustrate with a concrete example (in Section III), such conflict can de-stabilize the network and cause unpredictable forwarding behavior.

We will sketch a fine-grained, formal approach to the systematic understanding and prevention of the control conflicts (in Section IV). Our approach has several unique advantages that we believe are critical to real-world deployment. It enables precise understanding of the interaction of multiple SDN apps, how conflicts are caused, and how they can be resolved. It allows proactive detection of any potential conflict before the apps are deployed. Finally, it only requires the observable black-box behavior of the apps as an input, and does not need any implementation details such as the source code.

II. RELATED WORK

There has been relatively few prior works on control conflicts from running multiple SDN apps, as the problem has only recently begun to receive attention from the research community. Corybantic [11] and its newer version, Athens [5], are among the first work. They take a coarse-grained, at-runtime approach to preventing control conflicts. More specifically, this approach periodically lets individual SDN apps propose their intended changes for the network state, and then lets each app evaluate all proposals from their peers, based on some pre-defined criteria or voting mechanism. In the end only one proposal will win and be implemented by the controller, and thus the network is guaranteed to be free of control conflicts. However, this approach is unable to shed light on when and how a conflict is created; moreover, by admitting only one proposal at a time, it may miss out on opportunities to simultaneously optimize multiple aspects of the network by combining multiple proposals. [6] takes a similar approach (and thus suffers from similar drawbacks), except that it seeks to combine multiple proposals to form a globally optimal configuration using an evolutionary approach. In contrast, a fine-grained approach that formally models the interaction of multiple SDN apps, such as the one we will sketch in Section IV, will not have any of those drawbacks.

Prior works [7], [8] detect forwarding policy violations (such as black holes and loops) at individual-flow level. However, their approaches cannot be used to detect SDN control conflicts, as this is a higher-level issue, not a violation of the flow policy.

III. CONTROL CONFLICTS AND NETWORK STATE OSCILLATION: A MOTIVATIONAL EXAMPLE

Generally speaking, an SDN app is developed to manage/optimize some aspect of a network, by modifying the network's state. The state of a network includes multiple variables, such as traffic load, routing paths, power state of devices, up/down state of links, etc. [15]. While some of the variables are not controlled by the network control plane (e.g., the total amount of traffic in a network is generated by the end hosts), others are, typically via configuration of the forwarding devices (e.g., the link load can be minimized by configuring the routers to run a load-balancing scheme). SDN apps control the network state by issuing commands to the controller via the northbound API (Figure 1); the controller then compiles each command down to a set of new configurations (typically in the form of flow rules) to be installed on the data plane.

To meet its optimization objective, each SDN app independently issues commands to the controller, to dictate how the network states should be changed. When multiple SDN apps are running in parallel, their commands, i.e., the ways they intend to change the network, may conflict. To demonstrate the problem in a tangible manner, below we present a concrete example. The network topology used in this example is shown in Figure 2. End hosts attach to Routers A and B. Between them there are two routing paths: one goes through Router E and the other F. The traffic load (i.e., utilization rate) on each

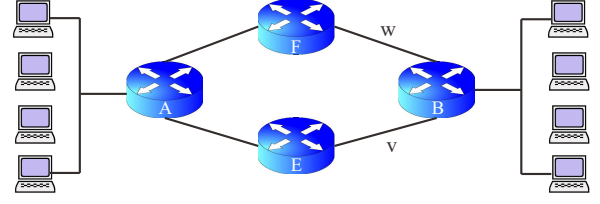


Fig. 2: The Network Topology Used in the Case Study

link is modeled as a step function that has three states: high (H), low (L), and zero (Z). Assume all devices and links have the same capacity. The network runs a *power-saving* app and a *load-balancing* app, both are commonly used in the real world. The detailed functionalities of the two apps are as follows:

- **Power-saving app:** its objective is to minimize the energy use, by selectively turning off part of the network. To realize this objective, it looks for opportunities to route all traffic to one of the two paths (without loss of generality, assuming it's always the top path), and then turn off the router on the other path. More specifically, it checks the utilization rates of the two links w and v every M seconds; when the utilization rates of both links are low or zero for two consecutive cycles, it issues a command to the controller, instructing the controller to turn off Router E and route all future packets to the top path. Afterwards, it continues to monitor the utilization rate of the top link w . Whenever the link utilization rate is high for two consecutive cycles, it issues a command to the controller, instructing the controller to turn on Router E and use both paths to route traffic.
- **Load-balancing app:** its objective is to minimize the traffic load of any link in the network. To realize this objective, it seeks to spread traffic evenly across all possible paths. More specifically, it checks the utilization rates of the two links w and v every M seconds. When one rate is high and the other is low or zero for three consecutive cycles, it issues a command to the controller, instructing the controller to route a larger percentage of incoming new flows in the next M seconds to the less utilized path. It stops issuing such commands once the utilization rates of both links become high, or both become not high (i.e., low or zero).

Now consider the following sequence of events:

- At Times M and $2M$, the utilization rates of both w and v are low; the power-saving app issues a command at Time $2M$ to route incoming flows to the top path, so that the bottom path may be turned off in the future.
- At Times $3M$, $4M$, and $5M$, as a result of the previous action by the power-saving app, w has a high rate, and v has a low rate; the load-balancing app issues a command at Time $5M$ to route more new flows to the bottom path, in an attempt to balance the loads of the two paths.
- At Times $6M$ and $7M$, as a result of the previous action by the load-balancing app, both links have a low rate. The power-saving app issues a command at Time $7M$ to

route incoming flows to the top path.

- At Times 8M, 9M, and 10M, as a result of the previous action by the power-saving app, w has a high rate, and v has a low rate; the load-balancing app issues a command at Time 10M to route more new flows to the bottom path.

Clearly, the network has entered a permanent oscillation state, and the two apps will never stop issuing commands to update the network configuration, but neither will ever meet its optimization objective.

Generally speaking, when two SDN apps, say A and B, are running in parallel, and if A always changes the network from State X to State Y, and B always changes the network state from Y to X, the network will permanently oscillate between the two states. Depending on what the differences between the two states are, this could mean constantly powering up and down devices, enabling and disabling links, routing traffic flows back and forth on different paths, etc. The problem can get much more complicated, as the conflict may be caused by more than two apps, and the network may oscillate among more than two states.

IV. SKETCH OF A RIGOROUS AND SYSTEMATIC APPROACH

In this section, we present our preliminary thoughts on a formal approach to the control conflict detection of SDN. Our approach models the SDN apps as automata, and then leverages standard automaton operations to compute how multiple SDN apps will interact and how the interaction will affect the network state. Below we describe more details of our preliminary thoughts. We leave the full investigation to a future full paper.

A. An Automaton-Based Model

We propose to use an automaton to model the working mechanism of an SDN app running on a network topology. More specifically, we follow the conventional definition of a Moore machine:

Definition 1 (Moore Machine): A Moore machine is a 6-tuple $(Q, \Sigma, \Gamma, \delta, \nu, q_0)$, where Q is a finite set of states, Σ is an input alphabet, Γ is an output alphabet, $\delta : Q \times \Sigma \rightarrow Q$ is the transition function, $\nu : Q \rightarrow \Gamma$ is the output function, and $q_0 \in Q$ is the starting state.

SDN apps can be modeled as a subset of Moore machines, of which the output alphabet includes the two special responses: 0 (the null response) and ω (the illegal response).

Definition 2 (Network Function Moore Machine): A network function Moore machine is a Moore machine $(Q, \Sigma, \Gamma, \delta, \nu, q_0)$ with $\{0, \omega\} \subset \Gamma$.

In this model, Q represents the set of software states of the SDN app (note that these are different from the states of the network); Σ represents inputs to the SDN app, which include the network states such as routing topology, link bandwidth, and traffic load information; Γ represents outputs from the SDN app, which are commands to the SDN controller that seek to change the network state; 0 and ω are special outputs representing the null output and the illegal output (i.e., output

that cannot possibly be generated by the app – this is defined for completeness purpose).

The benefit of using an automaton-based model is multi-fold, as follows:

- By precisely modeling the working mechanisms of the SDN apps, it sheds direct light on the root cause of the conflict, e.g., what apps will conflict with each other, and under what network conditions; it will also unambiguously determine the consequences of the conflict, e.g., whether the network will oscillate among multiple states, and if so, what those states are. We note that the prior approaches as we reviewed in Section II are unable to answer any of those important questions.
- To derive the automaton of an SDN app, we will only need the observable black-box behavior of the app; in particular, we do not need to know any implementation details, such as the source code. As such, this modeling approach is naturally independent of the implementation platform.
- Because the black-box behavior of an SDN app may be obtained from its user manual or by running it on a testbed, the automaton can be derived before the SDN app is deployed in the production network. Thus our approach is capable of answering what-if questions before deployment, such as how well a set of apps may work together, how well a new app may fit into a production environment, and ensure conflict-free deployment.

B. Deriving the Automaton

While the automata may be derived by hand, we believe such a manual approach will be tedious and error-prone, and will not scale well. We propose to leverage a well-established formal method for software specification, which is rigorous, constructive, and systematic. The method is termed *sequence-based specification (SBS)*¹ [10], [12]–[14]. SBS enables one to systematically discover and construct the automaton from an informal description in natural language of how the software is supposed to work (this description is termed *functional requirements* in the field of software engineering). The output of SBS is a Moore machine representing the working mechanism of the software (in our case, the SDN app). The approach has three key steps: identification of stimuli and responses, sequence enumeration, and automaton construction [10], [12]–[14].

The main innovation in adapting sequence-based specification to the SDN context is to factor in the network topology as part of its input. Topology information is needed in the model, because the interaction of SDN apps is affected by the topology – given the same set of apps, conflict may arise on one topology, but may not on another. We believe topology information can be included in the input to SBS, i.e., the informal description will specify how the SDN app works on

¹Sequence-based specification and its supporting tools, Proto_Seq [3] and REAL [4], were developed by the University of Tennessee Software Quality Research Laboratory.

a given network topology. We leave the investigation to future work.

C. Computing the Interaction of Multiple SDN Apps

Since each SDN app is modeled as an automaton, the interaction of two SDN apps can be straightforwardly modeled by the standard automaton product, which is defined below:

Definition 3 (Product Network Function Moore Machine):

Given two network function Moore machines $M_1 = (Q_1, \Sigma_1, \Gamma_1, \delta_1, \nu_1, q_{1,0})$ and $M_2 = (Q_2, \Sigma_2, \Gamma_2, \delta_2, \nu_2, q_{2,0})$, the *product* of M_1 and M_2 , denoted by $M_1 \times M_2$, is defined by $M_1 \times M_2 = (Q_1 \times Q_2, \Sigma_1 \times \Sigma_2, \Gamma_1 \times \Gamma_2, \delta, \nu, (q_{1,0}, q_{2,0}))$, where δ and ν are defined by:

$$\begin{aligned}\delta((p, q), (a, b)) &= (\delta_1(p, a), \delta_2(q, b)), \\ \nu((p, q)) &= (\nu_1(p), \nu_2(q)).\end{aligned}$$

We extend the *null* response and the *illegal* response to a product network function Moore machine by assuming $0 = (0, 0)$ and $\omega = (r_1, r_2)$ where either $r_1 = \omega$ or $r_2 = \omega$. An input $(a, b) \in \Sigma_1 \times \Sigma_2$ is *legal* if it is physically realizable; otherwise, it is *illegal*.

Note that the product operation can be extended to more operands, so to compute the interaction of more than two SDN apps, the pair-wise operation can be simply performed in a successive way. The final product Moore machine formally models the joint effect of all individual SDN apps.

D. Analysis of the Product Moore Machine

Many important properties of a network can be discovered by analyzing the product Moore machine. We are particularly interested in answering the following questions that are of strong interest to network managers:

- Will a given set of SDN apps cause any control conflict?
- When conflict is possible, will it always occur, or occur only under certain conditions (and what precisely are those conditions)?
- What are the consequences of the conflict, e.g., will the network oscillate among multiple states, and if so, what are those states?
- Besides the conflict, what is the joint effect of running the SDN apps in parallel? How will the network behave?

Through our preliminary investigation, we discovered that the key to answering the above questions is a critical concept that we termed *safe operational region*. Informally, this concept models the set of network states that will eventually lead to a stabilized network, which is when all SDN apps have met their optimization objectives and none intends to modify the network state any more. Note that the network states are included as part of the inputs to the Moore machines (Section IV-A). Thus we believe one will be able to compute the safe operational region by analyzing the product Moore machine. We leave the full investigation to future work.

Once the safe operational region is computed, the detection of control conflicts is straightforward: the network is guaranteed to be free of conflicts only when the safe operational region includes all possible network states; otherwise, there will be some state that once entered, will eventually destabilize the network.

V. CONCLUSION AND FUTURE WORK

We take the position that control conflicts arising from running multiple SDN apps must be carefully prevented. We showed with a motivational example that such a conflict could cause the network to enter permanent state oscillation. We then sketched a rigorous automaton-based approach for systematically computing the interaction of multiple SDN apps and detecting control conflicts. Furthermore, we proposed to apply sequence-based specification to derive the individual automata for each SDN app. In future work, we plan to continue our investigation and develop the proposed approach to a complete theoretical framework. We also plan to validate the proposed approach using more and larger-scale case studies.

ACKNOWLEDGMENT

This material is based upon work supported by the National Science Foundation (NSF) under Grant CNS-1660569, and by Ontario Systems through the NSF Security and Software Engineering Research Center (S²ERC). Any opinions, findings, conclusions, or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of NSF, Ontario Systems, or S²ERC.

REFERENCES

- [1] "HP launches industry's first SDN app store, unleashing new wave of networking innovations," <http://www8.hp.com/us/en/hp-news/press-release.html?id=1798074>.
- [2] "Software-Defined Networking: The New Norm for Networks," <https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf>.
- [3] 2018, Proto_Seq (Prototype Sequence Enumeration Tool). Software Quality Research Laboratory, The University of Tennessee. <https://sourceforge.net/projects/protoseq/>.
- [4] 2018, REALSBS (Requirements Elicitation and Analysis Tool for Sequence-Based Specification). Software Quality Research Laboratory, The University of Tennessee. <https://sourceforge.net/projects/realsbs/>.
- [5] A. AuYoung, Y. Ma, S. Banerjee, J. Lee, P. Sharma, Y. Turner, C. Liang, and J. C. Mogul, "Democratic resolution of resource conflicts between SDN control programs," in *Proc. of ACM CoNext*, 2014.
- [6] A. Bairley and G. Xie, "Orchestrating network control functions via comprehensive trade-off exploration," in *Proc. of IEEE NFV-SDN*, 2016.
- [7] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte, "Real time network policy checking using header space analysis," in *Proc. of USENIX NSDI*, 2013.
- [8] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. Godfrey, "VeriFlow: Verifying network-wide invariants in real time," in *Proc. of USENIX NSDI*, 2013.
- [9] D. Kreutz, F. Ramos, P. Verissimo, C. Rothenberg, S. Azodolmolky, and S. Uhlig, "Software-defined networking: A comprehensive survey," *Proc. of the IEEE*, vol. 103, no. 1, pp. 14–76, 2015.
- [10] L. Lin, S. Prowell, and J. Poore, "An axiom system for sequence-based specification," *Theoretical Computer Science*, vol. 411, no. 2, pp. 360–376, 2010.
- [11] J. Mogul, A. AuYoung, S. Banerjee, L. Popa, J. Lee, J. Muidgonda, P. Sharma, and Y. Turner, "Corybantic: Towards the modular composition of SDN control programs," in *Proc. of ACM HotNets*, 2013.
- [12] S. Prowell and J. Poore, "Sequence-based software specification of deterministic systems," *Software: Practice and Experience*, vol. 28, no. 3, pp. 329–344, 1998.
- [13] —, "Foundations of sequence-based software specification," *IEEE Transactions on Software Engineering*, vol. 29, no. 5, pp. 417–429, 2003.
- [14] S. Prowell, C. Trammell, R. Linger, and J. Poore, *Cleanroom Software Engineering: Technology and Process*. Reading, MA: Addison-Wesley, 1999.
- [15] P. Sun, R. Mahajan, J. Rexford, L. Yuan, M. Zhang, and A. Arefin, "A network-state management service," in *Proc. of ACM SIGCOMM*, 2014.