Leveraging Rigorous Software Specification Towards Systematic Detection of SDN Control Conflicts

Xin Sun and Lan Lin
Ball State University, Muncie, Indiana 47306, USA
{xsun6, llin4}@bsu.edu

***Abstract*—This paper leverages a well-established, rigorous method for software specification to approach a unique problem introduced by the emerging software-defined networking (SDN) paradigm, i.e., the potential control conflict arising from running multiple SDN apps in the same network. As individual SDN apps have different optimization objectives and each assumes full control of the network, their interaction is often unpredictable and can destabilize the network as a result. We propose a theoretical modeling framework for systematically detecting such conflicts, which is deeply rooted in automaton theory and software engineering. The key novelty and strength of our approach is its ability to model and reason about the interaction 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. To the extent of our knowledge our work is the first to adapt rigorous software specification to the constructive, formal modeling of SDN apps running on a network topology, and through a formal treatment not only straightforwardly detects and locates such conflicts but also examines and analyzes important network properties (e.g., safe operational regions) of interest to network managers.**

I. Introduction

This paper reports an application of rigorous software specification to the emerging software-defined networking (SDN) paradigm, to approach a unique problem introduced by "SDN apps" (also called network functions), which are software applications running on top of the SDN controller platform, offering a variety of functionalities such as load-balancing, power-saving, quality of service, access control, WAN optimization, network virtualization, to name a few (a comprehensive survey is presented in [1]).

Because (i) the apps are created by different developers (virtually anyone can develop and release them; marketplace exists today for selling and buying SDN apps, e.g., Hewlett-Packard App Store [2]), (ii) each app typically manages/optimizes a single aspect of the network (e.g., performance, security, resiliency, energy usage, etc.), and has a single optimization objective, and (iii) each app assumes full control of the whole network, they may seek to change the underlying network in conflicting ways. The interaction of their conflicting outputs can be unpredictable and, as a result, destabilize the network (a case study of such conflict is presented in Sec. III).

In our preliminary work [3] a fine-grained approach was proposed that models the SDN apps and their interactions using deterministic finite state automata. However, derivation of the automata was completely *manual*, and the conflicts were manually identified afterwards based on human insight. The modeling process was tedious with much trial and error, and we were not able to prove the correctness of either the derived automata or the located conflicts.

This paper presents an advanced approach that significantly extends our previous work, with the following contributions:

1) We adapted a well-established, rigorous method for software specification (i.e., *sequence-based specification* [4, 5, 6, 7]) to systematically derive a formal (automaton) model for each SDN app that runs on a network topology (Sec. IV). The adapted method is inherently rigorous, systematic, and constructive, and does not require knowledge of the implementation details of the apps; as such, we believe the method is very practical.
2) We developed a theoretical framework for analyzing the interaction of multiple SDN apps running in parallel (Sec. V). With our new theory, important network properties of interest to network managers, such as the safe operational region (i.e., network states under which multiple SDN apps can run free of conflict), can for the first time be formally defined and precisely analyzed. We demonstrated how potential control conflicts can be straightforwardly and systematically detected (Sec. VI).

II. Related Work

The problem of control conflicts caused by running multiple independently-developed SDN apps, and the resulting destabilization of the network, has recently started to receive attention from the research community. Corybantic [8] and Athens [9] take a coarse-grained approach that resolves potential conflicts at run-time (i.e., when the network is in operation). They let individual SDN apps generate "proposals" for network configuration changes, and then require each app to evaluate all proposals, based on some predefined policy or voting mechanism. While this approach can successfully resolve conflicts, it does so with significant costs: (i) it requires all SDN apps to implement the additional functionality of generating and evaluating proposals; (ii) it only selects a single proposal at a time, which can leave out potentially better solutions that combine multiple conflict-free proposals, leading to sub-optimal network configurations; (iii) it cannot identify the root cause of the conflicts. Bairley and Xie [10] take a similar approach (it 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, our approach does not have any of those drawbacks.

Prior works [11, 12] concern individual-flow-level forwarding behavior and can detect policy violations (such as black holes and loops) at that level; however, none of them can detect the network state oscillation caused by conflicts of running multiple SDN apps, since state oscillation is not a violation of flow policy (it is a higher-level issue).

DOI reference number: 10.18293/SEKE2019-014

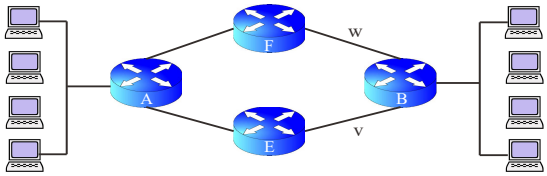

Fig. 1. The Network Topology Used in the Case Study

III. Control Conflicts and Network State Oscillation: A Case Study

A software-defined network typically has a three-tier architecture [13]. The bottom layer, also called the data plane, consists of "dumb" switches and other hardware boxes that primarily focus on packet streaming. The middle layer, also called the controller or the network operating system, is a software platform that directly manages the hardware boxes and offers an abstraction of the network resources via a set of application programming interfaces (APIs). The APIs in turn enable the development of network functions, also called "SDN apps", which form the top layer.

An SDN app generally seeks to optimize some aspect of a network, by modifying the network 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. [14] SDN apps control the network state by issuing commands to the controller via the northbound API; the controller then compiles each command down to a set of new configurations to be installed on the data plane.

We observed that, when multiple SDN apps are trying to control the network state, conflicts may occur. For the purpose of demonstrating both the problem and our solution in a tangible and unambiguous manner, we now present a concrete case study involving two popular types of SDN apps that have been extensively researched (e.g., [15, 16, 17]): a *power-saving* app and a *load-balancing* app, running on a toy network as depicted by Fig. 1. 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 on each link (w or v) 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 two apps work in the following way:

Power-saving app: it seeks to aggregate traffic to one of the two paths (without loss of generality, assuming it's always the top path), and then turn off Router E to reduce energy use. To realize this objective, it checks the utilization rates of Links w and v every M seconds, and whenever the utilization rates of both links are low or zero for two consecutive cycles, it issues a command to the controller to turn off E and route all future packets to the top path.

Load-balancing app: it seeks to spread traffic across all possible paths, to minimize the load of any link. It checks the utilization rates of w and v every M seconds. Whenever one is high and the other is low or zero for three consecutive cycles, it issues a command to the controller to route a larger fraction of new flows entering the network in the next M seconds through the less congested path. It stops issuing such commands once the utilization rates of both links become high, or both become low or zero (i.e., the load is balanced).

Intuitively, when the two apps run on the same network, control conflict might occur, as one seeks to aggregate traffic to a subset of paths, while the other seeks to spread traffic evenly on all paths. However, the simple intuition is unable to tell precisely when and how control conflict may arise, and the consequences of the conflicts. 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. Modeling Individual SDN Apps

This section first presents a formal model for individual SDN apps, and then describes our approach for deriving the model. The two SDN apps described in Sec. III are used as running examples to illustrate the modeling process.

A. An Automaton-Based Model

Our model of an SDN app running on a network topology follows 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, $\delta : Q \times \Sigma \to Q$ is the transition function, $\nu : Q \to \Gamma$ is the output function, and $q_0 \in Q$ is the starting state.

Specifically for the SDN apps, they 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). Additionally, the model only needs to include states that are reachable from the initial state; non-reachable states can be safely ignored as they will never be realized. To model reachable states, we need to extend the transition function of a Moore machine to input sequences, as follows: $\hat{\delta} : Q \times \Sigma^* \to Q$ is defined by (1) $\hat{\delta}(q, \lambda) = q$, where λ is the empty input sequence; and (2) $\hat{\delta}(q, wa) = \delta(\hat{\delta}(q, w), a)$ for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$.

Definition 2 (Network Function Moore Machine): A *network function Moore machine* is a Moore machine $(Q, \Sigma, \Gamma, \delta, \nu, q_0)$ satisfying (1) $\{0, \omega\} \subset \Gamma$, and (2) for any $q \in Q$ there exists $w \in \Sigma^*$ such that $\hat{\delta}(q_0, w) = q$.

Here Q represents the set of *software states* of the SDN app (note that these are different from the *network states*); Σ represents inputs to the SDN app, which includes the network states such as link load and topology; 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 (for an input sequence not possible to occur – this is defined for completeness purposes). The second condition ensures that any state in Q must be reachable from the initial state q_0.

B. A Rigorous Approach for Deriving the Automaton

Our approach for deriving the Moore machine of a given SDN app is rigorous (based on the automaton theory), systematic (we offer a systematic process to follow), and constructive (the state machine will be discovered at the end of the process). It is based on a well-established rigorous method for software specification, i.e., *sequence-based specification* [7, 6, 5, 4]. The input to our approach is an informal description of how the SDN app is supposed to work on the given network topology

(this description is termed *"functional requirements"* in the field of software engineering). The output is a Moore machine representing the working mechanism of the SDN app. The approach has three key steps: identification of stimuli and responses, sequence enumeration, and automaton construction.

1) **Step 1: Identification of stimuli and responses:** We first identify a list of *stimuli* (i.e., inputs) and *responses* (i.e., outputs) of the SDN app running on the given network topology. The power-saving app and the load-balancing app share a common set of inputs: $\{H_wH_v, H_wL_v, H_wZ_v, L_wH_v, L_wL_v, L_wZ_v, Z_wH_v, Z_wL_v, Z_wZ_v\}$, where H, L, and Z indicate link utilization is high, low, or zero, respectively, and the subscript indicates which link (i.e., w or v). Each input contains information about the utilization rates of both links. Outputs of the power-saving app are $\{$`OnE`, `OffE`, $0\}$, where `OnE`/`OffE` represents turning on/off the switch E (a request to turn on/off E, rather than a successful command or operation, as such request may be overridden [3]), respectively, and 0 represents the *null* response, i.e., no output issued by the SDN app that potentially changes the network's state. Outputs for the load-balancing app are $\{$`FtoE`, `EtoF`, $0\}$, where `FtoE`/`EtoF` represents moving flows from F/E to E/F, respectively.

2) **Step 2: Sequence enumeration:** We start with an informal description of the requirements for each SDN app running on the given topology, and tag (number) them to facilitate tracing decisions we have made (in the specification process) to the tagged requirements or derived requirements (as a by-product the process also leads to the discovery of derived requirements that were not originally stated, and the resolution/correction of inconsistent/incorrect requirements). Requirements for the power-saving app and for the load-balancing app are listed in Tables I and II, respectively. Original requirements were retrieved from descriptions of the functions in [3]. Tags that begin with D indicate new requirements we derived in the specification process (following the same assumptions as implied by [3]).

Next we perform the key step of sequence-based specification, called *sequence enumeration*, to discover/construct every detail of the state machine. We enumerate all finite sequences of stimuli (inputs) in length-lexicographical order (i.e., first by length, and within the same length lexicographically), and for each enumerated sequence make two decisions:

- Response mapping. We map the sequence of inputs to a response (output) of the network function. The response is the output the network function produces in response to the very last input in the sequence, given the input history. For instance, the sequence $L_wL_v.L_wL_v$ (we concatenate inputs with dots) is mapped to the response `OffE` by the power-saving function by Requirement 2 (Table I). We introduce two special responses in theory: the *null* response, denoted by 0, for the lack of an externally observable output (there might have been an internal state update), and the *illegal* response, denoted by ω, for an operationally unrealizable sequence of inputs (the sequence cannot occur in practice). A sequence is *illegal* if it is mapped to ω; otherwise, it is *legal*.
- Equivalence declaration. We determine if the sequence is *Moore equivalent* to (and hence can be reduced to) a previously enumerated sequence. Two sequences u and v are *Moore equivalent* if and only if for any input sequence w, uw and vw always map to the same response by the network function. This implies u and v are mapped to the same response as well (as w could be the empty sequence). For instance, $L_wL_v.L_wZ_v$ can be reduced to the prior sequence $L_wL_v.L_wL_v$ by Requirement 2 for the power-saving function. Two equivalent sequences arrive at the same state of the underlying Moore automaton starting from the initial state. We chose to model it using a Moore machine to be consistent with the transducer model in [3], whereas in software specification *Mealy equivalence* and a Mealy machine are used as they lead

TABLE I
REQUIREMENTS FOR THE POWER-SAVING FUNCTION

Tag	Requirement
1	Assume each link exhibits a utilization rate that can vary over time. This rate can be sampled on each cycle and is either high (H), low (L), or zero (Z).
2	Suppose a power-saving machine tries to power down E when the link utilization of both links v and w is low or zero for two consecutive cycles. Any new flows are then routed to F.
3	The machine waits for two consecutive cycles when link w is experiencing heavy load before restoring power to E.
4	The machine is designed such that the attempt to turn E off may have failed or been overridden. Thus it is treated more as a request than a command.
D1	Assume there is an attempt to turn on E at system start/initialization.
D2	Assume that turning on E is also treated as a request than a command.
D3	The latest two cycles' link utilization for both v and w is necessary to determine if a command needs to be issued to turn on/off E.
D4	Except for at system initialization, an attempt to restore power to E has to follow a recent power off attempt for which no other restore attempt has been made, and only a zero link rate of v and two consecutive high link rates of w have been observed since that power off attempt.
D5	After a recent power off attempt that appears unsuccessful (by a non-zero link utilization of v), another two consecutive cycles of both links v and w being low or zero need to be observed to power off E again.
D6	After a recent power off attempt, if v has been observed of zero utilization it suggests the latest power off attempt might have been successful, but is subject to future observations.

TABLE II
REQUIREMENTS FOR THE LOAD-BALANCING FUNCTION

Tag	Requirement
D1	Assume load is balanced at system start/initialization.
D2	Continued balanced load does not change the (load) balanced state the function is in.
D3	The load-balancing function needs to observe three consecutive cycles of the same imbalanced load patterns before functioning.
D4	If load is observed re-balanced from an imbalanced state, the load-balancing function returns to the (load) balanced state.
D5	If heavy load switches between the two links v and w, the load-balancing function transitions to the corresponding imbalanced state (based on which link has heavy load).
D6	After observing three consecutive cycles of the same imbalanced load patterns (i.e., high w and low/zero v, or high v and low/zero w), the load-balancing function directs load from the heavy link to the light link.
D7	Load-balancing operations are issued based on the most recent three consecutive cycles' loads only, irrespective of whether the same operation has been recently issued.

to a shorter enumeration table. One could easily transform between Moore and Mealy machines. When reducing a sequence to a prior sequence, we follow the reduction chain and get to the sequence that is itself unreduced. For instance, H_wZ_v is reduced to H_wH_v and not H_wL_v as H_wL_v is further reduced to H_wH_v (for the power-saving function). A sequence is *reduced* if it is Moore equivalent to a prior sequence in length-lexicographical order; otherwise, it is *unreduced*.

One starts with the empty sequence λ. To get all the sequences of Length $n+1$ (integer $n \geq 0$) one extends all the sequences of Length n by every stimulus, and considers the extensions in lexicographical order. This inherently combinatorial process can be controlled by two observations:

- If Sequence u is reduced to a prior sequence v, there is no need to extend u, as the behaviors of the extensions are defined by the same extensions of v.
- If Sequence u is illegal, there is no need to extend u, as all of the extensions must also be illegal (i.e., physically unrealizable).

Therefore, only legal and unreduced (also called *extensible*) sequences of Length n get extended by every stimulus for consideration at Length $n+1$. The process continues until all the sequences of a certain length are either illegal or reduced to prior sequences. The enumeration becomes *complete*. This terminating enumeration length is discovered in enumeration, and varies from application to application.

Excerpt of an enumeration for the power-saving function is shown in Table III. We show the enumeration until Length 3 due to lack of space (the enumeration terminates at Length 4). Columns of the table are for enumerated sequences, their mapped responses, possible reductions to prior sequences under Moore equivalence, and traces to requirements. We similarly performed the enumeration for the load-balancing function, but omitted it here due to lack of space.

3) **Step 3: Construction of the automaton:** We observe that the completed enumeration from the previous step encodes a Moore machine as follows. First we retrieve all the unreduced sequences; each represents a state, whose associated output is the mapped response of the unreduced sequence (that can be read off from the table). Table IV shows the mapping from unreduced sequences in the power-saving enumeration to Moore states of the power-saving automaton. The mapping table for the load-balancing function is omitted here due to lack of space.

With this observation we are now ready to construct the automaton. For doing so, we simply map each row in the enumeration table (except the empty sequence) to a transition in the Moore machine as follows: if the prefix sequence u concatenated with the current stimulus a is reduced to the sequence w (here we treat any unreduced sequence as being reduced to itself; an equivalence relation must be reflexive), then a transition triggered by the input a goes from the state represented by u to the state represented by w. For instance, $H_wH_v.H_wH_v$ being reduced to H_wH_v in the power-saving enumeration implies a transition from State r_0 to State r_0 on Input H_wH_v for the power-saving automaton.

The state machines for both apps constructed in this step are shown in Fig. 2 and Fig. 3. Their equivalent formal definitions are omitted for lack of space.

TABLE III
EXCERPT OF AN ENUMERATION FOR THE POWER-SAVING FUNCTION UNTIL LENGTH 3

Sequence	Response	Equivalence	Trace
λ	OnE		D1, D2
H_wH_v	0		D3
H_wL_v	0	H_wH_v	D3
H_wZ_v	0	H_wH_v	D3
L_wH_v	0	H_wH_v	D3
L_wL_v	0		D3
L_wZ_v	0	L_wL_v	D3
Z_wH_v	0	H_wH_v	D3
Z_wL_v	0	L_wL_v	D3
Z_wZ_v	0	L_wL_v	D3
$H_wH_v.H_wH_v$	0	H_wH_v	D3
$H_wH_v.H_wL_v$	0	H_wH_v	D3
$H_wH_v.H_wZ_v$	0	H_wH_v	D3
$H_wH_v.L_wH_v$	0	H_wH_v	D3
$H_wH_v.L_wL_v$	0	L_wL_v	D3
$H_wH_v.L_wZ_v$	0	L_wL_v	D3
$H_wH_v.Z_wH_v$	0	H_wH_v	D3
$H_wH_v.Z_wL_v$	0	L_wL_v	D3
$H_wH_v.Z_wZ_v$	0	L_wL_v	D3
$L_wL_v.H_wH_v$	0	H_wH_v	D3
$L_wL_v.H_wL_v$	0	H_wH_v	D3
$L_wL_v.H_wZ_v$	0	H_wH_v	D3
$L_wL_v.L_wH_v$	0	H_wH_v	D3
$L_wL_v.L_wL_v$	OffE		2
$L_wL_v.L_wZ_v$	OffE	$L_wL_v.L_wL_v$	2
$L_wL_v.Z_wH_v$	0	H_wH_v	D3
$L_wL_v.Z_wL_v$	OffE	$L_wL_v.L_wL_v$	2
$L_wL_v.Z_wZ_v$	OffE	$L_wL_v.L_wL_v$	2
$L_wL_v.L_wL_v.H_wH_v$	0	H_wH_v	4
$L_wL_v.L_wL_v.H_wL_v$	0	H_wH_v	4
$L_wL_v.L_wL_v.H_wZ_v$	0		3, D6
$L_wL_v.L_wL_v.L_wH_v$	0	H_wH_v	4
$L_wL_v.L_wL_v.L_wL_v$	0	L_wL_v	4, D5
$L_wL_v.L_wL_v.L_wZ_v$	0		D6
$L_wL_v.L_wL_v.Z_wH_v$	0	H_wH_v	4
$L_wL_v.L_wL_v.Z_wL_v$	0	L_wL_v	4, D5
$L_wL_v.L_wL_v.Z_wZ_v$	0	$L_wL_v.L_wL_v.L_wZ_v$	D6

TABLE IV
THE MAPPING FROM UNREDUCED SEQUENCES TO MOORE STATES FOR THE POWER-SAVING FUNCTION

Unreduced Sequence	State	Output
λ	r_5	OnE
H_wH_v	r_0	0
L_wL_v	r_1	0
$L_wL_v.L_wL_v$	r_2	OffE
$L_wL_v.L_wL_v.H_wZ_v$	r_4	0
$L_wL_v.L_wL_v.L_wZ_v$	r_3	0

V. MODELING JOINT EFFECT OF MULTIPLE SDN APPS

One benefit of using automata to model SDN apps is that, when two SDN apps run in parallel on the same network topology, their behavior can be modeled by the standard automaton product, which can be straightforwardly computed by applying the following definition:[1]

Definition 3 (Product of Network Function Moore Machines): 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

[1]This operation can be easily extended to more than two SDN apps and performed in a successive way.

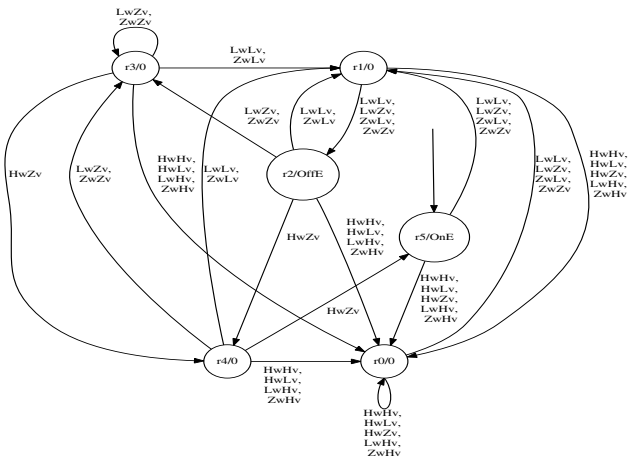

Fig. 2. The Power-Saving Moore Machine

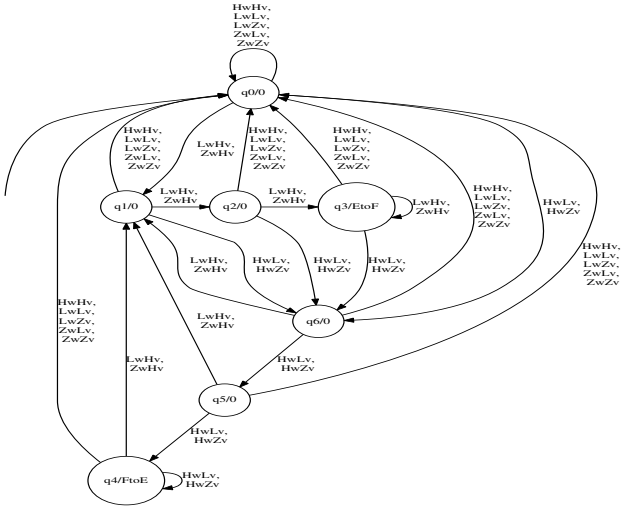

Fig. 3. The Load-Balancing Moore Machine

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:

$\delta((p,q),(a,b)) = (\delta_1(p,a), \delta_2(q,b))$,

$\nu((p,q)) = (\nu_1(p), \nu_2(q))$.

We extend the *null* response and the *illegal* response to the product of two network function Moore machines 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 we also extend the concept of being legal/illegal to inputs of a product automaton.

Example 4: The product of the power-saving Moore machine M_1 and the load-balancing Moore machine M_2 is defined by $M = M_1 \times M_2 = (Q_1 \times Q_2, \Sigma \times \Sigma, \Gamma_1 \times \Gamma_2, \delta, \nu, (r_5, q_0))$, where δ and ν are defined by:

$\delta((p,q),(a,b)) = (\delta_1(p,a), \delta_2(q,b))$,

$\nu((p,q)) = (\nu_1(p), \nu_2(q))$.

Observe that $\{(a,a) : a \in \Sigma\}$ is the set of all legal inputs of M.

One interesting observation on the constructed product Moore machine is that not all its states may be reachable from the initial state. Intuitively, for a state (p_j, q_j) in the product Moore machine $M = M_1 \times M_2$ to be reachable from another state (p_i, q_i), there must exist paths *of the same length* from p_i to p_j in M_1 and from q_i to q_j in M_2.

Definition 5 (Reachable States): Given the product $M = (Q, \Sigma, \Gamma, \delta, \nu, q_0)$ of two network function Moore machines and a set of legal inputs $I \subseteq \Sigma$, the set RS of *reachable states* of M is defined by $RS = \{q : q \in Q, \exists w \in I^*.\, \hat{\delta}(q_0, w) = q\}$.

This definition enabled us to develop a simple algorithm to automatically identify all reachable states in a product Moore machine, which we omitted here due to space constraint.

Definition 6 (Joint Network Function Moore Machine): Given the product $M = M_1 \times M_2 = (Q, \Sigma, \Gamma, \delta, \nu, q_0)$ of two network function Moore machines M_1 and M_2, and a set of legal inputs $I \subseteq \Sigma$, let RS be the set of reachable states of M, δ' be δ restricted to $RS \times I$, and ν' be ν restricted to RS: $\delta'(q,a) = \delta(q,a)$, $\nu'(q) = \nu(q)$. Let $\Gamma' = range(\nu') \cup \{0, \omega\}$, where $range(\nu')$ denotes the range of ν'. The *joint network function Moore machine* M' of M_1 and M_2 can be defined by $M' = (RS, I, \Gamma', \delta', \nu', q_0)$.

Clearly a joint network function Moore machine satisfies the definition for a network function Moore machine.

Example 7: The joint network function Moore machine M of the power-saving and the load-balancing automata is omitted due to space. Only 13 out of the 42 states of $Q_1 \times Q_2$ are reachable, and included in the joint automaton.

VI. Property Analysis and Conflict Detection

We present analysis of the joint network function Moore machine, which enables detection of control conflicts as well as answers the many questions regarding the network's behavior. Our analysis focuses on three critical and closely related concepts: stable states of a Moore machine, safe operational region of a network, and conflict freeness of SDN apps.

A. Stable States of a Moore Machine

Intuitively, an SDN app enters a stable state, if it no longer attempts to modify the network state (i.e., the network has reached a desirable state from the app's perspective). This intuition can be formally defined using the automaton model. To do so, we extend the output function of a Moore machine to pairs of states and input sequences as follows. $\hat{\nu} : Q \times \Sigma^* \to \Gamma^*$ is defined by (1) $\hat{\nu}(q, \lambda) = \nu(q)$, and (2) $\hat{\nu}(q, wa) = \hat{\nu}(q,w)\nu(\hat{\delta}(q, wa))$ for all $q \in Q$, $w \in \Sigma^*$, $a \in \Sigma$.

We can now formally define the stable states of a network function Moore machine as follows:

Definition 8 (Stable State): Let $q \in Q$ be a state of the network function Moore machine $M = (Q, \Sigma, \Gamma, \delta, \nu, q_0)$. q is a *stable state* of M iff the following hold for all $a \in \Sigma$:

(1) $\delta(q', a) = q$ implies $\hat{\nu}(q, a^n) = 0^{n+1}$ for all integer $n \geq 0$, and

(2) $q = q_0$ implies $\nu(q_0) = 0$.

Informally, two conditions need to be satisfied for a state to be a *stable state*: (1) on any input by which there is an incoming arc to this state, if a sequence of such input continues it will never land on any state that produces a non-null response, which could potentially change this input (i.e., a network state); and (2) the starting state must have the null output to be a stable state.

It is easy to see that this definition enforces any stable state be associated with the null response, as shown by the following theorem (proof is omitted due to lack of space).

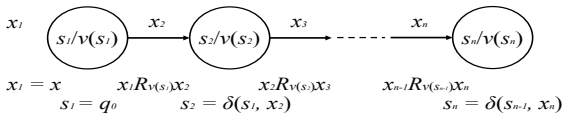

Fig. 4. An Example Sequence of Inputs and States for Analyzing the Safe Operational Region

Theorem 9: Let q be a stable state of the network function Moore machine $M = (Q, \Sigma, \Gamma, \delta, \nu, q_0)$. $\nu(q) = 0$.

Example 10: Applying Definition 8, it can be straightforwardly discovered that the constructed power-saving automaton M_1 has two stable states r_0 and r_3; the constructed load-balancing automaton M_2 has one stable state q_0; and the joint automaton M (Example 7) has two stable states (r_0, q_0) and (r_3, q_0). We omit the proofs due to space constraint.

B. Safe Operational Region of a Network

Intuitively, a network becomes "stabilized" when none of the SDN apps running on top of it attempts to change its state; that is, all the SDN apps have entered a stable state as defined by Definition 8. Thus we consider a network state to be "safe", if it can eventually (i.e., after a finite number of state changes) lead to a stabilized network, which guarantees no state oscillation. We term the set of all such safe states of a network to be the *safe operational region* of the network.

Note that a network's state is already encapsulated in the stimuli (i.e., inputs) of a network function Moore machine. Hence we are able to formally define the concept of safe operational region of a network, as follows:

Definition 11 (Safe Operational Region): Given a network function Moore machine $M = (Q, \Sigma, \Gamma, \delta, \nu, q_0)$, a set of binary relations R_r over Σ for each $r \in \Gamma - \{\omega\}$ such that (1) $R_0 : \Sigma \to \Sigma$ is the identify function; and (2) $(r \neq 0,\ aR_r b)$ implies either $\delta(p, a) = q,\ \nu(q) = r$ for some $p, q \in Q$ or $\nu(q_0) = r$, and $x \in \Sigma$, x is in the *safe operational region* iff the alternating sequence of inputs and states $x_1, s_1, x_2, s_2, \cdots, x_n, s_n$, in which $x = x_1$, $q_0 = s_1$, $x_{i-1} R_{\nu(s_{i-1})} x_i$, $s_i = \delta(s_{i-1}, x_i)$ for all $i \geq 2$, $\nu(s_i) \neq \omega$ for all $i \geq 1$, ends with some stable state s_n of M. The *safe operational region* of the network is the set of all such x's.

The set of binary relations R_r over I define how each output of an SDN app modifies the status of the network (notice that the null response does not modify the network status, as indicated by the identity function). Informally, a network state, as encapsulated in the input x, is in the safe operational region if and only if a stable state of the Moore machine will always be reached when starting from the initial state of the Moore machine with the input x. At that point the network state will no longer be changed by the running SDN app(s). Fig. 4 illustrates an example sequence of inputs and states that need to be examined from the initial state q_0 with a specific input (say x).

Example 12 (Safe Operational Region of Example Network): Applying Definition 11 on the joint automaton as defined in Example 7, it can be computed that the safe operational region of the example network (Fig. 1) includes $H_w H_v$, $L_w Z_v$, and $Z_w Z_v$. We omit the algorithm here due to lack of space.

C. Conflict Freeness of SDN Apps

An SDN network is free of conflict, if starting from any state it can eventually become stabilized; that is, every possible network state is in its safe operational region. We have the following formal definition:

Definition 13 (Conflict-Free SDN apps): Two or more SDN apps are *conflict-free* if and only if their joint network function Moore machine has the set of inputs identical to the safe operational region of the network.

Note that Definition 13 not only enables direct detection of potential conflict, but also precisely specifies under what network condition such conflict will arise.

Example 14 (Conflict in the Example Network): Applying Definition 13, we immediately see that the two SDN apps running on the example network are not conflict-free: this is because a subset of inputs to the joint network function Moore machine including $H_w L_v$, $H_w Z_v$, $L_w H_v$, $L_w L_v$, $Z_w H_v$, and $Z_w L_v$, is left out of the safe operational region.

VII. Conclusion and Future Work

We presented a theoretical framework, based on rigorous software specification, for detecting conflicts caused by running multiple SDN apps in parallel. Our future work includes experimental validation and further investigation on the scalability and applicability of the framework.

Acknowledgment

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

References

[1] D. Kreutz et al. "Software-Defined Networking: A Comprehensive Survey". In: *Proc. of the IEEE* 103.1 (2015), pp. 14–76.

[2] *Airheads Community SDN App Store*. https://community.arubanetworks.com/t5/SDN-Apps/ct-p/SDN-Apps.

[3] D. Volpano, X. Sun, and G. Xie. "Towards Systematic Detection and Resolution of Network Control Conflicts". In: *Proc. of ACM HotSDN*. Chicago, IL, 2014.

[4] S. Prowell and J. Poore. "Sequence-based software specification of deterministic systems". In: *Software: Practice and Experience* 28.3 (1998), pp. 329–344.

[5] S. Prowell et al. *Cleanroom Software Engineering: Technology and Process*. Reading, MA: Addison-Wesley, 1999.

[6] S. Prowell and J. Poore. "Foundations of sequence-based software specification". In: *IEEE Transactions on Software Engineering* 29.5 (2003), pp. 417–429.

[7] L. Lin, S. Prowell, and J. Poore. "An axiom system for sequence-based specification". In: *Theoretical Computer Science* 411.2 (2010), pp. 360–376.

[8] J. Mogul et al. "Corybantic: Towards the Modular Composition of SDN Control Programs". In: *Proc. of ACM HotNets*. 2013.

[9] A. AuYoung et al. "Democratic Resolution of Resource Conflicts Between SDN Control Programs". In: *Proc. of ACM CoNext*. Sydney, Australia, 2014.

[10] A. Bairley and G. Xie. "Orchestrating network control functions via comprehensive trade-off exploration". In: *Proc. of IEEE NFV-SDN*. Palo Alto, CA, 2016.

[11] A. Khurshid et al. "VeriFlow: Verifying Network-Wide Invariants in Real Time". In: *Proc. of USENIX NSDI*. 2013.

[12] P. Kazemian et al. "Real Time Network Policy Checking Using Header Space Analysis". In: *Proc. of USENIX NSDI*. 2013.

[13] *Software-Defined Networking: The New Norm for Networks*. https://www.opennetworking.org/images/stories/downloads/sdn-resources/white-papers/wp-sdn-newnorm.pdf.

[14] P. Sun et al. "A Network-state Management Service". In: *Proc. of ACM SIGCOMM*. Chicago, IL, 2014.

[15] M. Al-Fares et al. "Hedera: Dynamic Flow Scheduling for Data Center Networks". In: *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation*. San Jose, California, 2010.

[16] S. Jain et al. "B4: Experience with a Globally-deployed Software Defined WAN". In: *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*. Hong Kong, China, 2013.

[17] B. Heller et al. "ElasticTree: Saving Energy in Data Center Networks." In: *Proceedings of USENIX NSDI*. 2010.