A model checking based approach for detecting SDN races

Evgenii Vinarskii, Jorge López, Natalia Kushik, Nina Yevtushenko, Djamal Zeghlache

A Model Checking based Approach for Detecting SDN Races*

Evgenii Vinarskii[1], Jorge López[2], Natalia Kushik[2], Nina Yevtushenko[3], and Djamal Zeghlache[2]

[1] Lomonosov Moscow State University, 1 Leninskiye Gory street, 119991 Moscow, Russia
[2] SAMOVAR, CNRS, Télécom SudParis, Université Paris-Saclay, 9 rue Charles Fourier, 91000 Évry, France
[3] Ivannikov Institute for System Programming of the Russian Academy of Sciences, 25 Alexander Solzhenitsyn street, 109004, Moscow, Russia
vinevg2015@gmail.com, {jorge.lopez, natalia.kushik, djamal.zeghlache}@telecom-sudparis.eu, evtushenko@ispras.ru

Abstract. The paper is devoted to the verification of Software Defined Networking (SDN) components and their compositions. We focus on the interaction between three basic entities, an application, a controller, and a switch. When the application submits a request to the controller, containing a set of rules to configure, these rules are expected to be 'pushed' and correctly applied by the switch of interest. However, this is not always the case, and one of the reasons is the presence of races or concurrency issues in SDN components and related interfaces. We propose a model checking based approach for deriving test sequences that can identify SDN races. The test generation strategy is based on model checking, and related formal verification is performed with the use of extended automata specifying the behavior of the components of interest; Linear Temporal Logic (LTL) formulas are utilized to express the properties to check. We generalize the races of interest and propose an approach for deriving the corresponding LTL formulas that are later used for verifiation. The Spin model checker is used for that purpose and thus, Promela specifications for interacting components are also provided; those are: the ONOS REST API, the ONOS controller and an OpenFlow Switch. An experimental evaluation with the aforementioned components showcases the existence of race conditions in their compositions.

Keywords: Software Defined Networking (SDN) · Races · Controller · Switch · Verification · Testing

1 Introduction

Software Defined Networking (SDN) technologies are actively developing nowadays and are utilized in future network standards, such as for example 5G. Therefore, thorough testing, verification and validation of the components of such networks are crucial. A number of works are in fact dedicated to the verification and testing of SDN enabled

* The results in this work were partially funded by the Celtic-Plus European project SENDATE, ID C2015/3-1, and the Russian Foundation for Basic Research (RFBR), grant No 18-01-00854.

switches and SDN controllers (see, for example [14], [13], [20]). At the same time, even if two SDN components have been carefully verified up to some extent, it is still possible that their composition can cause network misconfigurations and inconsistencies. That is the reason why in this work, we focus on the *interaction* of three crucial SDN components, namely an SDN application, an SDN controller and an SDN enabled switch. Generally speaking, the communication between the latter two takes place via the OpenFlow protocol [10], and therefore in this work we focus on the OpenFlow specification as the base for further validation [4].

We note that the OpenFlow protocol does not guarantee that requests between the controller and switch are delivered in the same order as they have been sent; thus races between the requests are possible in the OpenFlow channels. In [16], the authors analyze the reasons and impact of such races. In fact, they identify three types of races: i) packet races on the data plane, ii) Post/Get/Delete races on the *southbound* interface (between the controller and the switch), and iii) a combination of data and control races on the southbound interface. To identify races in the components of interest, i.e., to simulate the races and conditions of those using a formal finite state model (or a composition of those) we propose another classification. We identify the following race types: races between inputs and outputs at a given state of the automaton and races in the communication channel when two or more automata are working in a dialog mode.

We note that the problem of SDN races has been previously raised and existing utilities, such as for example SDNracer, are quite effective [3]. The proposed approaches rely on the definition of a specific (partial) order between the possible SDN events and further monitoring if the defined order is violated. For example, in [11] the authors derive a specific *HB graph (happens-before model)* to identify the order of events. Another possibility is to take a *preventive path*, i.e., to derive the SDN components that are carefully synchronized, so that races cannot show up [9].

In this paper, we propose a complementary approach which is based on proactive testing, i.e., on generation of specific application requests that can lead to a race in an SDN framework. Test sequences that are executed against an SDN framework under test are derived using formal verification approaches. In particular, we employ the Spin model checker [5] that produces a counterexample to generate a sequence of actions that can induce a race. For this reason, we derive extended automata simulating the behavior of the interacting SDN components and later on verify potential (race) properties for this composition. Such properties are described in Linear Temporal Logic (LTL) formulas [1], which can be verified by Spin; likewise, the absence / presence of livelocks / deadlocks in the composition of interest can be verified. We highlight the necessity of the execution of a derived counterexample signaling a potential race in a given SDN component or in their composition. The reason is that in some cases an implementation under test can still be race insensible even if its model does not possess this property. Therefore, such proactive testing against SDN frameworks allows detecting races in *real* SDN enabled components.

When deriving the extended automata models for interacting components we have used both, an OpenFlow specification (OpenFlow 1.4.0), as well as the description of one of the widely utilized controllers. In this work, the experiments have been performed with the ONOS controller [2] and thus, we have extracted a state model specify-

ing its behavior. We note however, that without loss of generality, the controller and / or switch specifications can be easily modified / substituted. All the communicating components have been described in the Promela language (required by Spin). The derived composition has been later on verified by the Spin model checker. Once a counterexample is returned by Spin, the proactive race testing is performed, i.e., counterexamples for concurrency violations are applied to the implementations under study. These counterexamples in fact serve as test sequences to be applied to the implementation under test, in order to make sure the property violation over the model indeed corresponds to a race condition in the implementation.

The main contributions of this work are therefore i) a proactive model checking based testing method for SDN race detection; ii) extended automata specifications describing the behavior of an SDN controller, an SDN switch and a communication channel (southbound interface), as well as their Promela specifications that can be downloaded from [17] (which can be easily extended for specific needs); iii) the description of the potential races of two types for the considered composition in LTL, and iv) experimental results identifying an inconsistency in a common SDN composition (ONOS with Open vSwitch) with respect to the time constraints and delays for the rules to be pushed.

The structure of the paper is as follows. Section 2 presents the background. Section 3 is devoted to the description of the obtained automata together with the related Promela descriptions of the components of interest. Section 4 proposes a solution for deriving LTL formulas for detecting SDN races, as well as the algorithms for the race verification in a given application-controller-switch composition and includes an assessment of the probability to detect races for the SDN composition. Section 5 contains the experimental results while Section 6 concludes the paper and presents some avenues for future work.

2 Background

2.1 Software Defined Networking

Software defined networking is a technology that dynamically allows to centrally manage the network behavior via open interfaces by abstracting from the lower-level functionality [4]. This is done by decoupling the control plane from the data plane responsible for forwarding the traffic. The communication between the two critical SDN components, namely the controller and the switch, is performed according to a well-defined protocol (referred to as a southbound protocol). As an example, one can consider the widespread OpenFlow [4] protocol, and in fact, the specifications of the switch and communication channel with the controller in this work are extracted from the OpenFlow descriptions. In the OpenFlow protocol, the flow rules of the switch are configured by the application via the controller, through a defined set of protocol messages. Each flow rule is responsible for forwarding (or dropping) a received packet to an appropriate set of output ports; at the same time, each flow rule has a predefined set of values to which the packet header has to match. Additionally, rules can have timeouts so that they are deleted when a certain timeout expires after their installation (hard timeouts), or so that they are deleted after certain time units when they are not used / matched

(soft timeouts). Finally, rules are grouped in (flow) tables and each rule has a defined priority. Rules grouped in the tables with higher numbers are processed first, and in fact, the rule with the highest priority matching a given packet is applied. Network packets are processed according to the rules within flow tables. To better outline the working principles of SDN rules, consider the following rules installed at a given switch:

ID	Priority	Hard Timeout	TCP DST PORT	DST IP	Action
1	5000			10.0.1.22	OUT(2)
2	5001		22		OUT(3)
3	6000	20		10.0.1.23	CTRLLR

To simplify our explanation, and without loss of generality we consider that the rules are installed in the first table of the SDN-enabled switch (table 0). TCP DST PORT is the TCP destination port and DST IP is the destination IP (for further information on basic networking concepts the reader can refer to [8]). A network packet with the destination IP address 10.0.1.22 and destination TCP port 22 will be forwarded to the output 2 (due to the higher priority of Rule 1). Likewise, a network packet with destination IP address 10.0.1.21 and destination TCP port 22 will be forwarded to port 3 (the highest priority rule matching the network packet). Finally, if a network packet going to the destination IP address 10.0.1.23 (and the destination TCP port not equal to 22) arrives within the next 20 seconds, the switch sends this packet to the controller, asking for the action to take with the packet; after 20 seconds have passed, the rule will 'disappear', and a packet with destination IP 10.0.1.23 follows the default table policy, which is usually to drop the packet.

2.2 Extended Input / Output Automata and Related Races

In this paper, we model the SDN components by a simplified version of an Extended Finite Input / Output Automata, EIOA, for short. An EIOA **A** is a tuple (S, I, O, V, T, s_0), where S is a finite nonempty set of states with the designated initial state s_0; I and O are finite *input and output alphabets*; V is a finite, possibly empty set of *context variables* with the set D_V of vectors of context variables' values if $V \neq \emptyset$; T is a set of transitions. In our case, inputs and outputs are parameterized, i.e., inputs and outputs of the EIOA are pairs $(\textit{input, vector of input parameters' values})$ or $(\textit{output, vector of output parameters' values})$ and D_I (D_O) is the set of vectors of input (output) parameters' values if the set of parameters is not empty. A transition is a 6-tuple (s, a, P, v_p, v_o, s') where s, $s' \in S$ are initial and final states of the transition; $a \in I \cup O$; $P : D_V \times D_I \to \{True, False\}$ is *the transition predicate*; $v_p : D_V \to D_V$ is *the context update transition function*. The transition (s, a, P, v_p, v_o, s') is executed only when the transition predicate P evaluates to $true$ and the vectors of context variables' values and output parameters' values are updated according to the functions v_p and v_o after the transition execution. We note, however, that the EIOA model can be more complicated, for example, the set of states can have a defined subset of the final states or non-observable actions can be considered, etc. Nevertheless, these cases are not taken into account in the derived models for the controller, switch and application and we furthermore demonstrate that such simplification does not limit the targeted model checking with respect to the race related properties.

As usual, we also assume that when dealing with an EIOA, no input is accepted and no output is produced when a transition is executed. However, when both an input and an output are defined at a state, they can 'compete' between themselves, i.e., what the machine does first - accepts the input or produces the output is nondeterministically decided. We hereafter refer to such 'competition' as an *input / output race* (a race between input and output actions).

EIOA Composition and Related Races When an EIOA works in isolation, the races can occur within certain states, as discussed above. However, when the machine acts as a component of a multi-agent system, other types of races can take place. In particular, the communication channels can serve as 'tunnels' where the actions 'compete' to be faster for reaching the output, we refer to this type of races as *intra-channel* races. In this paper, we are concerned with the interactions of three entities, namely the application, controller, and switch. Therefore, potential races in the channels cover either the northbound or southbound interface. The interacting entities (application, controller and switch) are depicted in Fig. 1. Races of the latter type can occur for example if a message for deleting a rule is delivered to the switch after the rule expires via a timeout, i.e., the deletion is not performed as defined in the OpenFlow specification. We note, that it can also happen that due to races in the communication channel, both components can reach states where they wait for an input. If this happens, then a deadlock in the composition occurs as a consequence of such 'competition'.

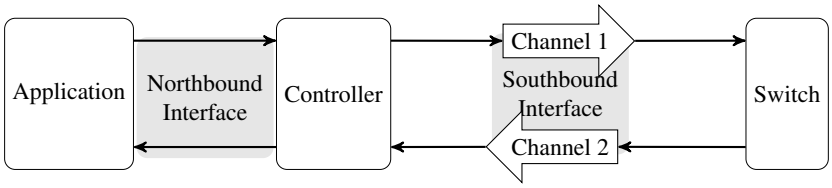


Fig. 1: SDN Topology considered in the paper

2.3 Linear Temporal Logic

For the race detection in EIOAs and their compositions, formal verification can be used and in this paper, we use Linear Temporal Logic (LTL) [6] formulas and the correlation between LTL formulas and EIOA properties.

The notion of an LTL formula is closely related to the Kripke structure [6]. A Kripke structure is a labeled directed graph where each state (node) s is labeled by a set $L(s)$ of *atomic propositions* which are $true$ (at s). Correspondingly, a path: $\pi = s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_j \rightarrow \ldots s_n \rightarrow \ldots$ is a possibly infinite sequence of consecutive states of the Kripke structure. For each $j \geq 1$, $\pi|^j$ is a suffix $s_j \rightarrow \ldots s_n \rightarrow \ldots$ of path π started with s_j.

An LTL formula is a formula $\varphi ::= a|\varphi_1 \wedge \varphi_2|\neg\varphi|\mathbf{X}\varphi|\varphi_1\mathbf{U}\varphi_2|\mathbf{F}\varphi|\mathbf{G}\varphi$, where: a is *an atomic proposition*, **X** denotes the 'next' operator, **U** denotes the 'until' operator,

F denotes the 'eventually' operator and **G** denotes the 'globally' operator. We briefly remind the conditions when a path $\pi = s_1 \to s_2 \to \cdots \to s_n \to \ldots$ *satisfies* ($\models$) the LTL formula φ:

- if $\varphi = a$ then $\pi \models \varphi$ iff $a \in L(s_1)$;
- if $\varphi = \mathbf{X}\varphi_1$ then $\pi \models \varphi$ iff $\pi|^2 \models \varphi_1$;
- if $\varphi = \mathbf{F}\varphi_1$ then $\pi \models \varphi$ iff there exists $i \geq 1$ such that $\pi|^i \models \varphi_1$;
- if $\varphi = \mathbf{G}\varphi_1$ then $\pi \models \varphi$ for all $i \geq 1$ $\pi|^i \models \varphi_1$;
- if $\varphi = \varphi_1 \mathbf{U} \varphi_2$ then $\pi \models \varphi$ iff there exist $i \geq 1$ $\pi|^i \models \varphi_2$ and for all $1 \leq j < i, \pi|^j \models \varphi_1$.

In this paper, atomic propositions are rather simple; an atomic proposition is *true* if and only if the value of a variable of interest is equal to a given integer.

3 Modeling the Application-Controller-Switch Interaction

In this section, we describe some of the EIOAs derived for checking the application-controller-switch interaction. We have derived the corresponding automaton for each of the interacting components and have elaborated a list of properties to be checked, described as corresponding LTL formulas. We note, that the chosen level of abstraction when modeling the SDN components' behavior plays a crucial role in their further verification.

We hereafter assume that the SDN infrastructure includes only one controller, one switch and one application. The latter means we abstract from any other entity on the data or control plane except the components of interest. To decrease the abstraction level in this case, a composition of switches or a whole data plane can be considered instead of a single switch. We however, note that even such restricted composition allows detecting the races of interest. At the same time, we consider two input channels for the controller, i.e., to accept inputs from the switch and the application of interest. Input and output alphabets of the components of interest can be defined in different ways. For the switch, for example, an OpenFlow specification provides the set of messages which the controller and the switch use for interacting while available specifications for the controllers of interest can also be considered. As a case study, for evaluating our methodology we chose the ONOS controller and thus, to extract the information about the controller-to-application interaction, we use ONOS documentation [2]. For the application of interest, in our experiments we consider a REST service [7] which allows defining the corresponding behavior of the application.

Given the assumptions listed above, we consider the topology shown in Figure 1. The ONOS documentation, the OpenFlow specification and ONOS REST service documentation allow to extract the following alphabets:

- An application-controller channel alphabet: $\{PostFlow, DeleteFlow, GetFlow\}$;
- A controller-application channel alphabet: $\{FlowRemoveExpire, ReplyFlow\}$;
- A controller-switch channel alphabet: $\{PostFlow, DeleteFlow, GetFlow\}$;
- A switch-controller channel alphabet: $\{FlowRemoveExpire, ReplyFlow\}$.

3.1 Extended Automaton Modeling the Behavior of a Controller

In this work, the EIOA for modeling the controller behavior has three states s_0, s_1, s_2. These states naturally correspond to the situations when messages from the controller to the switch (or vice-versa) change the behavior of the system. For example, at state s_0 a *GetFlow* (GF) request is not being processed; likewise, the *DelFlow* (DF) request cannot be produced. State s_1 describes the situation when a GF request has been produced from the controller (to the switch); at this state the EIOA can get *ReplyFlow* (RF) and should move to state s_2. At state s_2, the DF signal can be produced. We note that this interaction is a part of the aforementioned specifications (OpenFlow, ONOS, etc.).

The EIOA has a context variable; this variable corresponds to the number of installed flow rules, and thus, its domain is the set of integers $\{0,\ldots,n-1\}$. The EIOA has 11 transitions, and seven of them are unconditional, i.e., the transitions have no predicates. As an example, the transition $(s_0, GetFlow, s_1)$ models the situation when the controller has sent the *GetFlow* request moving to state s_1. The transition diagram of the EIOA modeling the controller behavior is shown in Figure 2.

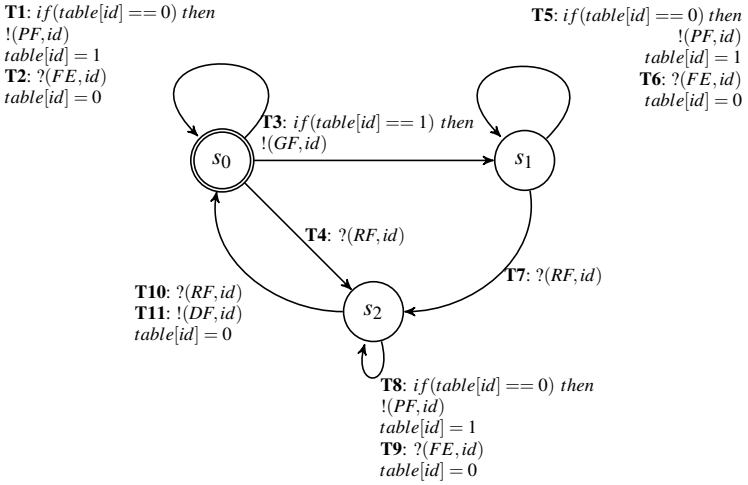


Fig. 2: EIOA modeling the controller behavior

For further model checking, namely races' detection, we describe the EIOA in Promela which is accepted by Spin as an input. In this case, the Promela code contains the variable `IDflow`, the rule's ID. Note that for convenience and without loss of generality the value of the `IDflow` is equal to the rule's priority. Moreover, the Promela description has an array `app_table` that contains all the current switch flow rules. The description of the controller behavior includes one process. A fragment[4] of the Promela description of the controller is illustrated in Figure 3. The complete version of the re-

[4] The actual description contains 632 lines.

lated program, as well as the Promela code for other interacting components, are available at [17].

```
proctype Controller(chan AppCont, ContApp, Ch2Cont,
                    ContCh1) {
    mtype mess_app_cont;
    int id_flow_app, id_flow_cont;
    S0:
        if
        :: AppCont?(mess_app_cont, id_flow_app);
           goto S0;
        :: (app_table[0] == false) ->
           atomic {
               do
               :: ContCh1!PostFlow(0);
                  app_table[0] = true;
                  mess = PostFlow;
                  flow_id = 0;
                  goto S0;
               :: Ch2Cont?FlowExpire(id_flow_cont);
                  ContApp!FlowExpire(id_flow_cont);
                  app_table[id_flow_cont] = false;
                  mess = FlowExpire;
                  flow_id = id_flow_cont;
                  goto S0;
               :: Ch2Cont?ReplyFlow(id_flow_cont);
                  ContApp!ReplyFlow(id_flow_cont);
                  mess = ReplyFlow;
                  flow_id = id_flow_cont;
                  goto S0;
               :: Ch2Cont?ReplyFlow(id_flow_cont);
                  ContApp!ReplyFlow(id_flow_cont);
                  mess = ReplyFlow;
                  flow_id = id_flow_cont;
                  goto S2;
               od;
}
```

Fig. 3: Controller description in Promela

3.2 Extended Automaton Modeling the Behavior of Channel 1

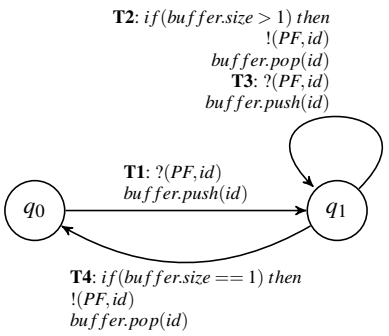
The EIOA that models channel 1's behavior has two states, namely q_0 and q_1. These states correspond to the state of the buffer. In particular, state q_0 corresponds to the situation when the buffer is empty while state q_1 reflects the opposite. The context variable `buffer.size` corresponds to the number of messages in the buffer. The EIOA has four transitions, and two of them are unconditional. As an example, a transition (q_1, PF, q_0) can be considered: it models the situation when channel 1 has sent a *PostFlow* request for a rule with ID *id* moving to state q_0. The transition diagram of the corresponding EIOA is shown in Figure 4a.

This EIOA is also described in (a fragment of) the Promela language. In this case, the Promela code contains the `IDflow` variable, the identifier for a given rule. Similarly, the Promela description has an array `buffer` that contains the messages in the buffer. A fragment of the Promela description of channel 1 (channel 2) is illustrated in Fig. 4b.

Given the room constraints, we do not include the detailed EIOA descriptions of other interacting components, such as the switch or application. For detailed description one can access the corresponding Promela code via [17].

4 Race detection in the Application-Controller-Switch Interaction

This section is devoted to the verification of the Application-Controller-Switch (ACS) interaction. We propose a methodology for detecting races of two types discussed above. Namely, we propose two randomized algorithms for two types of races. These are Monte Carlo algorithms, i.e., if a race is detected by the method then we guarantee that such race can happen in the ACS interaction under proper conditions, namely, for proper timeouts. However, if the algorithm returns '*FALSE*' then this answer can still be wrong, i.e., it cannot be guaranteed that the ACS interaction is free of races described



(a) EIOA model

```
1   proctype Channel1(chan ContCh1, Ch1Switch) {
2     S0: atomic {
3         ContCh1?PostFlow(channel1_flow_id);
4         buffer[channel1_flow_id] = true;
5         curr_size_buffer = curr_size_buffer + 1;
6         goto S1;
7       }
8     S1: if
9       :: atomic {
10          ContCh1?PostFlow(channel1_flow_id);
11          buffer[channel1_flow_id] = true;
12          curr_size_buffer = curr_size_buffer + 1;
13          goto S1;
14        }
15      :: (curr_size_buffer > 1) ->
16        if
17        :: (buffer[0] == true) ->
18          atomic {
19            Ch1Switch!PostFlow(0);
20            buffer[0] = false;
21            curr_size_buffer = curr_size_buffer - 1;
```

(b) Promela code

Fig. 4: Channel 1 descriptions

above. The main reasons for that are the following: i) the approach relies on the Spin model checker that does not guarantee that a counterexample is always found, and ii) the approach depends on the time each message spends in a channel and this time is nondeterministic.

4.1 Input / Output Races

These races can occur when there exists a state of a corresponding EIOA where an input and an output which have a parameter in common are allowed, i.e., at a given state the decision of accepting an input or producing an output is made nondeterministically. However, if the system is race insensible, then this fact does not influence the system behavior. In order to check this, the Spin model checker can be used. Therefore, the problem is reduced to constructing an appropriate LTL formula that checks whether the system is sensible to races between inputs and outputs at a given state.

We propose (a probabilistic approach in) Algorithm 1, that assures the presence and the detection of an input / output race. The algorithm is based on a choice of a state s of the EIOA, where both input i and output o *related via some parameter* are defined. The Spin model checker is used for verifying the LTL formula *prohibiting* a race between i and o. Let *curr_mess* denote an input or an output action (message) at state s while *next_mess* denote the action at the next time instance. To guarantee that the output o is never produced before the input i is received, an LTL formula of the following kind can be verified: $\mathbf{G}(\neg((curr\_mess == o) \rightarrow (next\_mess == i)))$ [5].

If the formula of interest can be violated for the component of interest, then Spin produces a counterexample α. Note that not each counterexample is feasible for the topology of interest; further checking should be performed by applying a sequence of inputs simulating α, with appropriate timeouts to an implementation (component of interest). Determining such timeouts is a separate task which for the moment is performed

[5] The reader might notice that the formula is quite generic and can be adjusted accordingly for various particular cases, for example, states, input / output parameters can be added.

heuristically for input / output races; while for intra-channel races (see Section 4.2), we propose an algorithmic solution (Algorithm 3). As the approach proposed in this paper is proactive, we execute the derived counterexample α potentially leading to a race against the implementation of the SDN framework. For that reason, we rely on an automated script; the script is executed at most N times, simulating α as an input to the component of interest. If during the execution of the script the *competition* between an input and output can be observed, then the script returns $TRUE$. Therefore, when $Execute\_script(\mathcal{S})$ returns $TRUE$ a race is indeed detected in the implementation. Note that the execution of the script can be performed against any application, controller, and switch, from different providers, as long as the corresponding versions are compatible.

Algorithm 1: Input / output race detection

Input : A composition Promela model $\mathcal{M}$; a number n of flow rules; a number N of testing iterations
Output: A Boolean value indicating if the race is detected
Choose an SDN component $\mathcal{C}$ of interest from the composition $\mathcal{M}$
Choose a state s of $\mathcal{C}$ where both, input i and output o are defined [6]
Derive an LTL-formula Φ simulating that the output o was produced before the input i was accepted
Verify Φ over $\mathcal{M}$ (e.g., using Spin)
if *a counterexample* α *is found (by Spin)* **then**
    if $Obtain\_timeout\_vector(\alpha) == FALSE$ **then**
        // *Obtain_timeout_vector* heuristically derives a timeout vector if it is feasible
        return *FALSE*
    Given the timeout vector, and the counterexample α, derive script $\mathcal{S}$ stimulating the SUT with α
    foreach $i \in \{1,\ldots,N\}$ **do**
        if $Execute\_script(\mathcal{S}) == TRUE$ **then**
            return $TRUE$
return *FALSE*

An important step of Algorithm 1 concerns the choice of the component of interest and its state (or a subset of those) for further verification. As an example, consider an SDN controller whose model is presented in Figure 2. Note that, at state s_2 both input (FE, id) and output (DF, id) are defined. These actions can compete and it must be assured that indeed (DF, id) occurs first. In other words, it should be checked that the controller can send a *DelFlow* message to remove a flow rule with the $flow\_id$ number only if this rule currently exists. This property can be expressed using the following LTL formula: $\mathbf{G}(((mess == DelFlow)\&\&(flow\_id == id)) \rightarrow (flow\_table[id] == true))$.

Algorithm 1 is a randomized algorithm that can be *trusted* on a race detection. The probability of such reply significantly depends on the structure of the specification

[6] The procedure can be repeated iteratively for a subset of states $J \subseteq S$.

EIOA **A**. If the simulation of this EIOA represents a Markov chain then the probability of success and, in some cases, its limit, can be evaluated.

Assume that the transitions of the EIOA **A** of interest are augmented with their probabilities of execution. In the simplest case, we assume that all the transitions are equally probable at a given state. Let Π denote the stochastic matrix for the related Markov process, i.e., $\Pi = \{p_{i,j}\}$, and $p_{i,j}$ is the probability to reach s_j from state s_i. Let also $J \subseteq S$ denote the set of *critical* states of **A** where a given input can compete with certain output. The probability of Algorithm 1 to terminate with a positive reply depends on the probability of reaching one of the states $j \in J$ from the initial state $s_0 \in S$. The latter is determined by the following proposition.

Proposition 1. *$Pr(s_1 \xrightarrow{\alpha} j \in J) = \sum_{j \in J} [p_{1,j}]^m$, where $\{[p_{i,j}]^m\} = \Pi^m$, and $|\alpha| = m$.*

Proof. Indeed, matrix Π^m consists of items $[p_{i,j}]^m$ for $i, j \in \{1, \ldots, |S|\}$. Moreover, state s_j is reached from state s_i via a sequence α, $|\alpha| = m$, i.e., $i \xrightarrow{\alpha} j$ with the probability $[p_{i,j}]^m$. Therefore, the probability of reaching any state of set J via a sequence of length m equals $\sum_{j \in J} [p_{1,j}]^m$. □

Corollary 1. *If the sum $\sum_{j \in J} [p_{1,j}]^m$ has the limit, then the probability of reaching a state where a race is possible equals $\lim_{m \to \infty} \sum_{j \in J} [p_{1,j}]^m$.*

The matrix Π^m can be computed by direct multiplication of Π matrices, or in some cases, the spectral decomposition of Π can be utilized (for more details one can see [15], for example), to improve the scalability of the Π^m as well as for the checking the criterion when such decomposition is possible.

We note also that the race sensibility in the SDN components of interest can be avoided by setting appropriate timeouts for processing inputs and producing outputs. However, as our experiments show (Section 5), for the ONOS controller those timeouts are not appropriately set.

4.2 Intra-channel Races

Having discussed the input / output races at a given state, we now turn to the detection of another type of races. In particular, this subsection is devoted to the intra-channel races, when for example the rules or requests submitted into a channel by a given component can be permutated. In other words, the order of the rules / requests can not necessarily be preserved once submitted into the channel. In the case of our experimental evaluation those are the rules submitted by the controller and later on pushed to the switch. Such detection is essential as it leads to different 'understanding' of a current network state at the control and data plane, and thus can cause network misconfigurations. In order to detect if there exists a potential permutation in the channel, we try to proactively create this race condition. To do so, the SDN application installs rules with their ID in chronological order; moreover, each rule has a hard timeout equal to its ID. For instance, the SDN application installs a rule with ID i and timeout i before installing the

rule $i+1$ with timeout $i+1$. Thus, to check the possibility of permutations, we propose Algorithm 4 that either detects an intra-channel race or returns *FALSE* if a race cannot be detected. The Promela model $\mathcal{M}$ of the composition of interest serves as one of the inputs. At the first step, the LTL formulas are derived via the call to Algorithm 2. The strategy to detect this type of races is to stimulate 'competing' messages in a channel. In order to better understand the properties that are investigated, we further explain the motivation behind them.

- Consider the following property: $\mathbf{G}((table[id] == true) \rightarrow (table[id'] == true))$. This guarantees that if the rule with the *id* number is pushed before the rule with id' number and has a timeout value less than that of id', then the *id* rule should be deleted before the id' one. In other words, the presence of the *id* rule in the flow table implies the presence of the id' as well.
- Consider the property: $(mess == (PF, id)) \rightarrow ((mess \neq (DF, id))\mathbf{U}(table[id] == true))$. This means that if a controller requested to push the rule with the *id* number, then at some point this rule will be pushed. Moreover, the rule cannot be deleted unless it was pushed.

Algorithm 2: Deriving LTL formulas

Input : The global variables of the Promela model ($\mathcal{V}_{\mathcal{M}}$)
Output: Φ, LTL-formulas to check intra-channel races
return *the following formulas*
foreach $i \in \{1, \ldots, size\_of(table) - 1\}$ **do**
  $\mathbf{G}((table[i+1] == 0) \rightarrow (table[i] == 0))$;
$\mathbf{G}((app\_send\_mess == (Post\_flow, id)) \rightarrow \mathbf{F}(table[id] == 1))$
$\mathbf{G}((app\_send\_mess == (Delete\_flow, id)) \rightarrow \mathbf{F}(table[id] == 0))$

At the next step, the derived properties together with the original model are 'fed' to Spin. If Spin does not detect a violation of any of the properties then not a single race of interest can be detected. This does not necessarily guarantee the absence of such races. On the other hand, if a counterexample α is produced then its feasibility is first verified (Algorithm 3). A non-feasible counterexample again results in the '*not detected*' conclusion. On the contrary, if α is feasible, the result is a vector τ representing the timeouts for each of the n rules. Similar to Algorithm 1, at the last step each feasible counterexample is executed N times as it is possible that the system does not incur into a race condition at all times. In this case, we analyze the probability of success to actually observe the race in the given implementation. In other words, given a counterexample α produced by Spin and a vector $\tau = (\tau_1, \ldots, \tau_n)$ of timeouts, we further define the probability of the randomized Algorithm 4 to terminate returning a *FALSE* verdict.

A message may spend an unknown time in a channel, we assume that this time is bounded by certain interval $[T_{min}, T_{max}]$; t_i defines the time that a message i spends in a channel, and there is no guarantee that the time spent in the channel is closer either to the left or to the right bound. Let us denote the timeout for a given rule i as

Algorithm 3: Permutation feasibility check

Input : T_{min} - the minimum time a message spends in a given channel; T_{max} - the maximum time a message spends in a given channel; n - the number of messages (rules) to submit to the channel

Output: A vector τ of timeouts or *FALSE* if races are not possible

Solve the following inequality[7]

$$t_i + \tau_i \geq t_j + \tau_j$$

where

$$t_i, t_j \in [T_{min}, T_{max}]$$
$$i < j$$
$$\tau_i < \tau_j$$
$$\tau_i > 0$$
$$\tau_j > 0$$

if *the inequality has a feasible solution* $(t_i', t_j', \tau_i', \tau_j')$ **then**
  select any $\tau = (\tau_1, \ldots, \tau_i', \ldots, \tau_j', \ldots, \tau_n)$ where
  $0 < \tau_1 < \tau_2 < \cdots < \tau_i' < \ldots < \tau_j' < \cdots < \tau_n$
  return τ
return *FALSE*

Algorithm 4: Intra-channel race detection

Input : A composition Promela model $\mathcal{M}$; A minimal and a maximal time T_{min}, T_{max} to cross the controller-to-switch Openflow channel by messages; A number n of $flow\_rules$; A number N of testing iterations

Output: A Boolean value indicating if the race is detected

Obtain a set of LTL-formulas Φ = Deriving_LTL_formulas ($\mathcal{V}_{\mathcal{M}}$)// $\mathcal{V}_{\mathcal{M}}$ `represents the global variables of` $\mathcal{M}$

Verify Φ over $\mathcal{M}$

if *a counterexample* α *is found* **then**
  if (τ == *Permutation_feasibility_check* (T_{min}, T_{max}, n)) **then**
    Obtain the script $\mathcal{S} = Produce\_script(\alpha, \tau)$
  else
    return *FALSE*
  foreach $i \in \{1, \ldots, N\}$ **do**
    if $Execute\_script(\mathcal{S}) == TRUE$ **then**
      return $TRUE$
return *FALSE*

τ_i, and the differences between the neighbour rule timeouts as d_i, i.e., $d_1 = \tau_2 - \tau_1$, $d_2 = \tau_3 - \tau_2$, …, $d_{n-1} = \tau_n - \tau_{n-1}$. Note that the rules in a switch are installed without permutations if and only if $\forall i \in \{1, \ldots, n-1\}$ $0 < \tau_i < \tau_{i+1}$, $t_i \in [T_{min}, T_{max}]$ it holds that $t_i + \tau_i < t_{i+1} + \tau_{i+1}$. The latter means that $\forall i \in \{1, \ldots, n-1\}$ $\tau_{i+1} - \tau_i > t_i - t_{i+1}$, i.e., $d_i > t_i - t_{i+1}$. We assume that for all i, t_i are absolutely continuous random variables defined over the interval $[T_{min}, T_{max}]$ with the probability density functions $f_i(z)$. We denote the length of this interval as $D = T_{max} - T_{min}$. We therefore assume that

t_i, $i \in \{1,\dots,n-1\}$, are independent and uniformly distributed on $[T_{min}, T_{max}]$, i.e., $\forall i \in \{1,\dots,n\}\ f_i(z) = \begin{cases} \frac{1}{D}, & z \in [T_{min}, T_{max}] \\ 0, & z \notin [T_{min}, T_{max}] \end{cases}$. In this case, $c_i = t_i - t_{i+1}$ are absolutely continuous random variables defined on the interval $[-D, D]$ with the probability density function $\varphi_i(z)$, for each $i \in \{1,\dots,n-1\}$. Taking into account the uniform distribution hypothesis for t_i and t_{i+1}, for all $i \in \{1,\dots,n-1\}$, c_i are also independent and uniformly distributed on the interval $[-D, D]$ with the probability density functions $\varphi_i(z) = \begin{cases} \frac{1}{2*D}, & z \in [-D, D] \\ 0, & z \notin [-D, D] \end{cases}$.

Proposition 2. *Given the uniform distribution hypothesis, the probability that no rules are permutated, i.e., no races occur in the channel, is equal to* $Pr(no\_permutations) = \frac{1}{(2*D)^{n-1}} * \prod_{i=1}^{n-1} (D + d_i)$.

Proof. According to the independence of random variables [19], it holds that: $Pr(no\_permutations) = Pr(d_1 > c_1, \dots, d_{n-1} > c_{n-1}) = Pr(c_1 < d_1) * \dots * Pr(c_{n-1} < d_{n-1}) = \int_{-D}^{d_1} \varphi_1(z)dz * \dots * \int_{-D}^{d_{n-1}} \varphi_{n-1}(z)dz$.

Under the assumption that $c_1, \dots, c_{n-1}$ are uniformly distributed, and $d_i \in [0, D]$, it holds that $Pr(no\_permutations) = \frac{1}{(2*D)^{n-1}} * \prod_{i=1}^{n-1} (D + d_i)$. □

Therefore, the probability of having races, caused by the rules' permutation in the channel is $Pr(permutations) = 1 - Pr(no\_permutations) = 1 - \frac{1}{(2*D)^{n-1}} * \prod_{i=1}^{n-1} (D + d_i)$.

We however note that the probability distribution for the time a rule spends in a channel is crucial. If it is known in advance the probability density function should be recalculated accordingly, for example, due to an experimental evaluation of the channel and corresponding communicating components. This affects the success of the randomized Algorithm 4, i.e., the positive reply when the races are detected. In our experiments, we however relied on the uniform distribution assumption and were able to detect the permutation and related races for the controller and switch of interest.

5 Experimental Results

Experiments were performed with the ONOS Controller and the Mininet [12] simulator, executed under a virtual machine running on VirtualBox Version 5.1.34 for Ubuntu 16.04 LTS with 4GB of RAM, and a quad AMD A6-7310 APU with AMD Radeon R4 Graphics processor. Our experimental setup corresponds to the topology in Figure 1, i.e., the simulated network contains a single switch running Open vSwitch version v2.11.0 and a single application implemented as a Perl [18] script. The goal of the

[7] Can be done using classical approaches, see for example Gauss-Jordan elimination [15].

experiments was to estimate the efficiency of the proposed algorithms for race detection and their impact on available SDN components of wide use, namely, on the communication between the ONOS controller and an Open vSwitch. All the Perl scripts utilized in the experimental setup are accessible at [17].

5.1 Input / Output Races

The first set of experiments was performed to detect the potential input / output races (at a given state). In this case, the component under test is the ONOS controller. With the proposed approach (Algorithm 1) we managed to detect a race at state s_2 for the specification provided in Figure 2. Consider the property: *an input* $(DelFlow, flow\_id)$ *should not be sent if the rule with the number* $flow\_id$ *is not defined in the flow table*. The corresponding LTL formula in this case is as follows: $\mathbf{G}(((mess == DelFlow) \wedge (flow\_id == id)) \rightarrow (flow\_table[id] == true))$.

A flow rule with a timeout with the number $flow\_id$ may expire in the switch before the controller sends the *DelFlow* request to remove the flow rule with this number. Thus, in such composition, the controller can be sensible to this type of races.

To detect such a race we used three rules ($n = 3$). The probability of success of the race detection can be estimated using the EIOA in Figure 3. Assume that the transitions at each state are equally probable, then the stochastic matrix $\Pi = \begin{bmatrix} 1/2 & 1/4 & 1/4 \\ 0 & 2/3 & 1/3 \\ 1/2 & 0 & 1/2 \end{bmatrix}$. The spectral decomposition allows computing $Pr(s_1 \xrightarrow{\alpha} s_3) = \lim_{m\to\infty} p^m_{1,3} = \frac{1}{3}$.

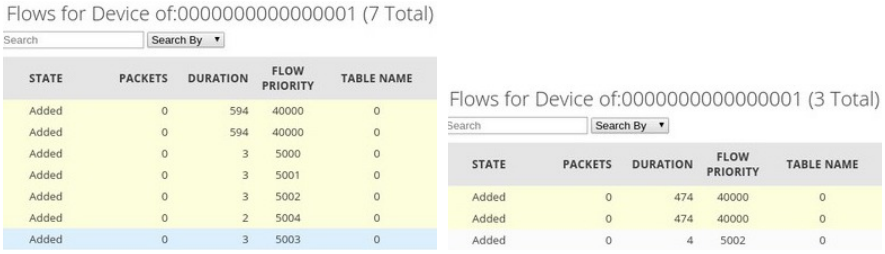
Given the probability of success, we executed the model checking and Spin found a counterexample (available at [17]) of length 21. To comply with the order of the requests in the counterexample α, following Algorithm 1 we derived a vector of timeouts τ that were implemented with the $sleep()$ procedure. The counterexample α (together with the values τ_i in the comments of the Perl script) can be checked at [17]. The script of interest was executed 10 times, i.e., $N = 10$ and at the 4-th iteration an unexpected behavior was observed. A rule deleted via a timeout, produced a $FlowExpire$ message from the switch to the controller, however, the controller at a later time instance produced a $FlowDelete$ message for the same rule ID. The latter means that the flow table *of the controller* can be dis-synchronized, i.e., the controller's knowledge of the network state is not always relevant and up-to-date. Moreover, it cannot be predicted which action removed the rule: $FlowExpire$ or $DelFlow$. The detailed logs of the script execution and race detection are also available at [17]. In the files `logs/test_1.txt`, `logs/test_2.txt` and `logs/test_3.txt` the $FLOW\_REMOVED$ signal is sent 5 times, however, in `logs/test_4.txt` the $FLOW\_REMOVED$ signal is sent 3 times. This means that to delete a rule with number 5000 in first 3 cases the controller sent $FLOW\_REMOVED$ signal and in the fourth case the switch sent a $FLOW\_EXPIRE$ message earlier. Thus, the input / output race is possible.

5.2 Intra-channel Races

When detecting races of this type in an SDN framework, we focused on the channel 1, i.e., the controller-to-switch interface. In particular, in our experiments we tried to detect a race between the $PostFlow$ requests in the related channel.

For that reason, Algorithm 4 was executed with the following parameters: $T_{min} = 0$, $T_{max} = 1.5$, $n = 5$, $N = 10$ and for each i, $d_i = 1$. Therefore $D = 1.5$, and thus, $Pr(no\_permutations) = \frac{2.5^4}{3^4}$. Consequently, $Pr(permutations) = 1 - \frac{2.5^4}{3^4} \approx 0,52$.

The counterexample α produced by Spin contains 10 requests (5 inputs and 5 outputs); the vector τ provided by Algorithm 3 is as follows: $\tau = \langle 5,6,7,8,9 \rangle$. Thus, for a rule with ID $5000 + i$, its timeout value is set to $i + 5$ seconds. Following Algorithm 4, the counterexample α was executed with the timeouts τ, using the corresponding Perl script. We executed the Perl script $N = 10$ times and at the 8-th execution an unexpected behavior was observed. Figure 5 showcases the flow table in the controller, which exhibits a race condition; Fig. 5a presents the rules as initially installed while Fig. 5b presents the rules at the end. Note that the rules 5003 and 5004 expired, however, the rule 5002 is still present, although the value of the timeout in the rule 5002 is set to 7 seconds while the timeout of the rules 5003 and 5004 are set to 8 and 9 seconds, correspondingly. Indeed, the nondeterministic behavior of the controller can lead to a rule permutation in the channel. In the counterexample of interest, two rules with the IDs 2 and 3 accordingly happened to permutate. The iterative execution of a Perl script allows to detect such permutations. Likewise, the permutation of rules with the numbers 5002 and 5003 can be observed (Figure 5). The latter means, that the sequence of actions executed by the controller can be different from the implemented sequence of actions.



(a) Initial rules (b) Final rules

Fig. 5: Rule permutation

6 Conclusions

In this paper, we considered concurrency issues in the SDN framework. In particular, we studied two different types of races for an application-controller-switch composition. For each of these types, we proposed a proactive testing approach for detecting such concurrency. The proposed approach complements the existing ones that are mostly based on effective monitoring and run-time verification, i.e., a model checking based design of test sequences that can push certain race to show, can be integrated into monitoring systems and thus, can later serve for the SDN components' certification.

This work opens a number of directions for future work. First, additional types of races should be considered, taking into account other types of interfaces, such as for

example between controllers. Despite the fact that the proposed approach is generic, experiments were only carried for the ONOS controller and one Open vSwitch, more experiments are needed to investigate other SDN components of wide use and their (in-)tolerance to certain types of races. The proposed approach relies on the LTL based model checking solutions for describing and detecting races in SDN; in the future, we plan to investigate other formal verification and model checking techniques and their applicability to the problem of interest. We plan to extend the proposed approach for stating clear recommendations for avoiding races in SDN frameworks, probably, taking advantage of some other model based techniques, such as for example Game Theory. Moreover, adaptive approaches are of a particular interest of the authors, namely we plan to investigate various testing strategies that could be used depending on the implemented race-avoidance mechanisms in an SDN framework. Finally, the application areas of the proposed approach are not limited with SDN, and in the future we plan to consider other distributed systems and related race conditions.

References

1. Baier, C., Katoen, J.P.: Principles of Model Checking. The MIT Press Cambridge, Massachusetts (2008)
2. Berde, P., Gerola, M., Hart, J., Higuchi, Y., Kobayashi, M., Koide, T., Lantz, B., O'Connor, B., Radoslavov, P., Snow, W., et al.: Onos: towards an open, distributed sdn os. In: Proceedings of the third workshop on Hot topics in software defined networking. pp. 1–6. ACM (2014)
3. El-Hassany, A., Miserez, J., Bielik, P., Vanbever, L., Vechev, M.T.: Sdnracer: concurrency analysis for software-defined networks. In: Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2016, Santa Barbara, CA, USA, June 13-17, 2016. pp. 402–415 (2016). https://doi.org/10.1145/2908080.2908124, https://doi.org/10.1145/2908080.2908124
4. Foundation, O.N.: Openflow switch specification version 1.4.0. https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.4.0.pdf (2013)
5. Holzmann, G.: The Spin model checker: primer and reference manual. Addison-Wesley Professional (2003)
6. Huth, M., Ryan, M.: Logic in Computer Science: Modelling and reasoning about systems. Cambridge university press (2004)
7. Koshibe, A., O'Connor, B., Milkey, R., Vachuska, T., Hall, J., Gebert, S., Higuchi, Y., Li, J., Hart, J., Lantz, B., Koti, S.P.: ONOS - Appendix B: REST API. https://wiki.onosproject.org/display/ONOS/Appendix+B%3A+REST+API (2014), last accessed: 2019-06-02
8. Kozierok, C.M.: The TCP/IP guide: a comprehensive, illustrated Internet protocols reference. No Starch Press (2005)
9. McClurg, J., Hojjat, H., Cerný, P.: Synchronization synthesis for network programs. In: Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II. pp. 301–321 (2017). https://doi.org/10.1007/978-3-319-63390-9_16
10. McKeown, N., Anderson, T., Balakrishnan, H., Parulkar, G., Peterson, L., Rexford, J., Shenker, S., Turner, J.: Openflow: enabling innovation in campus networks. ACM SIGCOMM Computer Communication Review **38**(2), 69–74 (2008)

11. Miserez, J., Bielik, P., El-Hassany, A., Vanbever, L., Vechev, M.T.: Sdnracer: detecting concurrency violations in software-defined networks. In: Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research, SOSR '15, Santa Clara, California, USA, June 17-18, 2015. pp. 22:1–22:7 (2015). https://doi.org/10.1145/2774993.2775004, https://doi.org/10.1145/2774993.2775004
12. de Oliveira, R.L.S., Schweitzer, C.M., Shinoda, A.A., Prete, L.R.: Using mininet for emulation and prototyping software-defined networks. In: 2014 IEEE Colombian Conference on Communications and Computing (COLCOM). pp. 1–6 (2014). https://doi.org/10.1109/ColComCon.2014.6860404
13. Scott, C., Wundsam, A., Raghavan, B., Panda, A., Or, A., Lai, J., Huang, E., Liu, Z., El-Hassany, A., Whitlock, S., H.B., A., Kyriakos, Z., Scott, S.: Troubleshooting blackbox sdn control software with minimal causal sequences. In: Proceeding of the ACM SIGCOMM 2014 Conference, Chicago, Illinois, USA (2014)
14. Shalimov, A., Zuikov, D., Zimarina, D., Pashkov, V., Smeliansky, R.: Advanced study of sdn/openflow controllers. In: 9th central & eastern european software engineering conference in russia. ACM (2013)
15. Strang, G.: Introduction to linear algebra, vol. 3. Wellesley-Cambridge Press Wellesley, MA (1993)
16. Sun, X.S., Agarwal, A., Ng, T.S.E.: Controlling race conditions in openflow to accelerate application verification and packet forwarding. IEEE Trans. Network and Service Management **12**(2), 263–277 (2015). https://doi.org/10.1109/TNSM.2015.2419975
17. Vinarskii, E.: Perl scripts, promela descriptions and counterexamples for sdn race detection. http://mks1.cmc.msu.ru/EvgeniiEM/detecting_SDN_races (2019)
18. Wall, L., Christiansen, T., Orwant, J.: Programming perl. O'Reilly Media, Inc. (2000)
19. William, F.: A Introduction to Probability Theory and Its Applications. Wiley (1971)
20. Zhang, Z., Yuan, D., Hu, H.: Multi-layer modeling of openflow based on efsm. In: 4th International Conference on Machinery, Materials and Information Technology Applications. pp. 209–214 (2016)