SDNRacer: Detecting Concurrency Violations in Software-Defined Networks

Jeremie Miserez[*]
miserezj@student.ethz.ch

Pavol Bielik[*]
pavol.bielik@inf.ethz.ch

Ahmed El-Hassany[†]
a.hassany@gmail.com

Laurent Vanbever[†]
lvanbever@ethz.ch

Martin Vechev[*]
martin.vechev@inf.ethz.ch

ETH Zurich
[*]Dept. of Computer Science
[†]Dept. of Information Technology and Electrical Engineering

ABSTRACT

Software-Defined Networking (SDN) control software executes in highly asynchronous environments where unexpected concurrency errors can lead to performance or, worse, reachability errors. Unfortunately, detecting such errors is notoriously challenging, and SDN is no exception.

Fundamentally, two ingredients are needed to build a concurrency analyzer: *(i)* a model of how different events are ordered, and *(ii)* the memory locations on which event accesses can interfere. In this paper we formulate the first happens-before (HB) model for SDNs enabling one to reason about ordering between events. We also present a commutativity specification of the network switch, allowing us to elegantly capture interference between concurrent events.

Based on the above, we present the first dynamic concurrency analyzer for SDNs, called SDNRACER. SDNRACER uses the HB model and the commutativity rules to identify concurrency violations. Preliminary results indicate that the detector is practically effective—it can detect harmful violations quickly.

Categories and Subject Descriptors

C.2.3 [**Computer-Communication Networks**]: Network Operations; D.2.4 [**Software Engineering**]: Software/Program Verification; D.2.5 [**Software Engineering**]: Testing and Debugging

General Terms

Software Defined Networking, OpenFlow, Commutativity Specification, Happens-before, Nondeterminism

1. INTRODUCTION

Software-Defined Networking (SDN) holds a great promise for managing network complexity. The key idea of SDNs is to enable *logically-centralized* and *direct* control of the forwarding behavior of a network. However, realizing this vision relies on the difficult task of building highly sophisticated and reliable SDN control software. Indeed, by design, control software operates on top of a highly asynchronous environment: events (e.g. packets) arriving at a switch, link or node failures, expiring flows can be dispatched to the controller at any time, all non-deterministically.

Highly asynchronous programs are prone to concurrency errors caused by interfering access (a form of data races) to shared memory locations [1], and SDN control software is no exception. However, it is well known that discovering those errors is difficult as they often depend on a particular ordering of specific events. At the same time, detecting these issues is important as they are often the root cause of deeper semantic problems (e.g., blackholes, forwarding loops or non-deterministic forwarding).

This paper present techniques for automatically detecting concurrency errors in SDNs. Conceptually, there are two places where interference can occur in SDNs: *(i)* within the control software itself (*e.g.*, if it is multi-threaded or distributed), and (ii) between the control software and the network switches. Indeed, network switches can be seen as memory locations which are *read* and *modified* by various events. The first kind of interference can be detected with standard approaches, e.g., [2]. Thus, in this short paper, we focus on techniques specific to the second kind.

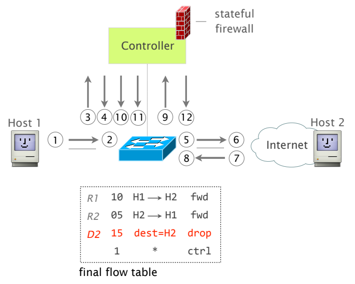
To capture asynchrony between a control program and the underlying network, we present the first formulation of a happens-before (HB) relation [3] for the most commonly used OpenFlow features. Our HB relation is based on an in-depth study of the OpenFlow specifications [4] as well as on the behavior of network switches [5]. The HB relation succinctly captures the partial ordering of events in an SDN.

In addition to the HB model, we present a *commutativity specification* of a network switch that precisely captures the conditions under which two operations on the switch commute. This specification elegantly abstracts the behaviors of the switch and is key to enabling precise analysis of the network.

Based on our models, we implemented SDNRACER, a dynamic and controller-agnostic concurrency analyzer for SDN networks and used it to detect various data races in real-world SDN applications. A manual inspection of a subset of the reported races confirmed that some of them were real bugs. The main contributions of this paper are:

- A *happens-before* (HB) model capturing the asynchronous interaction between an OpenFlow-based SDN controller and the underlying devices (§3).

SOSR2015, June 17 - 18, 2015, Santa Clara, CA, USA
© 2015 ACM. ISBN 978-1-4503-3451-8/15/06 ...$15.00.
DOI: http://dx.doi.org/10.1145/2774993.2775004



```
if ip_src in H:
 flow_mod(10,ip_src,
  ip_dst,fwd)
 flow_mod(10,ip_dst,
  ip_src,fwd)
 packet_out(pkt,sw)
else:
 flow_mod(15,ip_src,
  ip_dst,drop)
```

Figure 1: An example of stateful firewall application (right) and a sequence of events (left), which leads to a concurrency error which mistakenly *blocks* traffic from Host 2.

- A *commutativity specification* capturing the precise conditions under which two high-level operations on the network switch commute (§4).
- An implementation of a dynamic analyzer, called SDNRACER, which uses the HB model and the commutativity specification to automatically detect concurrency violations occurring on network switches.
- A preliminary evaluation indicating that the tool is able to uncover concurrency issues leading to harmful behaviors (*e.g.*, loss of reachability) (§5).

2. MOTIVATION

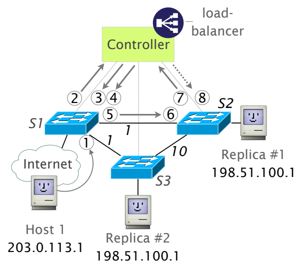
In this section, we explain in more detail how concurrency errors can arise in SDN with two motivating examples.

Sources of Races For our purposes, a SDN controller is an event-driven program in which events can occur both asynchronously (a packet received from the network, a link failure) or synchronously (as a result of a request issued by the controller). A SDN controller basically *writes to* and *reads from* the flow table of switches. The flow table of a switch is an ordered (by priority) list of forwarding entries against which packets are being matched and the corresponding forwarding action is taken. As such, a SDN switch can be thought of as a separate application that *reads from* the flow table whose state is written and queried by the controller.

In the following, we consider that a concurrency violation arises whenever we encounter two unordered accesses to the switch flow table, one of which must be a *write* produced by the controller.

Example#1. Stateful firewall *allows* traffic to be *blocked*. Consider a controller program running a stateful firewall, as shown in Fig. 1, that allows internal hosts to initiate communication with external hosts, but blacklists external hosts from sending unsolicited traffic to internal hosts.

A possible sequence of events is shown in Fig. 1. Host 1 sends a packet ① to Host 2 which hits the switch ② and is sent to the controller ③. Since the communication is initiated by internal host, the controller pushes down two `FLOW_MOD` rules and sends a `PACKET_OUT` message instructing the switch to forward the packet. Because the switch is allowed to execute messages out of order, it handles the `PACKET_OUT` message ④ first and sends ⑤ the packet further to Host 2. The Host 2 receives the packet ⑥ and responds immediately with a packet ⑦ that hits the switch back ⑧ *before* the rules have been installed. Consequently, the return packet goes to the controller ⑨. In the meantime, the two rules enabling the bi-directional communication are installed ⑩–⑪. As the return packet comes from Host 2, the controller instructs the switch to install a drop rule ⑫ which drops the communication.



```
if dst == server:
 _rep = rep[idx]
 idx = (idx+1)%2
 install_path(src,_rep)
 packet_out(pkt,in_sw)
...
def install_path(s,d):
 path =  dijkstra)(s,d)
 for i in path:
  flow_mod(i,s,d,
   fwd(p[i+1]))
```

Figure 2: An example of a simple load-balancing application (right) and a sequence of events (left), which leads to a forwarding loop.

In this example, there exists a concurrency error due to non-deterministic order between the *write event* ⑪, the installation of the rule matching packets from Host 2, and the *read event* ⑧, the reception by the switch of the return packet from Host 2. A simple fix is for the controller to issue a `BARRIER_REQUEST` message after the two rule installation requests and before sending the packet out to Host 2.

Example#2. A non-deterministic forwarding loop in a load balancer. In this example we consider a controller that is running a simple load-balancer application, Fig. 2.

Consider the sequence of events shown in Fig. 2: Host 1 sends a request directed to a farm of web server replicas identified by the IP address `198.51.100.1`. That request hits the switch ① and is sent to the controller ②. The controller elects Replica#1, computes the shortest-path between $S1$ and $S2$, pushes down two `FLOW_MOD` on $S1$ and $S2$, and sends a `PACKET_OUT` message instructing $S1$ to forward the packet to $S2$. $S1$ installs the rule ③ and handles the message `PACKET_OUT` ④ that it sends to $S2$ ⑤. The packet hits $S2$ ⑥ before the corresponding flow rule is installed ⑧ and is sent back to the controller ⑦. Assuming a round-robin selection algorithm, the controller now elects Replica#2, computes the shortest-path between $S2$ and $S3$ and pushes down the corresponding flow rules on $S2$, $S1$ and $S3$. From this point on, the traffic is being processed in a non-deterministic manner as $S1$ and $S2$ each have two rules with the same priority that match each direction of the traffic. Concretely, the traffic either ends up in a forwarding loop, if $S1$ uses the rule to forward the traffic to $S2$, and vice-versa or hits one of the two replica (again non-deterministically).

In this example, the concurrency error arises between the *read event* ⑥, the outbound packet received by $S2$ and the *write event* ⑧ on $S2$ matching it.

3. CAPTURING THE ASYNCHRONY

We present a formal model capturing the asynchrony arising in OpenFlow. The formal model is defined in terms of two core building blocks. Each of these blocks is a required component for developing a concurrency analyzer:

- A definition of the (potentially concurrent) operations and atomic events.
- A definition of the happens-before (HB) relation between the events.

3.1 Defining Operations and Events

We begin by defining a small set of events, denoted as *Event*, which succinctly encapsulate the relevant operations performed by network switches and hosts in the network. The operations are defined in §4.2 and contain the *reads* and *writes* (updates) to the flow

PACKETS:

$$\frac{\alpha \in \{PacketHandle, MsgHandle, PacketSend, HostSend\} \quad \beta \in \{PacketHandle, HostHandle, PacketSend\} \quad \beta.pid\_in \in \alpha.pids\_out}{\alpha \prec \beta}$$

MESSAGES:

$$\frac{\alpha \in \{PacketHandle, MsgHandle, MsgSend\} \quad \beta \in \{MsgHandle, MsgSend\} \quad \beta.mid\_in \in \alpha.mids\_out}{\alpha \prec \beta}$$

HOST:

$$\frac{\alpha \in HostHandle \quad \beta \in HostSend \quad \alpha.pids\_out = \beta.pid\_in}{\alpha \prec \beta}$$

FLOWREMOVED:

$$\frac{\alpha = MsgHandle \quad \alpha.msg\_type = \texttt{FLOW\_MOD} \quad \beta = MsgSend \quad \beta.msg\_type = \texttt{FLOW\_REMOVED} \quad \alpha <_\pi \beta \quad \alpha.switch\_id = \beta.switch\_id \quad \alpha.\texttt{match} = \beta.\texttt{match} \quad \alpha.\texttt{cookie} = \beta.\texttt{cookie} \quad \alpha.\texttt{priority} = \beta.\texttt{priority}}{\alpha \prec \beta}$$

BARRIERPOST:

$$\frac{\alpha, \beta \in MsgHandle \quad \alpha.switch\_id = \beta.switch\_id \quad \alpha.msg\_type = \texttt{BARRIER\_REQUEST} \quad \alpha <_\pi \beta}{\alpha \prec \beta}$$

BARRIERPRE:

$$\frac{\alpha, \beta \in MsgHandle \quad \alpha.switch\_id = \beta.switch\_id \quad \beta.msg\_type = \texttt{BARRIER\_REQUEST} \quad \alpha <_\pi \beta}{\alpha \prec \beta}$$

Figure 3: Happens-before rules capturing ordering of packets and OpenFlow messages for a trace π.

table. For each event, we define a set of attributes that describe the event and are later used to build the HB ordering. The set of attributes is as follows:

$$\langle pid\_in, pids\_out, mid\_in, mids\_out, msg\_type, switch\_id\rangle$$

where pid_in, $pids\_out$ denote identifiers assigned to packets and mid_in, $mids\_out$ denote identifiers assigned to sets of OpenFlow messages. The msg_type is an OpenFlow message type. The relevant message types for concurrency analysis are `PACKET_IN`, `PACKET_OUT`, `BARRIER_REQUEST`, `PORT_MOD`, `FLOW_REMOVED` and `FLOW_MOD`. Finally, $switch\_id$ is a switch identifier. The out keyword for identifiers denotes that the instrumentation always generates a new identifier whereas the in keyword means that a previously generated identifier is used. If an event generates no further events (*e.g.* simply the flow table is updated) the attributes $pids\_out$ and $mids\_out$ are initialized to the empty set $\varnothing$. Similarly, pid_in and mid_in are initialized to the undefined value $\perp$ if no existing packet or message was processed. Depending on the event type, only a subset of attributes is used. Following events capture the behavior of the switches, hosts and controllers:

PacketHandle($pid\_in, pids\_out, mids\_out$**)** denotes that a switch processed a data plane packet pid_in. As a result of processing this packet, either an OpenFlow message is generated and sent to the controller (in which case $mids\_out$ contains a message identifier and $pids\_out$ the identifier of the packet stored in the buffer) or the packet is forwarded (in which case $pids\_out$ contains the new packet identifier).

MsgHandle(mid_in**,** pid_in**,** $pids\_out$**,** $mids\_out$**,** msg_type**)** denotes that switch processed the OpenFlow message mid_in with type msg_type. The pid_in attribute is set to $\perp$ unless a packet is read from the switch buffer. The mid_in attribute is filled in by the controller instrumentation.

PacketSend($pid\_in, pids\_out$**)** denotes that the packet pid_in was sent out to the data plane with the identifier in $pids\_out$.

MsgSend(mid_in**,** $mids\_out$**)** denotes that the OpenFlow message with mid_in was sent to the controller with the new identifier in $mids\_out$.

HostHandle($pid\_in, pids\_out$**), HostSend(**$pid\_in, pids\_out$**)** denote packet receive and send by a host respectively.

3.2 Defining the Happens-Before Ordering

Having defined what the relevant events for concurrency analysis are, we now define a happens-before (HB) relation between them. The HB relation denoted as $\prec \;\subseteq\; Event \times Event$ is a binary relation that is irreflexive and transitive. For convenience, we use notation $\alpha \prec \beta$ instead of $(\alpha, \beta) \in \prec$. For each finite trace consisting of a sequence of events $\pi = \alpha_0 \cdot \alpha_1 \cdot \cdots \cdot \alpha_n$ we use $\alpha <_\pi \beta$ to denote that event α occurs before event β in π. We formalize the HB rules for a given trace π in Fig. 3.

PACKETS This rule orders events that send a packet before events that receive the packet. Each event handling a packet pid_in generates new, unique $pids\_out$. This guarantees that there is at most one pair in the trace for which $pid\_in = pids\_out$. There could be no pair in case the packet is dropped. In Fig. 1, this rule introduces the orderings ① $\prec$ ②, ④ $\prec$ ⑤, ⑤ $\prec$ ⑥ and ⑦ $\prec$ ⑧.

MESSAGES This rule orders messages sent to the controller from the switch with the responses the controller sends back and the other way around. Note that there could be multiple $MsgHandle$ with the same mid_in in case the controller sends multiple messages in the response. The controller instrumentation captures the mapping from $MsgSend$ to $MsgHandle$. In Fig. 1, this rule introduces the ordering ② $\prec$ ③, ③ $\prec$ ④, ③ $\prec$ ⑩, ③ $\prec$ ⑪, ⑧ $\prec$ ⑨ and ⑨ $\prec$ ⑫.

BARRIER For performance reasons, the switch is allowed to handle messages received from the controller in a different order from the one they were sent. To enforce ordering, the controller can issue a `BARRIER_REQUEST` message which ensures that the network switch finishes processing of all previously received messages, before executing any messages beyond the `BARRIER_REQUEST`.

HOST There is an ordering between the host receiving a packet and responding to it. We note that the implementation of this rule is speculative as we treat the host as a black box that can run arbitrary applications or protocols. In Fig. 1, this rule introduces the ordering ⑥ $\prec$ ⑦.

FLOWREMOVED This rule captures the fact that the switch can send a `FLOW_REMOVE` message only after the corresponding flow was added to the table using the `FLOW_MOD` message.

Key Points Our goal was to model the ordering of the events as precisely as possible, while at the same time having a succinct and precise model. For example, one could simply define the events at finer granularity (such as write packet to a buffer or a read packet from buffer). However, such a definition would be more difficult and cumbersome to work with as it would expose internal implementation details of the switch (that might differ between implementations) and contain more events and happens-before rules.

4. COMMUTATIVITY SPECIFICATION

In this section we provide a complete commutativity specification for the network switch based on the OpenFlow specification 1.0 [4]. We first informally discuss the operation of an OpenFlow switch and then introduce the notation used throughout the rest of the section.

4.1 Flow Table: Entries

A basic component of each OpenFlow switch is the flow table. This table is responsible for performing packet lookups and packet forwarding. The flow table contains a set of entries used to match incoming packets.

Packet. The packet contains a *header* and a *payload*. The *header* consists of a set of fields (*e.g.*, IP source, IP destination or VLAN id) used to match packets against flow table entries. The *payload* is a sequence of bits and does not affect our specification (discussed later). For a packet pkt we use the notation $pkt.h$ to refer to the header associated with pkt.

Flow Table Entry. The flow table entry contains the fields *match*, *priority*, *counters* and *actions*. The *match* can be either an exact match or a wildcard match. *Priority* is a number specifying entry preference in case the packet matches multiple flow entries. *Counters* are used for statistics and *actions* specify a set of forwarding operations to be performed on a matching packet.

For a flow table entry e we use the notation $e.m$, $e.p$ and $e.a$ to refer to the *match*, *priority* and *actions* respectively.

A match between two entries e_1 and e_2 is exact, denoted as $e_1.m = e_2.m$, when all *match* fields are exactly the same (including the wildcards). A match between e_1 and e_2 is wildcard, denoted as $e_1.m \subseteq e_2.m$, if some of the fields in $e_1.m$ are not an exact match but contained in $e_2.m$ due to more permissive wildcards. The same definition of wildcard and exact match applies to packet and flow table entry.

4.2 Flow Table: Operations

There are four types of operations that can be performed on the flow table. A $read$ operation is performed on each received packet while add, mod and del are issued by the controller using the `FLOW_MOD` command. In our work we use the OpenFlow specification 1.0 [4] to define the semantics of all of the above operations.

$read(pkt)/e_{read}$: The $read$ operation denotes that a packet pkt is matched against the flow table to determine the highest priority flow table entry e_{read} that should be applied. If there is no such flow table entry, e_{read} is set to the empty value $none$. Note that the value of e_{read} depends on the state of the flow table against which the packet pkt is being matched.

$add(e_{add}, no\_overlap)$: An add operation tries to add a new entry e_{add} to the flow table. If $no\_overlap$ is true then a new entry is not added if a single packet may match both the new entry and an entry already in the flow table, and both entries have the same priority.

$mod(e_{mod}, strict)$: A mod operation modifies existing entries in the flow table. A boolean flag $strict$ is used to distinguish between two types of modifications issued by the controller – `MODIFY` and `MODIFY_STRICT`. In strict mode, exact match (including the priorities) is used to determine whether an entry should be modified whereas in non-strict mode a wildcard match is used. Note that mod is also allowed to add entry in case no match is found.

$del(e_{del}, strict)$: A del operation deletes all entries that match the entry e_{del} in the flow table. Similarly to the mod operation, $strict$ affects how the matching is performed.

4.3 Flow Table: Commutativity

Intuitively, the commutativity captures whether changing the order of two operations affects the computation result. The computation results include relevant flow table state together with the return values (if any) of the participating operations. We consider two flow tables to be in the same state if they contain identical flow table entries, except for counters which are ignored.

The commutativity specification is conveniently specified in the form of a predicate φ over pairs of operations using formulas written in propositional logic. For a pair of operations a and b, the predicate φ_b^a evaluates to $true$ if operations commute and to $false$ otherwise.

Auxiliary Relations. To avoid clutter we define three auxiliary functions. First, we overload the set intersection operator $e_1 \cap e_2$ for two entry match structures (or packet headers) and use it to compute all packet headers than may match both. Next, we use $e_1 \overset{strict}{\subseteq} e_2$ to model the semantics of the table entry matching in the $strict$ mode, defined as follows:

$$e_1 \overset{strict}{\subseteq} e_2 \;:=\; \begin{array}{ll} e_1.m = e_2.m \wedge e_1.p = e_2.p & \text{if } strict \\ e_1.m \subseteq e_2.m & \text{if } \neg strict \end{array}$$

A $deletes$ predicate models the semantics of a delete operation and specifies whether an entry e can be deleted:

$$\begin{array}{l} deletes(e_{del}, e, strict) := \\ \quad e \overset{strict}{\subseteq} e_{del} \wedge e.\texttt{out\_port} \subseteq e_{del}.\texttt{out\_port} \end{array}$$

Commutativity Specification. The commutativity specification of an OpenFlow switch is shown in Fig. 4. All of the rules are written in the form that specifies when the operations do not commute which is then negated. We adopt this approach as the resulting rules are more intuitive to read. What follows is a description of some of the non-trivial rules.

$\varphi(add, add)$**:** Adding two entries does not commute if: i) the second entry overwrites the first one, or ii) the second entry is not added because the first entry is already in the table. The entries can overwrite each other only if both are added without $no\_overlap$ option and their $match$ and $priority$ is identical. In this case the old entry is replaced with the new one and as long as their actions are different they do not commute. If at least one entry specifies the $no\_overlap$ option, then they do not commute if they have the same $priority$ and there exists an entry that can be matched by both entries.

$\varphi(add, mod)$**:** In case the $no\_overlap$ option is not set, the add and mod do not commute in cases when they are allowed to modify the same entry with different actions. If $no\_overlap$ is set, then the mod can add a new entry that overlaps with add which would result in add not being added.

$\varphi(del, mod)$**:** If modify affects only a single entry ($strict$ mode), we simply check whether this entry can be deleted. Otherwise, as long as both rules can match the same entry, they do not commute.

$\varphi(add, del)$**:** The add and del do not commute if: i) the added entry can be removed by a subsequent delete, or ii) the delete does not remove the entry to be added but might enable adding it by removing some other entries. This situation arises when headers that may match add and del overlap.

$\varphi(mod, mod)$**:** If neither modify operation uses $strict$ mode then they do not commute if there is an entry that may match both.

$$\varphi^{read(pkt)/e_{read}}_{add(e_{add},\, no\_overlap)} := \begin{cases} \neg(e_{read} \neq none \wedge e_{read} = e_{add}) & \text{if } add < read \\ \neg(pkt.h \subseteq e_{add}.m \wedge (e_{read} = none & \text{if } read < add \\ \quad \vee(e_{read}.p \leq e_{add}.p \wedge e_{read}.a \neq e_{add}.a))) & \end{cases}$$

$$\varphi^{read(pkt)/e_{read}}_{mod(e_{mod},\, strict)} := \begin{cases} \neg(e_{read} \neq none \wedge e_{read} \overset{strict}{\subseteq} e_{mod} \wedge e_{read}.a = e_{mod}.a) & \text{if } mod < read \\ \neg(e_{read} \neq none \wedge pkt.h \subseteq e_{mod}.m \wedge e_{read}.a \neq e_{mod}.a) & \text{if } read < mod \end{cases}$$

$$\varphi^{read(pkt)/e_{read}}_{del(e_{del},\, strict)} := \begin{cases} \neg(pkt.h \subseteq e_{del}.m) & \text{if } del < read \\ \neg(e_{read} \neq none \wedge deletes(e_{del}, e_{read}, strict)) & \text{if } read < del \end{cases}$$

$$\varphi^{del(e_{del},\, strict_{del})}_{mod(e_{mod},\, strict_{mod})} := \begin{cases} \neg(deletes(e_{del}, e_{mod}, true)) & \text{if } strict_{mod} \\ \neg(e_{del}.m \cap e_{mod}.m \neq \emptyset) & \text{otherwise} \end{cases}$$

$$\varphi^{add(e_{add},\, no\_overlap)}_{del(e_{del},\, strict)} := \neg(deletes(e_{del}, e_{add}, strict) \vee (no\_overlap \wedge e_{add} \cap e_{del} \neq \emptyset))$$

$$\varphi^{mod(e_1,\, strict_1)}_{mod(e_2,\, strict_2)} := \begin{cases} \neg(e_1.m \cap e_2.m \neq \emptyset \wedge e_1.a \neq e_2.a) & \text{if } \neg strict_1 \wedge \neg strict_2 \\ \neg(e_1.m = e_2.m \wedge e_1.p = e_2.p \wedge e_1.a \neq e_2.a) & \text{if } strict_1 \wedge strict_2 \\ \neg((e_1 \overset{strict_2}{\subseteq} e_2 \vee e_2 \overset{strict_1}{\subseteq} e_1) \wedge e_1.a \neq e_2.a) & \text{otherwise} \end{cases}$$

$$\varphi^{add(e_{add},\, no\_overlap)}_{mod(e_{mod},\, strict)} := \begin{cases} \neg(e_{add} \overset{strict}{\subseteq} e_{mod} \wedge e_{add}.a \neq e_{mod}.a) & \text{if } \neg no\_overlap \\ \neg(e_{add} \cap e_{mod} \neq \emptyset) & \text{otherwise} \end{cases}$$

$$\varphi^{add(e_1,\, no\_overlap_1)}_{add(e_2,\, no\_overlap_2)} := \begin{cases} \neg(e_1.m \cap e_2.m \neq \emptyset \wedge e_1.p = e_2.p) & \text{if } no\_overlap_1 \vee no\_overlap_2 \\ \neg(e_1.m = e_2.m \wedge e_1.p = e_2.p \wedge e_1.a \neq e_2.a) & \text{otherwise} \end{cases}$$

Figure 4: Commutativity specification of an OpenFlow switch. Two $read$ and two del operations always commute.

If they are both *strict* then this entry needs to be exactly the same. Otherwise they do not commute if they are allowed to change the entry of each other.

$\varphi(read, add/mod/del)$: For $read$ operations we distinguish two cases depending on the order in which the operations are executed in the trace. If the $read$ happens first, the operations do not commute if the matched entry is not guaranteed to match after second operation is performed. Since we know the concrete flow entry that matched the initial read, such check can be performed precisely. In the case of a $read$ executing second, we simply check whether the matched rule is identical to the one added or modified. For the delete operation, we conservatively check whether an entry that matches the packet can be removed.

Key Points. Note, that for the $read$ operations our commutativity specification incorporates parts of the flow table state by using the return values. Further, commutativity rules for $read$ are specialized based on the trace order, which is a direct consequence of depending on the state in which the operations were performed. This allows us to significantly reduce the number of reported conflicts by not including operations that commute in the current flow table state but might not necessarily commute in all possible states.

5. IMPLEMENTATION

Our implementation of SDNRACER consists of two parts: *i)* an instrumentation of STS [6], a SDN troubleshooting system and network simulator, and *ii)* a concurrency analyzer that implements the happens-before rules and commutativity checks.

STS and Controller Instrumentation STS contains a network simulator that simulates switches and hosts. We extended the simulator with extra instrumentation which tracks the path that each packet takes through each of the switches and hosts, and records all operations that interact with the flow table. Due to having full insight into the switch, it is possible to accurately keep track of packet modifications that would otherwise be difficult to capture precisely. To capture controller related happens-before we instrumented two mainstream OpenFlow controllers: Floodlight [7] and POX [8]. The instrumentation includes a wrapper around the event handler for incoming messages, and adding happens-before relations automatically whenever a message is sent, thus capturing the order between $MsgSend$ and $MsgHandle$ events. Our current approach does not capture happens-before edges introduced by explicit synchronization used inside a controller. As such, an edge can only be added if messages are sent by the event handler itself, and not at a later time.

Types of Controllers The approach used is sufficient for reactive controllers using a single thread per switch connection such as the ones used by our examples. Messages sent out by the controller are always a direct result of handling an incoming OpenFlow message, thus appropriate edge can be added by the instrumenation. However, SDNRACER can not make the connection between the messages received by the controller and the messages sent by it for multithreaded, distributed, or proactive controllers. Thus, erroneous *read-write* conflicts are detected between each write and all reads in the trace, although they might not represent an actual issue. Still, *write-write* races introduced by missing barriers can still be properly detected even for with proactive controllers. To extend SDNRACER to completely handle all types of controllers will require extending the HB rules to fully capture what happens inside each controller implementation, and adding further instrumentation to capture this behaviour.

Concurrency Analyzer The analysis is performed by constructing a directed graph capturing the happens-before relations from all events in the current trace. Given the happens before graph we check all flow table *reads* and *writes* for possible high level conflicts consisting of two unordered accesses to the same flow table, one of which must be a *write*. For each of these high level conflict, we use the commutativity specification to exclude pairs of operations that commute. We then report the remaining, potentially concurrency conflicts to the user. SDNRACER runs the analysis offline due to the fact that the STS and controller instrumentations are currently not synchronized. However, there is nothing inherent in the analysis itself that would prevent it from running online.

6. EVALUATION

We have successfully used SDNRacer to find concurrency errors in both controllers implemented in Floodlight using the examples described in §2.

Running the simulation shown in Fig. 1 (Example#1) we observe three high level concurrency violations: ⑩–⑪, ⑧–⑩ and ⑧–⑪. Out of these three, two commute: ⑩–⑪ commute as the two `FLOW_MOD` events match disjoint set of packets, and ⑧–⑪ commute as the rule in ⑪ applies only to packets originating from Host 1. The only concurrency violation ⑧–⑩ which does not commute is reported to the user and it describes exactly the harmful scenario in which the switch receives the packet before it installs the `FLOW_MOD` ⑩. Fixing the controller by inserting a `BARRIER_REQUEST` after the two `FLOW_MOD` messages reduces the number of high level concurrency violations to a single one (between the two `FLOW_MOD` messages) which commutes, thus SDNRacer ceases to report violations.

Similarly for the Example#2 we report only a single concurrency violation between events ⑥ and ⑧, corresponding to packet and `FLOW_MOD` message hitting the $S2$ respectively. It should be noted that thanks to the happens-before model SDNRacer was able to discover these concurrency errors despite the fact that both examples were tested on traces which did not exhibit the harmful scenario (e.g., the `FLOW_MOD` message was processed before the data plane packet arrived).

7. RELATED WORK

Several research projects aimed at verifying correctness of SDN networks: HSA [9, 10] and Libra [11] take snapshots of the network forwarding state and check if they violate certain properties. Like SDNRacer, these tools can detect interesting invariant violations. However, they cannot tell what precise sequence of events led to them. STS [6] extended these works by also considering the events that caused the controller to trigger invariant violations. Unlike SDNRacer, STS does not have a precise formal specifications of the partial orderings between events or the conditions under which two operations commute. As a result it cannot detect bugs unless the invariant is actually violated in the given trace.

Several approaches seek to eliminate the possibility of bugs altogether, by synthesizing controllers that can be proven to be correct: Machine verified controllers [12] can be proven to correctly implement the network behaviour specified by policies written in NetCore [13]. A similar approach is taken by FlowLog [14], where rulesets are partially compiled to NetCore policies and then verified. Instead of verifying the controller, VeriFlow [15] and Anteater [16] set a middle layer between the controller and the data-plane to make sure that no invariant is violated at runtime. An extension [17] of VeriFlow allows using assertions to check network properties during controller execution.

A different approach to the verification problem is taken by tools such as NICE [18], Kuai [19] and Vericon [20]. NICE [18] uses concolic execution of Python controller programs with symbolic packets and then runs a model checker to determine invariant violations. A simplified model is used for the OpenFlow switch in order to reduce the number of states that need to be explored. The Kuai [19] checker similarly uses a simplified version of an OpenFlow switch as well as a custom controller language, but then applies partial order reduction techniques to reduce the number of states the model checker has to explore. Although performance is significantly improved, it still suffers from the state-space explosion problem associated with full model checking. Vericon [20] converts programs into first-order logic formulas and uses a theorem prover to verify safety properties. One interesting aspect is that Vericon can prove that the invariants hold under network topologies of any size, without having to explicitely test each topology. In contrast, SDNRacer is a dynamic analyzer that operates on concrete traces of real controllers and can quickly detect concurrency issues, the root cause of many bugs. The speed of the analysis is independent of controller size and only depends on the size of the trace, which is important when analyzing real-world controllers. The approaches above could benefit from our formal specifications in order to speed-up their verification time (*e.g.*, by not checking operations that do not interfere with the current network state).

8. CONCLUSION

In this paper, we introduced the first happens-before relation for Software Defined Networks capturing ordering between concurrent events. We also introduced a commutativity specification of the switch enabling precise analysis of the network.

Based on these two ingredients, we developed SDNRacer, the first dynamic analyzer able to detect violations occurring between an SDN controller and the underlying network switches. Our first prototype of SDNRacer is promising: it is able to detect races in real-world SDN applications, including harmful ones capable of causing anomalies such as loss of reachability.

In future work we plan to: *i)* extend state aware commutativity checks to other operations, *ii)* perform more complex analysis of the controller to add happens-before orderings beyond the currently supported reactive mode, and *iii)* explore stateless model checking to ensure better coverage.

9. REFERENCES

[1] V. Raychev, M. Vechev, and M. Sridharan. Effective race detection for event-driven programs. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages &#38; Applications*, OOPSLA '13, pages 151–166, New York, NY, USA, 2013. ACM.

[2] C. Flanagan and S. N. Freund. Fasttrack: Efficient and precise dynamic race detection. In *Proceedings of the 2009 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '09, pages 121–133, New York, NY, USA, 2009. ACM.

[3] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Communications of the ACM*, 21(7):558–565, 1978.

[4] OpenFlow Switch Specification. Version 1.0.0. `https://www.opennetworking.org/images/stories/downloads/sdn-resources/onf-specifications/openflow/openflow-spec-v1.0.0.pdf`.

[5] Open vSwitch. Production Quality, Multilayer Open Virtual Switch. `http://openvswitch.org/`.

[6] C. Scott, A. Wundsam, B. Raghavan, A. Panda, A. Or, J. Lai, E. Huang, Z. Liu, A. El-Hassany, S. Whitlock, H. Acharya, K. Zarifis, and S. Shenker. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 395–406, New York, NY, USA, 2014. ACM.

[7] Floodlight Controller. `http://projectfloodlight.org/floodlight`.

[8] J. Mccauley. POX: A Python-based OpenFlow Controller. `http://www.noxrepo.org/pox/about-pox/`.

[9] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, NSDI'12, pages 9–9, Berkeley, CA, USA, 2012. USENIX Association.

[10] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 99–112, Berkeley, CA, USA, 2013. USENIX Association.

[11] H. Zeng, S. Zhang, F. Ye, V. Jeyakumar, M. Ju, J. Liu, N. McKeown, and A. Vahdat. Libra: Divide and Conquer to Verify Forwarding Tables in Huge Networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 87–99, Berkeley, CA, USA, 2014. USENIX Association.

[12] A. Guha, M. Reitblatt, and N. Foster. Machine-verified Network Controllers. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 483–494, New York, NY, USA, 2013. ACM.

[13] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A compiler and run-time system for network programming languages. *SIGPLAN Not.*, 47(1):217–230, Jan. 2012.

[14] T. Nelson, A. D. Ferguson, M. J. G. Scheer, and S. Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*, NSDI'14, pages 519–531, Berkeley, CA, USA, 2014. USENIX Association.

[15] A. Khurshid, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying Network-wide Invariants in Real Time. *SIGCOMM Comput. Commun. Rev.*, 42(4):467–472, Sept. 2012.

[16] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with anteater. *ACM SIGCOMM Computer Communication Review*, 41(4):290–301, 2011.

[17] R. Beckett, X. K. Zou, S. Zhang, S. Malik, J. Rexford, and D. Walker. An Assertion Language for Debugging SDN Applications. In *Proceedings of the Third Workshop on Hot Topics in Software Defined Networking*, HotSDN '14, pages 91–96, New York, NY, USA, 2014. ACM.

[18] M. Canini, D. Venzano, P. Peresini, D. Kostic, J. Rexford, and others. A NICE Way to Test OpenFlow Applications. In *NSDI*, volume 12, pages 127–140, 2012.

[19] R. Majumdar, S. D. Tetali, and Z. Wang. Kuai: A Model Checker for Software-defined Networks. In *Proceedings of the 14th Conference on Formal Methods in Computer-Aided Design*, FMCAD '14, pages 27:163–27:170, Austin, TX, 2014. FMCAD Inc.

[20] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, pages 282–293, New York, NY, USA, 2014. ACM.