Decentralizing SDN Policies

Oded Padon
Tel Aviv University
odedp@mail.tau.ac.il

Neil Immerman
University of Massachusetts, Amherst
immerman@cs.umass.edu

Aleksandr Karbyshev
Tel Aviv University
karbyshev@post.tau.ac.il

Ori Lahav
Tel Aviv University
orilahav@post.tau.ac.il

Mooly Sagiv
Tel Aviv University
msagiv@post.tau.ac.il

Sharon Shoham
The Academic College of Tel Aviv Yaffo
sharon.shoham@gmail.com

Abstract

Software-defined networking (SDN) is a new paradigm for operating and managing computer networks. SDN enables logically-centralized control over network devices through a "controller" — software that operates independently of the network hardware. Network operators can run both in-house and third-party SDN programs on top of the controller, e.g., to specify routing and access control policies.

In practice, having the controller handle events limits the network scalability. Therefore, the feasibility of SDN depends on the ability to efficiently decentralize network event-handling by installing forwarding rules on the switches. However, installing a rule too early or too late may lead to incorrect behavior, e.g., (1) packets may be forwarded to the wrong destination or incorrectly dropped; (2) packets handled by the switch may hide vital information from the controller, leading to incorrect forwarding behavior. The second issue is subtle and sometimes missed even by experienced programmers.

The contributions of this paper are two fold. First, we formalize the correctness and optimality requirements for decentralizing network policies. Second, we identify a useful class of network policies which permits automatic synthesis of a controller which performs optimal forwarding rule installation.

Categories and Subject Descriptors D.3.2 [Programming Languages]: Language Classifications—Specialized application languages; I.2.2 [Artificial Intelligence]: Automatic Programming—Program synthesis

Keywords Software-defined networking; Network programming languages; Synthesis; Logic programming; Distributed systems

1. Introduction

Software-defined networking (SDN) is a paradigm for operating and managing computer networks [1]. SDN enables logically-centralized control over network devices through a "controller" — software that operates independently of the network hardware. In practice, having the controller handle all network events limits network scalability. Therefore, the feasibility of SDN depends on the ability to efficiently and correctly decentralize handing of network events by installing forwarding rules on the switches. This problem can be difficult and error prone even for experienced programmers.

In particular, there is a subtle local-versus-global issue that is a frequent cause of errors in SDN implementations. Namely, when a packet arrives at a switch, the switch's forwarding rules determine whether the packet is handled locally or sent to the controller. But when should a packet be sent to the controller?

We say that an event e is *relevant* if there is *some sequence of future events* that can only be handled correctly if the controller knows about e. If e is relevant then the switch *must* inform the controller about it. On the other hand, if e is irrelevant, then from the viewpoint of network traffic and efficiency, it is best for the switch to handle e locally and *not* inform the controller about it. Thus, a switch needs to test whether or not an incoming event is relevant. However, relevance is in general a second-order property — does there exist a sequence of future events — that may be difficult or impossible for the switch to test.

In this paper, we solve this problem. We show how to automatically go from the specification of a network policy to its decentralized implementation as forwarding rules to be initially installed on the switches and updated in response to events occurring on the switches.

We make the following contributions:

1. We define the correctness and optimality of a decentralized implementation of a network policy. Our definitions are given in English and more formally in second-order logic.
2. We specify a first-order language, $\mathcal{L}_1$, for expressing network policies. $\mathcal{L}_1$ is reasonably expressive. Furthermore, we show how to automatically synthesize an optimal decentralized implementation of any network policy expressible in $\mathcal{L}_1$.
3. We identify a property that we call k-*bounded causality* and show that any network policy F expressed in $\mathcal{L}_1$ satisfies k-bounded causality with the parameter k easily determined from F. Intuitively, if a network policy has k-bounded causality, then instead of considering all possible sequences of future events, we only need to consider sequences of length at most k.
4. Using the k-*bounded causality* of F, we show that the above second-order definitions of correctness and optimality for a decentralized implementation of F can be written simply in a fragment of first-order logic with equality.

POPL '15, January 15–17, 2015, Mumbai, India.

http://dx.doi.org/10.1145/2676726.2676990

5. Using quantifier elimination on the resulting formulas, we show how to automatically derive quantifier-free implementation rules for F. The latter can be automatically translated to forwarding rules which when installed on the switches, result in a correct and optimal decentralized implementation of the input network policy F.

What this paper does not do: There are many fascinating issues about the **distributed** control of a network, i.e., how to send forwarding rules over the network to keep all the switches in a correct state as cheaply as possible. We will consider some of these questions in future work.

In this paper, we consider the problem of computing the local forwarding rules, not how we send them over the network to the switches. A related interesting question which we also do not consider in this paper is how to compute the incremental changes of the forwarding rules. That is, which rules should we add and remove in reaction to each new relevant event.

This paper is organized as follows: Section 2 provides an informal overview of the problem addressed by this paper. The terminology used in this paper is described in Section 3. The problem addressed by this paper is defined formally in Section 4. Section 5 provides necessary and sufficient conditions for decentralization and optimality. Our first order policy language is defined in Section 6. The algorithm for synthesizing forwarding rules is described in Section 7. Related work is described in Section 8.

2. Overview

This section provides an informal overview of the decentralization problem and how we solve it. The reader need not be familiar with Software-Defined Networks (SDNs).

2.1 Forwarding policies

We assume that we have a finite set of switches, each of which has a fixed set of ports, `Ports`, connected to hosts and other switches. The events of interest are "packet-receive" events, each of which represents a packet arriving at some switch s on some input port. The possible resulting action is for s to forward the packet along some set of ports $P \subseteq$ `Ports`. A policy is a specification of what action to take given an event and the history of all previous events. The policy dictates the behavior of every switch as a function of the global history of events.

As a running example, consider the well-studied policy of a *learning switch* with host migration. The learning switch learns locations of hosts in the network by analyzing the packets incoming on its ports. Initially, it is supposed to flood any arriving packet (i.e., send it to all ports except where it came from), while learning the location of its sender. Future packets, whose destination's location is already learned, should be forwarded directly to the learned port(s) instead of flooding it to all ports. This eliminates unnecessary traffic. It is possible for hosts to change their location, i.e., to *migrate*. The learning switch that supports host migration should forward the packet to the port where the packet's destination was last seen. More precisely, for a packet with destination d arriving at switch s on port p, if the history does not include any previous packet that arrived at s with source d, then s should flood the packet, i.e., send it over all ports except p. On the other hand, if there was such a packet and the most recent of these arrived on port p', then s should send the packet only on port p'.

2.2 Distributed forwarding

SDNs implement forwarding policies using switches that have limited expressive power, along with a centralized controller. Events can either be handled locally by the switch receiving the event, or communicated to the controller.

Algorithm 1 Controller ⟨Policy⟩

```
State := initial_state
while new packet-receive event e do
    action := compute_action(Policy, State, e)
    update_state(State, e)
    rules := infer_rules (Policy, State)
    configure_switches(rules)
    send_to_switches(action)
end while
```

The switches follow forwarding rules that determine the port(s) to which the arriving packets are forwarded according to the attributes of the packet. These rules also determine when the switch must forward the packet to the controller.[1] Each set of rules defines a "stateless" behaviour in the sense that it does *not* refer to the previous history of events. The rules are installed and removed from the switches only by the controller.

In contrast to the switches, the centralized controller is *stateful*: its behavior, which includes instructing the switches how to forward packets that were sent to it, as well as installing forwarding rules on the switches for future packets, may depend on the entire log of events that it has seen so far. For that, the controller may maintain the history of events it has received as part of its state. Algorithm 1 depicts a simple scheme of a stateful controller, parametrized by the policy.

Note that the policy of the learning switch described above cannot be implemented by a switch alone as the handling of the current packet requires access to the history of previous events.

In this work, we focus on the rule inference component of the controller. For scalability and performance, as many events as possible should be handled locally by the switches. However, as shown next it is subtle to correctly install forwarding rules. In fact even current high-level network programming languages including [6, 14, 18] essentially require that the programmer specifies conditions that determine when forwarding rules can be installed (see Section 8).

2.3 Correctness of forwarding rule installation

This paper formulates the notion of correctness of forwarding rule installation. Our formulation is inspired by the OPENFLOW standard [2], but it can be adapted to other distributed systems.

Roughly speaking, we say that a forwarding rule installation policy is *correct* with respect to a policy if all packets are forwarded to the right ports as defined by the policy. Since the policy defines the actions to be performed based on the full sequence of events, while rule installation depends only on the packets that arrive to the controller, correctness is in fact divided to the following two conditions: (1) every packet is forwarded to the right ports (either by the installed rules or by the controller), and (2) packets are sent to the controller whenever necessary.

To formalize (2) we define a notion of relevance. We call an event *relevant* if it may affect the future behavior of the system. Thus the second condition for correctness requires that all relevant events must be forwarded to the controller.

Learning switch forwarding rules Table 1 contains several rule installation policies for the learning switch example. Each of them specifies a set of switch forwarding rules for any history H. In the rules, e denotes the current event (arriving packet). Notice that H does not appear in the rules themselves, as the switches cannot access the history.

[1] For simplicity, we assume that the switches send a packet to the controller iff the installed forwarding rules do not determine any local action for the packet. Our techniques can also deal with rules that explicitly send some events to the controller.

	Rule Installation Policy
trivial	none
$\mathbf{LS}_1$	for any $e_1 \in H$: $\texttt{forward}(e_1.in) \quad \leftarrow \quad e.sw = e_1.sw,\ e.dst = e_1.src$
$\mathbf{LS}_2$	for any $e_1, e_2 \in H$: $\texttt{forward}(e_2.in) \quad \leftarrow \quad e.sw = e_1.sw = e_2.sw,\ e.src = e_1.src,\ e.dst = e_2.src$
$\mathbf{LS}_3$	for any $e_1 \in H$ such that $\forall e' \in H.\,(e'.sw = e_1.sw \wedge e' > e_1 \implies e'.src \neq e_1.src)$ and $e_2 \in H$ such that $\forall e' \in H.\,(e'.sw = e_2.sw \wedge e' > e_2 \implies e'.src \neq e_2.src)$: $\texttt{forward}(e_2.in) \quad \leftarrow \quad e.sw = e_1.sw = e_2.sw,\ e.src = e_1.src,\ e.dst = e_2.src,\ e.in = e_1.in$

Table 1. Sample rule installation policies for the learning switch in terms of current events and histories, e, H respectively. $e_1 > e_2$ means that e_1 occurred later than e_2 in H.

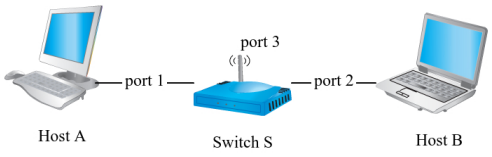

Figure 1. A sample topology for a learning switch s with three ports 1, 2, and 3.

The trivial rule installation policy does not install any rules and thus is correct, but it is far from optimal, as all events are handled by the centralized controller. The $\mathbf{LS}_1$ policy installs a rule to forward an event e to port p if the history includes an event that arrived to the same switch as e on the port p and its source is the destination of e. While this policy seems natural, it is *incorrect* since it installs rules "too early". Figure 1 shows a sample topology (with a single switch s and two hosts A and B) and Table 2 presents a scenario where following $\mathbf{LS}_1$ results in incorrect behaviour. The first event is a packet from A to B that occurred in the switch s and arrived on port 1. Since the switch initially has no rules, this packet is sent to the controller which floods it, and also installs a forwarding rule (according to $\mathbf{LS}_1$) to forward all packets arriving in switch s whose destination is A to port 1. This appears to be a correct forwarding. As a result of this rule, the next event of a packet from B to A is (correctly) forwarded to port 1 by the switch, but it is not sent to the centralized controller. Hence, the controller does not learn that B is connected to port 2. Consequently, the third event (another packet from A to B) is flooded, while it should have been forwarded only to output port 2.

The $\mathbf{LS}_2$ installation policy avoids the bug of premature rule installations, as it installs rules matching packets for which both the source and the destination locations have already been learned. $\mathbf{LS}_2$ is implemented in most SDN systems. It installs a rule to forward e to port p if the history includes: (1) an event that arrived on switch s in input port p and its source is the destination of e; and (2) another event that arrived on s with the same source as e. This is a more modest installation policy which installs rules "per-flow". However, it cannot cope with host migration. Table 3 demonstrates an incorrect behaviour that happens when host A relocates from port 1 to port 3. Indeed, after the second event, when both the locations of hosts A and B were learned, the switch will locally handle packets between A and B. The third event, a packet from A to B that arrived on port 3 after A migrated, is handled locally and is not sent to the controller. Accordingly, the controller is not aware of A's migration. This, again, results in an incorrect behaviour, as the fourth event from B to A is forwarded to port 1, while A informed its new location in port 3.

$\mathbf{LS}_3$ is a correct rule installation policy with correct handling of host migration. It installs a rule to forward e to port p: if (1) the last event in the history at switch s whose source is the destination of e arrived in input port p; (2) the last event in the history at switch s whose source is the source of e arrived in the same input port as e. Thus, it only installs rules for handling packets that come from the last known location of some host. Table 4 demonstrates how $\mathbf{LS}_3$ properly handles the scenario presented in Table 3. After the second event, the rules installed match only events coming on the *known* locations of A and B. Consequently, the third event (after A migrates), is handled by the controller, and thus the controller learns that A migrated. As a result, the controller replaces the previously installed forwarding rules with two new rules, and the forth event is handled correctly by the switch.

The example of the learning switch with migration shows that deriving a correct rule installation policy for a given forwarding policy is challenging even for simple and well-studied examples.

2.4 Optimality of forwarding rule installation

With correctness being the first criterion, our goal is to also ensure that the forwarding rules installed on the switches allow the switches to handle as many events as possible locally. Such a forwarding rule installation policy is called *optimal*. The trivial installation is suboptimal, whereas $\mathbf{LS}_3$ is optimal.

We show that having the installed rules handle *all* irrelevant events is both necessary and sufficient condition for optimality.

2.5 Our approach

Our approach for inferring optimal rule installation consists of a "static" part, which takes place at compile time, and a "dynamic" part which operates at runtime. The static part derives from the given policy a description of the optimal rule installation policy which determines for each history the rules that should be installed (for the learning switch example, this corresponds to obtaining $\mathbf{LS}_3$, see Table 1). The runtime component is executed by the controller whenever an event is forwarded to it, and it is responsible for computing the updated rules to be installed on the switches by applying the rule installation policy on the current history of events. We consider the history of events received by the controller as its state.

2.5.1 Inferring rule installation policies

In general inferring correct forwarding rule installation is undecidable (see Section 6.4). In this paper we define a restricted variant of first-order logic, called $\mathcal{L}_1$, which is powerful enough to capture many useful forwarding policies, and is amenable to automatic synthesis of optimal rule installation policies. We then describe an algorithm for synthesizing an optimal rule installation policy for any given forwarding policy expressed in our logic. For example, for the learning switch, it infers $\mathbf{LS}_3$.

Applying our procedure to the Firewall example from [8] actually improves its implementation by proactively installing more rules, as demonstrated in Section 7.4.

event	action	rule-updates
$\langle s, 1, A \to B \rangle$	flood	**install:** forward(1) $\leftarrow$ $e.sw = s,\ e.dst = A$
$\langle s, 2, A \to B \rangle$	forward(1)	
$\langle s, 1, A \to B \rangle$	flood	

Table 2. A scenario which reveals a bug in the forwarding rule installation policy $\mathbf{LS}_1$. Controller actions are underlined.

event	action	rule-updates
$\langle s, 1, A \to B \rangle$	flood	
$\langle s, 2, B \to A \rangle$	forward(1)	**install:** forward(2) $\leftarrow$ $e.sw = s,\ e.src = A,\ e.dst = B$ forward(1) $\leftarrow$ $e.sw = s,\ e.src = B,\ e.dst = A$
A migrates from 1 to 3		
$\langle s, 3, A \to B \rangle$	forward(2)	
$\langle s, 2, B \to A \rangle$	forward(1)	

Table 3. A scenario which reveals a bug in the the forwarding rule installation policy $\mathbf{LS}_2$. Controller actions are underlined.

event	action	rule-updates
$\langle s, 1, A \to B \rangle$	flood	
$\langle s, 2, B \to A \rangle$	forward(1)	**install:** forward(2) $\leftarrow$ $e.sw = s,\ e.src = A,\ e.dst = B,\ e.in = 1$ forward(1) $\leftarrow$ $e.sw = s,\ e.src = B,\ e.dst = A,\ e.in = 2$
A migrates from 1 to 3		
$\langle s, 3, A \to B \rangle$	forward(2)	**remove the previous two rules** **install:** forward(2) $\leftarrow$ $e.sw = s,\ e.src = A,\ e.dst = B,\ e.in = 3$ forward(3) $\leftarrow$ $e.sw = s,\ e.src = B,\ e.dst = A,\ e.in = 2$
$\langle s, 2, B \to A \rangle$	forward(3)	

Table 4. A scenario demonstrating the correct forwarding rule installation policy $\mathbf{LS}_3$. Controller actions are underlined.

Formulation of correctness and optimality in first-order logic The correctness and optimality conditions of installation policies refer to irrelevant events, where irrelevance of events is examined with respect to unbounded future sequences of events, which naturally requires second-order quantifiers. Nevertheless, we identify a "bounded causality" property of policies and show that for policies admitting this property the condition for an event being irrelevant can be equivalently specified in first-order logic. The bounded causality property states that there is a (computable) bound k, such that if an event affects the behaviour for some sequence of future events, then it will also affect the behaviour for some sequence of k future events. This allows us to construct first-order definitions of an optimal (correct) installation policy.

Specification of policies We specify policies by formulas in $\mathcal{L}_1$, which are interpreted with respect to a history and a "current" event. $\mathcal{L}_1$ allows any Boolean combination of components, where components are existentially quantified formulas, in which all (existential) quantifiers are *guarded*: they either refer to the existence of an event in the history (denoted by $\exists \bar{y} \in \mathbb{H}$, where $\mathbb{H}$ is a special symbol and $\bar{y}$ is an event variable), or they refer to the last event in the history that has some property ψ (denoted by $\exists \bar{y} = Last_\psi$, where $\bar{y}$ is an event variable and ψ is a formula).

For example, the policy for the learning switch that supports migrations of hosts is specified in $\mathcal{L}_1$ by the collection of formulas φ^p, one for each port p, that define when to forward to p:

$$\varphi^p_{\text{ls}} = \exists \bar{y} = Last_{(\bar{x}.sw \approx \bar{y}.sw) \wedge (\bar{x}.dst \approx \bar{y}.src)}.\ (\bar{y}.in \approx p)$$

In addition, the formula that defines when to perform flood is given by:

$$\varphi^{\texttt{flood}}_{\text{ls}} = \bigwedge_{p \in \texttt{Ports}} \neg \varphi^p$$

If instead of φ^p above one takes

$$\varphi^p_{\text{ls}} = \exists \bar{y} \in \mathbb{H}.\ (\bar{x}.sw \approx \bar{y}.sw) \wedge (\bar{x}.dst \approx \bar{y}.src) \wedge (\bar{y}.in \approx p),$$

this results in a policy that forwards each packet to all ports in which the destination of the packet was seen before.

Our design of $\mathcal{L}_1$ was guided by common SDN applications, as well as by the ability to obtain the bounded causality property: $\mathcal{L}_1$-policies have this property. $\mathcal{L}_1$ allows us to express restricted temporal properties, using the "last" quantifiers (or similar variants, see Section 6.2, Remark 4). A possible extension is to also explicitly refer to order on events. We did not augment this into $\mathcal{L}_1$ since it complicates the presentation of our approach.

On the other hand, $\mathcal{L}_1$ formulas cannot express policies that involve counting events in the history or require arithmetic operations such as advanced load balancing (we can, however, handle simple load balancing based on fixed IP distributions). Another source of limitation is the restriction to Boolean combinations of alternation-free quantified formulas. If we allow alternation of quantifiers, the bounded causality property breaks (see Section 7.1).

Static computation of rule installation policy For a policy expressed in $\mathcal{L}_1$ which enjoys the bounded causality property, we are able to specify the optimal rule installation policy in first-order logic (extended with guarded quantifiers). For example, for a learning switch with migration, for each port p, we obtain the following formula:

$$\varphi^p = \varphi^p_{\text{ls}} \wedge \exists \bar{y} = Last_{(\bar{x}.sw \approx \bar{y}.sw) \wedge (\bar{x}.src \approx \bar{y}.src)}.\ (\bar{y}.in \approx \bar{x}.in)$$

Note that when the rule installation policy refers to the history of events via guarded quantifiers, it refers to the filtered history that only consists of events that were sent to the controller, as opposed to the forwarding policy that refers to the full history.

2.5.2 Runtime rule installation

After generating the first-order formulas that express the optimal installation policy, it remains to show that given a concrete history of events that arrived to the controller, the controller can dynamically compute the rules that need to be installed on the switches based on the installation policy (where the installed rules do not refer to the history of events).

For this purpose, we first remove the guarded quantifiers (that still exist in the definition of the installation policy) by using the actual history at hand. Next we remove the unguarded quantifiers using a quantifier elimination process. The result is a set of quantifier-free formulas describing the forwarding rules to be installed.

3. Terminology

SDN is a particular form of a distributed event-driven system, composed of a collection of *decentralized components* and a single *centralized controller*. The decentralized components are configurable switches, and the centralized controller is the OPENFLOW network controller. Events are packet-receive events occurring in the switches.

Next, we precisely formulate the notions involved in defining the decentralization problem.

Events. $\mathcal{Q}$ is a finite set of event attributes. Each attribute $q \in \mathcal{Q}$ is associated with a (possibly, infinite) set of values $\mathcal{E}_q$. Events are records with one field for each event attribute. We will denote by $\mathcal{E}$ the set of all events.

In the context of SDN, events are packet-receive events (packets that arrive at a switch and may or may not be forwarded to the centralized controller). The event attributes $\mathcal{Q}$ include sw, in, src, dst, where sw is the switch receiving the packet; in is the port through which the packet is received ($\mathcal{E}_{in} = \texttt{Ports}$ is a finite set of ports); src is the source field in the packet header; and dst is the destination field in the packet header. If needed, other fields may be included in $\mathcal{Q}$ (e.g. time stamp, logical port, VLAN information etc.).

Histories. A *history* is a finite sequence of events from $\mathcal{E}$. In particular, an event e is considered a history of length one. $H_1 \cdot H_2$ denotes the concatenation of sequences H_1 and H_2. We write $H_1 \preceq H_2$ to mean that H_1 is a subsequence of H_2. $\mathcal{H}$ is the set of all histories. The *size* $|H|$ of a history is the number of elements it contains, $|e_1 \cdot \ldots \cdot e_n| = n$.

Actions. $\mathcal{A}$ is a finite nonempty set of *actions*. In SDN-related examples in this paper, the set of actions is $\mathcal{A} = \texttt{Ports} \cup \{\texttt{drop}, \texttt{flood}\}$, where an action $p \in \texttt{Ports}$ represents forwarding a given packet to port p, the action `flood` represents "flooding" the packet, i.e. forwarding it to all ports except for the port in which it was received, and the action `drop` means dropping the packet without forwarding it.

Policies. A *policy* is a function from $\mathcal{H}$ to $\mathcal{E} \to P^+(\mathcal{A})$, the set of non-empty subsets of $\mathcal{A}$. A policy defines the behaviour of the system, where its input is the sequence of events (history) that occurred so far. Given a history H and a "current" event e_1, $F(H)(e_1)$ determines the (nonempty) set of actions that should be performed in reaction to e_1. When the next event e_2 occurs, the considered history also includes e_1, meaning that the reaction to e_2 will be each action in $F(H \cdot e_1)(e_2)$.

Here are some examples of policies for SDN:

Hub (flooding switch) Flood all arriving packets (i.e., on all switches forward each packet to all ports except for its input port).

Firewall Forward to port 2 all packets arriving on port 1; forward to port 1 a packet arriving on port 2 from source s if the history includes an incoming event on port 1 whose destination is s. In this scenario, we assume that ports 1 and 2 are the only ports of any switch in the network, where trusted hosts are all connected through port 1, and untrusted hosts are connected through port 2.

Learning switch (w/o migration) For a packet arriving in switch s with destination d, forward it to port p if the history includes a packet that arrived in s on the input port p and its source was d (the port p must be unique since there is no migration of hosts). If no such packet exsits in the history, flood the arriving packet.

Learning switch with migration For a packet arriving in switch s with destination d, forward it to port p if the history includes a packet that arrived in s from source d and the most recent such packet has input port p. If no such packet exists in the history, flood the arriving packet.

Authorization Server There is a designated host h_a serving as an *authorization server*, which sends special *authorization* and *deauthorization* packets to the controller. Communication is allowed only between *authorized* hosts. A host becomes authorized when the controller receives an authorization packet authorizing the host, and may become deauthorized if the controller receives a deauthorization packet for that host.

It is also possible to compose several of the aforementioned policies by applying different policies in each switch.

Note that any computable policy, F, may be implemented via a *centralized controller* that keeps track of the entire history: whenever an event e occurs corresponding to switch s receiving packet p, s informs the controller about e. The controller then instructs s to perform the actions $F(H)(e)$ and adds e to its log, $H' := H \cdot e$.

However, sending all events to a centralized controller creates a bottleneck and makes the system not scalable. A scalable system requires that many events will be handled locally by stateless decentralized components, implementing the policy in a decentralized and scalable way.

Event Handlers. To formalize the concept of locally handling events by the decentralized components, we introduce *event handlers*. An event handler is a *partial* function from $\mathcal{E}$ to $P^+(\mathcal{A})$, which returns a set of actions to perform for the events for which it is defined. At any point in time, the behaviour of the decentralized components is defined by an event handler R. The domain of R is the set of events that are locally processed by the decentralized components without informing the controller. Whenever an event e occurs in some decentralized component, if $R(e)$ is defined then each action in $R(e)$ is performed locally. If $R(e)$ is not defined, then the decentralized component informs the controller and awaits further instructions. Below $\mathcal{R}$ denotes the set of all event handlers.

In the context of SDN, event handlers correspond to sets of forwarding rules, e.g., "drop all packets arriving into port 1, and flood all packets arriving into port 2 with source field `192.168.1.123`". The empty event handler corresponds to the empty set of forwarding rules. Note that the controller also implements a (total) event handler at each point in time.

Remark 1. We view the set of switches as one abstract unit that implements a single, joint, event handler. In practice, the joint event handler is divided between the individual switches by taking into account the sw attribute of the events.

Event handler generators. An *event handler generator* is a function from $\mathcal{H}$ to $\mathcal{R}$ (it assigns event handlers to histories).

In the context of SDN, event handler generators correspond to forwarding rule installation policies. Whenever the controller processes a packet-receive event (sent to it by some switch), it also computes a new event handler and installs it on the switches. Note

that if there are different types of switches, then a given switch only needs the forwarding rules pertaining to its type. In practice, only relevant changes to the set of forwarding rules must be sent to each switch.

Let I be an event handler generator. The *I-history*, $\mathcal{C}_I(H)$, for a history H is defined inductively: $\mathcal{C}_I(\epsilon) = \epsilon$, and

$$\mathcal{C}_I(H \cdot e) = \begin{cases} \mathcal{C}_I(H) \cdot e & \text{if } I(\mathcal{C}_I(H))(e) \text{ is not defined} \\ \mathcal{C}_I(H) & \text{otherwise} \end{cases}$$

For a sequence of events $H = e_1 \cdots e_n$, $\mathcal{C}_I(H)$ is the subsequence consisting of those events that are forwarded to the controller, i.e., not handled locally. Thus, $\mathcal{C}_I(H)$ is the log that is saved in the controller. Since $\mathcal{C}_I$ strips out exactly those events that are handled locally and thus not forwarded to the controller, we have,

Lemma 1. *For any event handler generator I, $\mathcal{C}_I$ is idempotent, i.e., for every history H, $\mathcal{C}_I(\mathcal{C}_I(H)) = \mathcal{C}_I(H)$.*

Proof. By induction on H. □

Definition 1. Given an event handler generator I, we say that a history H is *I-reachable* if $H = \mathcal{C}_I(H')$ for some history H'.

Namely, H is I-reachable if there is a sequence of events for which H is the corresponding log. By Lemma 1 we have that H is I-reachable iff $H = \mathcal{C}_I(H)$.

Combined policies. Let F be a policy and I an event handler generator. To see how well F and I play together, we define the *combined policy* of F with respect to I, $F_I : \mathcal{H} \to \mathcal{E} \to P^+(\mathcal{A})$ as follows:

$$F_I(H)(e) = \begin{cases} I(\mathcal{C}_I(H))(e) & \text{if } I(\mathcal{C}_I(H))(e) \text{ is defined} \\ F(\mathcal{C}_I(H))(e) & \text{otherwise.} \end{cases}$$

Thus F_I is the policy that decentralizes according to I and defers to F when I is not defined.

4. The Decentralization Problem

In this section we precisely define the decentralization problem.

Definition 2 (Decentralization)**.** An event handler generator I *decentralizes* a policy F if $F_I = F$.

Definition 3 (Optimality)**.** An event handler generator I that decentralizes a policy F is *optimal* if for every history H and every event handler generator I' that decentralizes F, $\mathcal{C}_I(H) \preceq \mathcal{C}_{I'}(H)$.

Let I be a decentralizer of F. I is optimal if it sends to the controller exactly those events that every decentralizer of F sends to the controller, i.e., it only sends those events that it must send.

In the decentralization problem we are given a policy F and the goal is to produce an optimal event handler generator I that decentralizes F. In addition to the challenge of decentralization, another challenge stems from the fact that the event handlers generated by I should be implemented in some restricted language that is amenable for installation on the decentralized components. More formally, suppose we have some restricted programming language T such that programs of this language can run on the decentralized components. This language is used to implement event handlers, i.e. partial functions from $\mathcal{E}$ to $2^{\mathcal{A}}$.

Remark 2. In this paper we assume that the rules supported by the decentralized components are Boolean conditions on event attributes. Each rule defines a Boolean condition (involving event attributes and perhaps some constants) and an action that should be performed if the condition is satisfied by the current packet-receive event. In the context of SDN, the decentralized components are the switches, which support forwarding rules according to the OPENFLOW standard [2]. The translation of Boolean conditions of this kind to OPENFLOW rules was addressed in [13] in the context of the NetCore network programming language.

An event handler generator I is called *T-suitable* if there is an algorithm that given a history H, produces a T-description of $I(H)$.

Definition 4. The *Decentralization Problem* of a policy F relative to T is given by:

Input. A specification of a computable policy F.
Output. A T-suitable event handler generator I that decentralizes F and is optimal.

Note that without the optimality requirement, this problem has a trivial solution: let $I_\emptyset$ be the event handler generator that always returns the empty event handler. For this generator, we obviously have $F_{I_\emptyset} = F$.

In general, the problem of decentralization is undecidable (for example, if F is defined in some Turing-complete formalism, or using full first-order logic, as shown in Section 6.4). In the sequel, we will identify a useful family of policies for which the decentralization problem can be solved automatically.

5. Characterization of Decentralization and Optimality

In this section, we provide necessary and sufficient conditions for decentralization and optimality. The results presented in this section were verified by means of the proof assistant COQ [17][2].

5.1 Decentralization

We show that an event handler generator I decentralizes F if and only if I agrees with F and does not hide any events relevant to the behaviour of the system.

Definition 5 (Compatibility)**.** An event handler generator I is *compatible* with a policy F if for every event e and I-reachable history H, $I(H)(e) = F(H)(e)$ whenever $I(H)(e)$ is defined.

Intuitively, I is compatible with F if the event handlers generated by I always perform exactly the actions defined by the policy F.

Definition 6 (Transparency)**.** An event handler generator I is *transparent* with respect to a policy F if $F(\mathcal{C}_I(H)) = F(H)$ holds for every history H.

Recall that $\mathcal{C}_I(H)$ is the subsequence of H that is sent to the controller when using I. Thus, events in $\mathcal{C}_I(H)$ are those events that the centralized controller "sees", and the only ones that can affect its behaviour. In general, this filtering of events can compromise the correctness of the whole system. An event handler generator I is transparent with respect to a policy F exactly if this filter does not break the policy F.

The following theorem gives a useful characterization of when an event handler generator decentralizes a policy (Def. 2).

Theorem 1. *An event handler generator I decentralizes a policy F iff it is both compatible and transparent with respect to F.*

We now obtain an equivalent definition for transparency, using the notion of an *irrelevant event*:

Definition 7. An event e is *irrelevant* with respect to a policy F and a history H if $F(H \cdot e \cdot H') = F(H \cdot H')$ for every history H'.

[2] The COQ scripts are available at `https://github.com/karbyshev/decentralize`

Theorem 2. *I is transparent with respect to F iff for every I-reachable history H, $I(H)$ is only defined for events that are irrelevant with respect to F and H.*

From the inductive proof of the previous theorem, we get the following as a corollary.

Corollary 1. *I is transparent with respect to F iff for all histories H and H', $F(H \cdot H') = F(C_I(H) \cdot H')$.*

5.2 Optimality

We call an event handler generator *proactive* for a policy F if it is defined on all possible events:

Definition 8 (Proactiveness)**.** An event handler generator I is *proactive* with respect to a policy F if for every I-reachable history H, $I(H)$ is defined for all events that are irrelevant with respect to F and H.

Theorem 3. *An event handler generator I that decentralizes a policy F is optimal iff it is proactive with respect to F.*

5.3 Optimal decentralizing event handler generator

Combining Theorems 1 and 3, we obtain that an event handler generator I optimally decentralizes F iff it is compatible, transparent, and proactive with respect to F. For any policy F, we define the event handler generator I_F^{opt} that decentralizes F and is optimal with respect to F as follows:

Definition 9. Given a policy F, the event handler generator I_F^{opt} is defined by:

$$I_F^{\text{opt}}(H)(e) = \begin{cases} F(H)(e) & e \text{ is irrelevant w.r.t. } H, F \\ \text{undefined} & \text{otherwise} \end{cases}$$

Obviously, I_F^{opt} is compatible with F and proactive with respect to F. Its transparency follows directly from Theorem 2. Therefore, I_F^{opt} optimally decentralizes F. However, note that even when F is computable, I_F^{opt} may not be computable, since irrelevance is defined in terms of all possible histories H'. In the following section we show that for a useful family of policies, the above definition can be used to automatically produce an optimal event handler generator.

6. First-Order Specifications of Policies

In this section we introduce a formalism for describing policies via a fragment of first-order logic. This fragment is expressive enough for a variety of useful and interesting policies in the context of SDN. In the next section we provide a solution to the decentralization problem for policies specified in this fragment, i.e., an algorithm that synthesizes an optimal suitable event handler generator for a given policy.

6.1 The language $\mathcal{L}$

Syntax. We use an extension of many-sorted first-order language, that we call $\mathcal{L}$, as the basis for our specification languages:

- $\mathcal{L}$'s sorts are the attributes in $\mathcal{Q}$.
- Equality, denoted by $\approx$, is the only predicate symbol.
- $\mathcal{L}$ includes a constant symbol for any element $d \in \bigcup_{q \in \mathcal{Q}} \mathcal{E}_q$ (we will use the elements themselves as constant symbols).
- $\mathcal{L}$ employs ordinary connectives and quantifiers as in first-order logic.
- To succinctly represent events, $\mathcal{L}$ allows "event variables", "event constants" and "selector functions" as syntactic sugar. For example, assuming that $\mathcal{Q} = \{sw, in, src, dst\}$ and $1 \in \mathcal{E}_{src}$, the expression $\forall \bar{y}.\, (1 \approx \bar{y}.src) \wedge (2 \not\approx \bar{y}.dst)$ is a shorthand for the $\mathcal{L}$-formula $\forall v_1 : sw, v_2 : in, v_3 : src, v_4 : dst.\, (1 \approx v_3) \wedge (2 \not\approx v_4)$ (to have shorter formulas, one can equivalently take $\forall v_1 : src, v_2 : dst.\, (1 \approx v_1) \wedge (2 \not\approx v_2)$ in this case).
- $\mathcal{L}$ has a designated free event variable, $\bar{x}$, which we will use to refer to the current event.
- $\mathcal{L}$ has two additional kinds of quantifiers:
  - History-guarded existential quantifiers of the form $\exists \bar{y} \in \mathbb{H}$, where $\mathbb{H}$ is a special symbol and $\bar{y}$ is an event variable.
  - *Last*-quantifiers of the form $\exists \bar{y} = Last_\psi$ where $\bar{y}$ is an event variable and ψ is a quantifier-free formula whose only free variables are $\bar{y}$ and possibly the designated variable $\bar{x}$.

Semantics. Define the many-sorted logical structure $\mathcal{M} = \mathcal{M}(\mathcal{Q}, \mathcal{E}_q : q \in \mathcal{Q})$ as follows:

- $\mathcal{M}$ has a domain $\mathcal{E}_q$ for each sort $q \in \mathcal{Q}$.
- $\approx$ is interpreted as equality.
- Each constant symbol d is interpreted as the element $d \in \mathcal{E}_q$.

The model $\mathcal{M}$ and a given event e naturally induce a satisfaction relation for quantifier-free $\mathcal{L}$-formulas φ with $\bar{x}$ as their only free variable. Since $\mathcal{M}$ is usually fixed, we write $e \models \varphi$ to mean that $\mathcal{M}$ is a model (in the usual sense) of $\varphi\{e/\bar{x}\}$.

The semantics of $\mathcal{L}$ extends usual first-order semantics. It is defined with respect to the model $\mathcal{M}$, and a given history H and event e. Since $\mathcal{M}$ is usually fixed, we write $H, e \models \varphi$ to mean $\mathcal{M}, H, e \models \varphi$, defined inductively as follows:

- If φ is a quantifier-free formula with $\bar{x}$ as its only free variable, then $H, e \models \varphi$ iff $e \models \varphi$.
- The propositional connectives are interpreted as usual.
- $H, e \models \forall v : q.\, \theta$ if $H, e \models \theta\{d/v\}$ for every $d \in \mathcal{E}_q$.
- $H, e \models \exists v : q.\, \theta$ if $H, e \models \theta\{d/v\}$ for some $d \in \mathcal{E}_q$.
- $H, e \models \exists \bar{y} \in \mathbb{H}.\, \theta$ if there is some e' in H for which $H, e \models \theta\{e'/\bar{y}\}$.
- $H, e \models \exists \bar{y} = Last_\psi.\, \theta$ if $H, e \models \theta\{e'/\bar{y}\}$ where e' is the last event in H for which $e \models \psi\{e'/\bar{y}\}$. If no such event e' exists then $H, e \not\models \exists \bar{y} = Last_\psi.\, \theta$.

$\mathcal{L}$-specification of event handler generators. Recall that the set of event handlers, $\mathcal{R}$, is the set of partial functions from $\mathcal{E}$ to $P^+(\mathcal{A})$ and an event handler generator is a function from $\mathcal{H}$ to $\mathcal{R}$. Let $I : \mathcal{H} \to \mathcal{R}$ be an event handler generator. Let $\Phi = \{\varphi^a \mid a \in \mathcal{A}\}$ be a set of $\mathcal{L}$-formulas, where each φ^a includes the variable $\bar{x}$ as its only free variable. We say that Φ *defines* I, if for every history H and event e:

- $I(H)(e) = \{a \in \mathcal{A} \mid H, e \models \varphi^a\} \neq \emptyset$ whenever $I(H)(e)$ is defined.
- $\{a \in \mathcal{A} \mid H, e \models \varphi^a\} = \emptyset$ whenever $I(H)(e)$ is not defined.

6.2 Specifications of policies

Policies are functions from $\mathcal{H}$ to $\mathcal{E} \to P^+(\mathcal{A})$. As with event handler generators, we specify a policy via a set of formulas $\Phi = \{\varphi^a \mid a \in \mathcal{A}\}$. Since policies define total event handlers, it must be the case that for all histories H and events e, $\{a \in \mathcal{A} \mid H, e \models \varphi^a\} \neq \emptyset$.

We will define policies in a restriction $\mathcal{L}_1$ of $\mathcal{L}$ which we now define. First define a *component* to be a prenex formula of $\mathcal{L}$ that has only history-guarded and last-quantifiers:

$$\exists \bar{y}_1 = Last_{\psi_1} \ldots \exists \bar{y}_m = Last_{\psi_m} \exists \bar{z}_1 \in \mathbb{H} \ldots \exists \bar{z}_r \in \mathbb{H}.\, \theta$$

where θ is quantifier-free. The language $\mathcal{L}_1$ consists of all Boolean combinations of components.

Notation 1. For an $\mathcal{L}_1$-formula φ, we denote by $k_{Last}(\varphi)$ the number of occurrences of last-quantifiers in φ, and by $k_{\exists}(\varphi)$ the number of occurrences of guarded existential quantifiers in φ. We also define $k_\varphi = k_{Last}(\varphi) + k_{\exists}(\varphi)$.

An $\mathcal{L}_1$-*policy* is a policy defined by a set of $\mathcal{L}_1$-formulas. The following property of $\mathcal{L}_1$-formulas will be crucial for our main results. Among other things, it will allow us to automatically check that a set $\Phi = \{\varphi^a \mid a \in \mathcal{A}\} \subseteq \mathcal{L}_1$ indeed defines a policy, F_Φ, i.e., that for all histories H, $F_\Phi(H)$ is total. We first introduce the following definition.

Definition 10. Let φ be an $\mathcal{L}_1$-formula, e an event, and $H = e_1 \cdot \ldots \cdot e_n$ a history. The *kernel* of H w.r.t. φ and e, denoted $kernel_{\varphi,e}(H)$, is defined to be a minimal set of indices $\subseteq \{1, \ldots, n\}$ such that for every component α of φ of the form

$$\exists \bar{y}_1 = Last_{\psi_1} \ldots \exists \bar{y}_m = Last_{\psi_m} \exists \bar{z}_1 \in \mathbb{H} \ldots \exists \bar{z}_r \in \mathbb{H}.\, \theta :$$

- for every $1 \leq j \leq m$, $kernel_{\varphi,e}(H)$ contains the maximal index i_j^{Last} such that $e \models \psi_j\{e_{i_j^{Last}}/\bar{y}_j\}$, if such an index exists ($e_{i_j^{Last}}$ is the *last* event in H that satisfies the above), and
- if $H, e \models \alpha$ then $kernel_{\varphi,e}(H)$ includes some indices $i_1^{\exists}, \ldots, i_r^{\exists}$ s.t. $e \models \theta\{e_{i_1^{Last}}/\bar{y}_1\} \ldots \{e_{i_m^{Last}}/\bar{y}_m\}\{e_{i_1^{\exists}}/\bar{z}_1\} \ldots \{e_{i_r^{\exists}}/\bar{z}_r\}$.

Note that the size of $kernel_{\varphi,e}(H)$ is at most k_φ. Intuitively, $kernel_{\varphi,e}(H)$ includes the indices of all events from H that determine whether $H, e \models \varphi$. Therefore:

Theorem 4. *Let φ be an $\mathcal{L}_1$-formula. Then for every history $H = e_1 \cdot \ldots \cdot e_n$ and event e, and for every subsequence $H_1 = e_{j_1} \cdot \ldots \cdot e_{j_\ell}$ of H such that $kernel_{\varphi,e}(H) \subseteq \{j_1, \ldots, j_\ell\}$, we have $H_1, e \models \varphi$ iff $H, e \models \varphi$.*

Proof. Let $H = e_1 \cdot \ldots \cdot e_n$ be some history and e some event. Let $H_1 = e_{j_1} \cdot \ldots \cdot e_{j_\ell}$ be a subsequence of H such that $kernel_{\varphi,e}(H) \subseteq \{j_1, \ldots, j_\ell\}$. We claim that for every component α of φ we have: $H, e \models \alpha$ iff $H_1, e \models \alpha$. Indeed, consider some component $\alpha = \exists \bar{y}_1 = Last_{\psi_1} \ldots \exists \bar{y}_m = Last_{\psi_m}. \exists \bar{z}_1 \in \mathbb{H} \ldots \exists \bar{z}_r \in \mathbb{H}.\, \theta$ of φ. A crucial observation is that for $1 \leq j \leq m$, any event e' that is the last event in H that satisfies $e \models \psi_j\{e'/\bar{y}\}$ is also the last event in H_1 that satisfies this property. Furthermore, there is no event e' in H that satisfies $e \models \psi_j\{e'/\bar{y}\}$ iff there is no such event in H_1. Suppose that $H, e \models \alpha$. Then $H_1, e \models \alpha$ by definition of $kernel_{\varphi,e}(H)$ (and due to the fact that any last event in H satisfying ψ remains a last event in H_1). For the converse, suppose that $H_1, e \models \alpha$. Then since all last elements of H_1 are maintained in H, and since $H_1 \preceq H$ which ensures that the events satisfying the history guarded quantifiers are also maintained, we get that $H, e \models \alpha$. □

Remark 3. In Theorem 4 we use the set $kernel_{\varphi,e}(H)$ of indices to define a subsequence H_0 of $e_1 \cdot \ldots \cdot e_n$ and we require inclusion of $kernel_{\varphi,e}(H)$ in the set of indices defining the subsequence $H_1 = e_{j_1} \cdot \ldots \cdot e_{j_\ell}$ rather than requiring that $H_0 \preceq H_1$. The reason for that is that $e_1 \cdot \ldots \cdot e_n$ may contain duplicates of the same event. For an event that appears more than once we must make sure to keep in H_1 the correct *occurrence* (index) of the event to preserve the last elements. Therefore, it is not enough to require that $H_0 \preceq H_1 \preceq H$. For example, suppose that $\varphi = \exists \bar{y} = Last_\psi.\theta$ and let $H = e_1 e_2 e_1$, where e_1 and e_2 both satisfy ψ but only e_1 satisfies θ. Then in this case $H, e \models \varphi$ and $kernel_{\varphi,e}(H) = \{3\}$, consisting of the index of the last element satisfying ψ. If we just consider the corresponding subsequence $H_0 = e_1$, then for $H_1 = e_1 e_2$ we cannot guarantee $H_1, e \models \varphi$ iff $H, e \models \varphi$, even though $H_0 \preceq H_1$. This is because we kept in H_1 the first occurrence of e_1 where in fact we needed the second occurrence, thus changing the last element. When considering the sets of indices we see that indeed $\{3\} \not\subseteq \{1, 2\}$.

Using Theorem 4 and the property that $|kernel_{\varphi,e}(H)| \leq k_\varphi$, we can check if $\Phi = \{\varphi^a \mid a \in \mathcal{A}\} \subseteq \mathcal{L}_1$ defines a policy. We need to check that for all histories and events, at least one of the $\mathcal{L}_1$-formulas in $\{\varphi^a \mid a \in \mathcal{A}\}$ holds. This amounts to checking that the $\mathcal{L}_1$-formula $\bigvee_{a \in \mathcal{A}} \varphi^a$ is valid. Theorem 4 ensures that for the validity (or satisfiability) of an $\mathcal{L}_1$-formula φ it suffices to consider only histories that include up to k_φ events. Furthermore, if we want to check that some set S of actions should never take place together (e.g. $\{\texttt{flood}, \texttt{drop}\}$), we can verify that the policy admits these restrictions by checking that the formula $\bigwedge_{a \in S} \varphi^a$ is unsatisfiable.

Remark 4. The last quantifiers allow us to select some (occurrences of) events from a history and refer to their properties. One can extend $\mathcal{L}_1$ to include additional selectors, such as "last k", "first k" etc., as long as they select a fixed number of (occurrences of) events from the history, and in addition have the property that applying the selection on a subsequence of the history that includes all the selected indices will result in the same selection. These properties ensure that Theorem 4 still holds, when Definition 10 (kernel) is adapted accordingly.

6.2.1 $\mathcal{L}_1$-specifications of SDN policies.

Now, we give $\mathcal{L}_1$ specifications for the example policies from Section 3. Recall that in the context of SDN, $\mathcal{Q} = \{sw, in, src, dst\}$, and the set of actions is $\mathcal{A} = \texttt{Ports} \cup \{\texttt{flood}, \texttt{drop}\}$, where $\texttt{Ports} = \mathcal{E}_{in}$ is a finite set of ports present in each switch. In the following, we use p to denote a port in $\texttt{Ports}$, and the formulas for the actions that are not specified are assumed to be $\bot$.

Hub (flooding switch). The policy for the hub is specified by:

$$\varphi_{\text{hub}}^{\texttt{flood}} = \top$$

Firewall The policy for the firewall is specified by:

$$\varphi_{\text{fw}}^1 = (\bar{x}.in \approx 2) \wedge \exists \bar{y} \in \mathbb{H}.(\bar{y}.in \approx 1) \wedge (\bar{y}.dst \approx \bar{x}.src)$$
$$\varphi_{\text{fw}}^2 = (\bar{x}.in \approx 1)$$
$$\varphi_{\text{fw}}^{\texttt{drop}} = \neg\varphi_{\text{fw}}^1 \wedge \neg\varphi_{\text{fw}}^2$$

Learning switch The policy for the learning switch is specified by:

$$\varphi_{\text{ls}}^p = \exists \bar{y} \in \mathbb{H}.\, (\bar{x}.sw \approx \bar{y}.sw) \wedge (\bar{x}.dst \approx \bar{y}.src) \wedge (\bar{y}.in \approx p)$$
$$\varphi_{\text{ls}}^{\texttt{flood}} = \bigwedge_{p \in \texttt{Ports}} \neg\varphi_{\text{ls}}^p$$

Learning switch with migrations The policy for the learning switch that supports migrations of hosts is specified by:

$$\varphi_{\text{lsm}}^p = \exists \bar{y} = Last_{(\bar{x}.sw \approx \bar{y}.sw) \wedge (\bar{x}.dst \approx \bar{y}.src)}.\, (\bar{y}.in \approx p)$$
$$\varphi_{\text{lsm}}^{\texttt{flood}} = \bigwedge_{p \in \texttt{Ports}} \neg\varphi_{\text{lsm}}^p$$

Authorization server To model the scenario, we introduce an attribute *type* in $\mathcal{Q}$, along with two constants A and D meaning that the sent message is an authorization or deauthorization massage, respectively. We assume that the authorization server connects directly to the switch, so we can model the authorization (deauthorization) of host h by a packet whose source is h_a, its destination is h, and its type is A (D).

The policy for the switch connected to an authorization server is specified by:

$$\varphi_{\text{auth}}^{\texttt{flood}} = \bar{x}.src \not\approx h_a \wedge \bar{x}.dst \not\approx h_a \implies (\exists \bar{y} = Last_{\psi(\bar{x}.src)}.\, \bar{y}.type = A) \wedge (\exists \bar{y} = Last_{\psi(\bar{x}.dst)}.\, \bar{y}.type = A)$$
$$\varphi_{\text{auth}}^{\texttt{drop}} = \neg\varphi_{\text{auth}}^{\texttt{flood}}$$

where $\psi(h) = \bar{y}.src = h_a \wedge \bar{y}.dst = h$.

6.3 Specification of event handlers

We finally complete the description of the specification language by choosing the programming language T for the decentralized components to be the quantifier-free fragment of $\mathcal{L}$. The rationale is that this is convenient for us and even very primitive components, such as switches in the context of SDN, can run this T. More precisely, T is the fragment $\mathcal{L}_0$ of $\mathcal{L}$ consisting of quantifier-free $\mathcal{L}$-formulas whose only free variable is $\bar{x}$. A collection $\{\varphi_R^a \mid a \in \mathcal{A}\}$ of such formulas *defines* an event handler R if the following hold for every event e:

- $R(e) = \{a \in \mathcal{A} \mid e \models \varphi_R^a\} \neq \emptyset$ whenever $R(e)$ is defined.
- $\{a \in \mathcal{A} \mid e \models \varphi_R^a\} = \emptyset$ whenever $R(e)$ is not defined.

For example, the event handler R which drops all packets arriving in port 1, and is undefined for all other events, is specified in $\mathcal{L}_0$ as $\varphi_R^{\texttt{drop}} = (\bar{x}.in \approx 1)$, and $\varphi_R^a = \bot$ for all other actions a.

6.4 Undecidability of decentralization of $\mathcal{L}$-policies

If we allow to define policies using formulas in the unrestricted language $\mathcal{L}$ instead of $\mathcal{L}_1$, the decentralization problem becomes undecidable. To prove this, we show that the finite satisfiability problem for first-order logic with a single binary predicate can be reduced to the problem of (optimally) decentralizing a policy expressed using $\mathcal{L}$-formulas. Let ψ be a formula in (unsorted) first-order logic with one binary predicate B. We transform ψ into an $\mathcal{L}$-formula by simulating every pair of elements satisfying B by an event in the history. Technically, let ψ' be the formula obtained from ψ by replacing any occurrence of the binary predicate $B(u, w)$ by the formula $(\exists \bar{y} \in \mathbb{H}.(\bar{y}.src \approx u) \wedge (\bar{y}.dst \approx w))$. In addition, all the original variables (and quantifiers) in ψ which are unsorted, become sorted with sort "host". Finally, all free variables in ψ are existentially quantified in ψ'. ψ' is an $\mathcal{L}$-formula (note that it has no free variable), and it can be used to define the following policy F: $\varphi^{\texttt{flood}} = \psi', \varphi^{\texttt{drop}} = \neg\psi'$.

We now claim that ψ has no finite model iff $I_F^{\text{opt}}(\epsilon) = R_{\texttt{drop}}$ such that $R_{\texttt{drop}}(e) = \texttt{drop}$ for every event e, where I_F^{opt} is the optimal event handler generator for F.

The construction of ψ' ensures that ψ has no finite model iff there is no finite history that satisfies ψ'. In addition, if there is no finite history that satisfies ψ', then the policy simply instructs to always drop all events. This means that $I_F^{\text{opt}}(\epsilon) = R_{\texttt{drop}}$. For the converse, assume that $I_F^{\text{opt}}(\epsilon) = R_{\texttt{drop}}$. Then by induction, for every history H, $\mathcal{C}_{I_F^{\text{opt}}}(H) = \epsilon$ and hence $F_{I_F^{\text{opt}}}(H) = I_F^{\text{opt}}(\mathcal{C}_{I_F^{\text{opt}}}(H)) = I_F^{\text{opt}}(\epsilon) = R_{\texttt{drop}}$, meaning all events are always dropped, and by the assumption that I_F^{opt} is correct with respect to F, we get that $F(H) = R_{\texttt{drop}}$ for any H, hence no (finite) history satisfies ψ'.

7. Our Solution

In this section, we introduce a method for synthesizing programs that implement optimal event handler generators for given $\mathcal{L}_1$-policies. We start by showing that for any given set of $\mathcal{L}_1$-formulas that defines the policy F, one can automatically derive a set of $\mathcal{L}$-formulas that defines the optimal decentralizing event handler generator I_F^{opt} (see Definition 9).

Then, we will show that any event handler generator I that is defined by a set of $\mathcal{L}$-formulas is $\mathcal{L}_0$-suitable. That is, we present an algorithm (see Section 7.3) that given I and a history H, produces an $\mathcal{L}_0$-description of $I(H)$.

Using these ingredients we obtain an implementation of a controller in the context of SDN, as described in Algorithm 2. The controller is parametrized by the policy F, and by I_F^{opt}, which is expressed in $\mathcal{L}$ and precomputed from F by the method described in Section 7.2. The controller accumulates the sequence H of events it receives as its state. For a new event, it computes the action to be performed by evaluating the $\mathcal{L}_1$-formulas defining the policy on the current history. It then infers rules by applying the algorithm from Section 7.3 to compute $I_F^{\text{opt}}(H)$.

Algorithm 2 Runtime System $\langle F, I_F^{\text{opt}} \rangle$

```
H := ϵ
while new packet-receive event e do
    action := F(H, e)        # compute an action
    H := H · e               # update state
    rules := I_F^opt(H)      # compute rules (Section 7.3)
    configure_switches(rules)
    send_to_switches(action)
end while
```

7.1 Bounded causality

For a general policy F, I_F^{opt} may not be computable. This is due to the fact that the relevance of an event may depend on the behavior of the policy for infinitely many possible future sequences of events (see Definition 7). However, we now show that for $\mathcal{L}_1$-definable policies, we can bound the size of future sequences of events that must be considered.

Definition 11. For an integer k, an event e is *k-relevant* with respect to policy F and history H if there exists a history H' of size at most k that distinguishes whether e occurred, i.e., $F(H \cdot e \cdot H') \neq F(H \cdot H')$.

Definition 12 (k-bounded causality)**.** For an integer k, we say that a policy F has *k-bounded causality* if for any event e and history H, if e is relevant with respect to F and H then it is also k-relevant.

In other words, a policy F has k-bounded causality if whenever an event e is relevant, it can cause a change after at most k additional events.

Theorem 5. *Let $\Phi = \{\varphi^a \mid a \in \mathcal{A}\} \subseteq \mathcal{L}_1$ define a policy F_Φ. Then F_Φ has k_Φ-bounded causality, for*

$$k_\Phi := \max\{k_{Last}(\varphi^a) + 2 \cdot k_\exists(\varphi^a) \mid a \in \mathcal{A}\}. \tag{1}$$

Proof. We show that F has k_Φ-bounded causality. Given H, H' and e such that $F(H \cdot e \cdot H') \neq F(H \cdot H')$, it follows that there exists an event e' and an action $a \in \mathcal{A}$ such that $a \in F(H \cdot e \cdot H')(e')$ but $a \notin F(H \cdot H')(e')$ or vice versa. W.l.o.g. assume the former. This means that $H \cdot e \cdot H', e' \models \varphi^a$ but $H \cdot H', e' \not\models \varphi^a$. We construct a history $H_0 \preceq H'$ of size at most k_Φ such that the same holds when H' is replaced by H_0.

In order to construct H_0, we construct the set of indices of the events in H' that participate in H_0. Let

$$J_1' = \mathit{kernel}_{\varphi^a, e'}(H \cdot e \cdot H') \cap \{|H \cdot e| + 1, \ldots, |H \cdot e \cdot H'|\}$$

be all the indices in $\mathit{kernel}_{\varphi^a, e'}(H \cdot e \cdot H')$ which refer to events taken from H'. To consider the actual indices in H' we subtract $|H \cdot e|$ from all of the indices. The result, $J_1^{H'} = \{i - |H \cdot e| \mid i \in J_1'\}$, is a set of indices from H'. Similarly, let

$$J_2' = \mathit{kernel}_{\varphi^a, e'}(H \cdot H') \cap \{|H| + 1, \ldots, |H \cdot H'|\}$$

be all the indices in $\mathit{kernel}_{\varphi^a, e'}(H \cdot H')$ of events taken from H', and let $J_2^{H'} = \{i - |H| \mid i \in J_2'\}$ be the corresponding set of indices in H'.

We define a set of indices $J_0 = J_1^{H'} \cup J_2^{H'}$ of events in H' and define a subsequence H_0 of H' accordingly. Namely, suppose that $H' = e_1 \cdot \ldots \cdot e_{|H'|}$. Then $H_0 = e_{i_1} \cdot \ldots \cdot e_{i_k}$ where $1 \leq e_{i_1} < \ldots < e_{i_k} \leq |H'|$ and $J_0 = \{e_{i_1}, \ldots, e_{i_k}\}$.

Recall that the kernel size of a history w.r.t. φ^a is at most $k_{\varphi^a} = k_{Last}(\varphi^a) + k_\exists(\varphi^a)$. Since both $J_1^{H'}$ and $J_2^{H'}$ include all the indices of the "last" events from H' (see Definition 10), which are the same, then the size of J_0 and therefore also the size of H_0 is bounded by $k_{Last}(\varphi^a) + 2 \cdot k_\exists(\varphi^a) \leq k_\Phi$. Further, since $H \cdot e \cdot H_0 \preceq H \cdot e \cdot H'$ such that the set of indices of $H \cdot e \cdot H_0$ in $H \cdot e \cdot H'$ is $(\{1, \ldots, |H \cdot e|\} \cup J_1') \supseteq \mathit{kernel}_{\varphi^a, e'}(H \cdot e \cdot H')$, then by Theorem 4, $H \cdot e \cdot H', e' \models \varphi^a$ iff $H \cdot e \cdot H_0, e' \models \varphi^a$. Similarly, since $H \cdot H_0 \preceq H \cdot H'$ such that the set of indices of $H \cdot H_0$ in $H \cdot H'$ is $(\{1, \ldots, |H|\} \cup J_2') \supseteq \mathit{kernel}_{\varphi^a, e'}(H \cdot H')$, then by Theorem 4, $H \cdot H', e' \models \varphi^a$ iff $H \cdot H_0, e' \models \varphi^a$. We conclude that $H \cdot e \cdot H_0, e' \models \varphi^a$ but $H \cdot H_0, e' \not\models \varphi^a$ as required. □

Remark 5. By inspecting Definition 10 and the proof of Theorem 4 and Theorem 5, it follows that in several cases we can obtain a slightly better bound (lower k_Φ), that will simplify our examples below. In particular:

- If the given set of formulas $\Phi = \{\varphi^a \mid a \in \mathcal{A}\}$ defining F contains *only* history guarded quantifiers, then we can take
$$k_\Phi := \max\{k_\exists(\varphi^a) - 1 \mid a \in \mathcal{A}\}. \quad (2)$$
To see this, note that without last quantifiers, it cannot be the case that $H \cdot H', e' \models \alpha$ but $H \cdot e \cdot H', e' \not\models \alpha$ for some component α. Furthermore, if we have $H \cdot H', e' \models \theta\{e_1/\bar{z}_1\} \ldots \{e_r/\bar{z}_r\}$ for some $e_1, \ldots, e_r$, then $H \cdot e \cdot H', e' \models \theta\{e_1/\bar{z}_1\} \ldots \{e_r/\bar{z}_r\}$ for the same events (none of these properties is guaranteed in the presence of last quantifiers). This allows us to choose $k_\exists(\varphi^a)$ elements (indices) to comprise H_0, instead of $2 \cdot k_\exists(\varphi^a)$: for components that are satisfied in both $H \cdot H'$ and $H \cdot e \cdot H'$ it suffices to choose the elements based on $H \cdot H'$ (rather than both). Finally, we need one event less since there is at least one component α for which $H \cdot e \cdot H', e' \models \alpha$ but $H \cdot H', e' \not\models \alpha$. For this component we can choose $k_\exists(\alpha) - 1$ events, knowing that the additional event must be e itself.
- If the formulas φ^a contain only last quantifiers then we can take
$$k_\Phi := \max\{k_{Last}(\varphi^a) - 1 \mid a \in \mathcal{A}\}. \quad (3)$$
By the proof of Theorem 5 it suffices to consider $k_{Last}(\varphi^a)$ "last" events from H' in the construction of H_0. In fact, since there is at least one component α for which $H \cdot e \cdot H', e' \models \alpha$ but $H \cdot H', e' \not\models \alpha$ or vice versa, while the last events in H' are clearly the same, then the difference must stem from a last event in H or $H \cdot e$. Either way, this event is not in H' making the size of H_0 at most $k_{Last}(\varphi^a) - 1$.

Unbounded causality in the presence of quantifier alternation If we allow alternation of quantifiers in the specification of policies, then the k-bounded causality property does not necessarily hold. As an example, consider a policy F which defines two possible actions, a and b, where $\varphi^a = \forall \bar{z}_1 \in \mathbb{H}. \exists \bar{z}_2 \in \mathbb{H}. \bar{z}_1.src = \bar{z}_2.dst \wedge \bar{z}_1.dst = \bar{z}_2.src$ and $\varphi^b = \neg\varphi^a$. Namely, if in the current history H every event has a matching event with opposite source and destination, then action a is performed on the current event e (for any e). Otherwise b is performed. Assume to the contrary that F has the k-bounded causality property for some k. Consider a history H of length $k+1$ which consists of events such that all sources are different, all destinations are different, and all sources are different from all destinations. Further, consider an event e whose source and destination also differ from all sources and destinations in H. Clearly, e is relevant. However, e is not k-relevant. Indeed, for any future sequence of events H'' with $\leq k$ events, at least one of the events in H remains unmatched, whether H'' follows H or $H \cdot e$. Thus, $F(H \cdot H'')(e') = F(H \cdot e \cdot H'')(e') = b$ for any e'.

7.2 Construction of an optimal event handler generator

In this section we show that for any policy F defined by a set of $\mathcal{L}_1$ formulas, we can define the optimally decentralizing event handler generator I_F^{opt} using a set of $\mathcal{L}$-formulas. First, we introduce the following notation.

Notation 2. Let φ be an $\mathcal{L}$-formula with one free event variable $\bar{x}$. Let $\bar{y}_1, \ldots, \bar{y}_n$ be event variables. We denote by $\varphi[\bar{y}_1, \ldots, \bar{y}_n]$ the formula obtained from φ as follows:

- Each subformula of φ of the form $\exists \bar{y} \in \mathbb{H}. \psi$ is replaced by
$$(\exists \bar{y} \in \mathbb{H}. \psi) \vee \bigvee_{1 \leq i \leq n} \psi\{\bar{y}_i/\bar{y}\}.$$
- Each subformula of φ of the form $\exists \bar{y} = \mathit{Last}_\psi. \theta$ is replaced by
$$\big(\exists \bar{y} = \mathit{Last}_\psi. \theta \wedge N_\psi^0[\bar{y}_1, \ldots, \bar{y}_n]\big) \vee \bigvee_{1 \leq i \leq n} \Big(\theta\{\bar{y}_i/\bar{y}\} \wedge \psi\{\bar{y}_i/\bar{y}\} \wedge N_\psi^i[\bar{y}_1, \ldots, \bar{y}_n]\Big)$$
where $N_\psi^i[\bar{y}_1, \ldots, \bar{y}_n] = \bigwedge_{i<j\leq n} \neg\psi\{\bar{y}_j/\bar{y}\}$

Intuitively, the notation $\varphi[\bar{y}_1, \ldots, \bar{y}_n]$ is used to encode the formula φ when the history is extended by a suffix of events described by $\bar{y}_1, \ldots, \bar{y}_n$. This affects both the history-guarded quantifiers and the last quantifiers. Formally, this is stated by the following lemma.

Lemma 2. *For any $\mathcal{L}$-formula φ with one free variable $\bar{x}$, history H, events $e_1, \ldots, e_n, e$, and distinct event variables $\bar{y}_1, \ldots, \bar{y}_n$, we have $H \cdot e_1 \cdot \ldots \cdot e_n, e \models \varphi$ iff $H, e \models \varphi[\bar{y}_1, \ldots, \bar{y}_n]\{e_1/\bar{y}_1\} \ldots \{e_n/\bar{y}_n\}$.*

Using this notation, the synthesis of $\{\varphi^a_{I_F^{\text{opt}}} \mid a \in \mathcal{A}\}$ from $\{\varphi^a \mid a \in \mathcal{A}\}$ is done using the following definition:

Definition 13. Given $\{\varphi^a \mid a \in \mathcal{A}\}$, for any action $a \in \mathcal{A}$ the formula $\varphi^a_{I_F^{\text{opt}}}$ is defined by: $\varphi^a_{I_F^{\text{opt}}} := \varphi^a \wedge \bigwedge_{\substack{0 \leq m \leq k_\Phi \\ a' \in \mathcal{A}}} \psi_m^{a'}$ where:

- k_Φ is the bound defined in Theorem 5, such that F has the k_Φ-bounded causality property.
- $\psi_m^{a'}$ denotes the formula:
$$\forall \bar{y}_1, \ldots, \bar{y}_m, \bar{y}.\ \varphi^{a'}\{\bar{y}/\bar{x}\}[\bar{x}, \bar{y}_1, \ldots, \bar{y}_m] \iff \varphi^{a'}\{\bar{y}/\bar{x}\}[\bar{y}_1, \ldots, \bar{y}_m]$$

We present several clarifications:

- The formulas φ^a, $\psi_m^{a'}$, and therefore also $\varphi^a_{I_F^{\text{opt}}}$, all have $\bar{x}$ as their only free event variable, which represents the "current" event.
- Each φ^a_I is an $\mathcal{L}$-formula, but not an $\mathcal{L}_1$-formula since it includes unguarded quantifiers.
- The conjunction of the formulas $\psi_m^{a'}$ over all actions $a' \in \mathcal{A}$ encodes the property that the (current) event e, represented by $\bar{x}$, is irrelevant when considering exactly m events into the future (where to obtain k_Φ-irrelevance we consider the conjunction over all $0 \leq m \leq k_\Phi$). Technically, $\bar{y}_1, \ldots, \bar{y}_m$ represent the m future events $e_1, \ldots, e_m$, and $\bar{y}$ represents an event to which F is applied in the future with or without considering e. On the left hand side of the comparison we therefore apply $\varphi^{a'}$ on $\bar{y}$ and use $[\bar{x}, \bar{y}_1, \ldots, \bar{y}_m]$ to extend the history by the events $e, e_1, \ldots, e_m$. Similarly, on the right hand side we use $[\bar{y}_1, \ldots, \bar{y}_m]$ to extend the history by only $e_1, \ldots, e_m$ (excluding e).

The above explanations can be straightforwardly summarized in the following theorem that establishes the correctness of our

construction. In particular, note that since F is defined by $\mathcal{L}_1$-formulas, it has k_Φ-bounded causality, and so k_Φ-irrelevance is equivalent to full irrelevance.

Theorem 6. *Let F be a policy defined by a set $\Phi = \{\varphi^a \mid a \in \mathcal{A}\}$ of $\mathcal{L}_1$-formulas. Then the set of $\mathcal{L}$ formulas $\Phi^{opt} = \{\varphi^a_{I^{opt}_F} \mid a \in \mathcal{A}\}$ as in Definition 13 defines the event handler generator I^{opt}_F.*

Proof. Follows from Theorem 2, Theorem 5 and Lemma 2. □

This completes the construction of an $\mathcal{L}$-definition of an optimal event handler generator I that decentralizes F.

Remark 6. In Definition 13, in the construction of $\varphi^a_{I^{\text{opt}}_F}$, it is sufficient to include only $m = 0$ and $m = k_\Phi$ in the conjunction $\bigwedge_{0 \le m \le k_\Phi, a' \in \mathcal{A}} \psi^{a'}_m$, omitting the intermediate values. The reason is that $\mathcal{L}_1$-formulas cannot detect stuttering in non-empty histories. Namely, for any $\varphi \in \mathcal{L}_1$, we have

$$H \cdot e_1 \cdot H', e \models \varphi \iff H \cdot e_1 \cdot e_1 \cdot H', e \models \varphi.$$

As a result, if $\psi^{a'}_m$, which considers m events into the future, does not hold for some $0 < m < k_\Phi$, it also does not hold for $m = k_\Phi$ by repeating one of the m future "refuting" elements.

7.3 $\mathcal{L}_0$-suitability of $\mathcal{L}$-definitions

Here we show that the event handler generator I defined before as a set of $\mathcal{L}$-formulas is $\mathcal{L}_0$-suitable. Assume we are given such a set $\{\varphi^a_I \mid a \in \mathcal{A}\}$. We describe a general algorithm that will take as input a concrete history H, and produce a set $\{\varphi^a_{I(H)} \mid a \in \mathcal{A}\}$ of $\mathcal{L}_0$-formulas that defines the event handler $I(H)$.

Let H be a particular history. For $\{\varphi^a_{I(H)} \mid a \in \mathcal{A}\}$ to define $I(H)$ the following has to hold:

Correctness For any event e and action a:

$$H, e \models \varphi^a_I \quad \text{iff} \quad e \models \varphi^a_{I(H)}$$

The computation is done in two steps, as described next.

Step 1. First, we eliminate the history guarded quantifiers and the last quantifiers according to their semantics. Suppose $H = e_1 \cdot \ldots \cdot e_n$. Then:

- Every subformula in each φ^a_I that begins with a history guarded quantifier $\exists \bar{y} \in \mathbb{H}.\theta$ is replaced by a disjunction: $\bigvee_{1 \le i \le n} \theta\{e_i/\bar{y}\}$
- Every subformula in each φ^a_I that begins with a last quantifier $\exists \bar{y} = Last_\psi.\theta$ is replaced with the following disjunction:

$$\bigvee_{1 \le i \le n} \Big(\theta\{e_i/\bar{y}\} \wedge \psi\{e_i/\bar{y}\} \wedge N^i_\psi[e_1, \ldots, e_n]\Big)$$

where $N^i_\psi[e_1, \ldots, e_n] = \bigwedge_{i<j\le n} \neg\psi\{e_j/\bar{y}\}$

Note that this transformation is similar to the one defined in Notation 2, where we assume that the full history is $e_1 \cdot \ldots \cdot e_n$. The next proposition justifies correctness.

Proposition 1. *Let φ be an $\mathcal{L}$-formula with one free event variable $\bar{x}$, and suppose that φ' was obtained from φ as described above. Then $H, e \models \varphi$ iff $e \models \varphi'$.*

Step 2. In this stage we have a set of formulas in which equality is the only predicate symbol. To make them $\mathcal{L}_0$-formulas, it remains to eliminate the quantifiers. For this simple case, it can be done by "equality enumeration" as explained next.

Definition 14. For a quantifier-free $\mathcal{L}$-formula φ in which equality is the only predicate symbol and a variable v, $\varphi^{\not\approx v}$ is the $\mathcal{L}$-formula, inductively denied by:

$$\varphi^{\not\approx v} = \begin{cases} \top & \varphi = (v \approx v) \\ \bot & \varphi = (v \approx t) \text{ or } \varphi = (t \approx v) \text{ where } t \neq v \\ \varphi & \varphi = (t_1 \approx t_2) \text{ where } t_1 \neq v \text{ and } t_2 \neq v \\ \varphi_1^{\not\approx v} \circ \varphi_2^{\not\approx v} & \varphi = \varphi_1 \circ \varphi_2 \text{ where } \circ \in \{\wedge, \vee, \to, \leftrightarrow\} \\ \neg\varphi_1^{\not\approx v} & \varphi = \neg\varphi_1 \end{cases}$$

Intuitively, this operator resolves any equality involving v in φ under the assumption that the value of v is different from the values of all other variables and constants that appear in φ. Note that the variable v does not occur anymore in $\varphi^{\not\approx v}$, and the following is valid in first-order logic: $\Big(\bigwedge_{t \in \mathcal{V}_\varphi} v \not\approx t\Big) \implies \big(\varphi \iff \varphi^{\not\approx v}\big)$ where $\mathcal{V}_\varphi$ is the set of free variables and constant symbols that occur in φ, except for the variable v itself.

Equipped with this operator, we can now explain what we mean by equality enumeration.

The elimination of quantifiers from the formulas that we obtained after step 1 directly follows from the following two equivalences, that hold for any quantifier-free formula φ that has only equality as a predicate symbol:

1. If $\mathcal{E}_q$ is infinite then $\forall v : q.\varphi \equiv \Big(\bigwedge_{t \in \mathcal{V}_\varphi} \varphi\{t/v\}\Big) \wedge \varphi^{\not\approx v}$, where $\mathcal{V}_\varphi$ is the set of free variables and constants in φ except for v.
2. If $\mathcal{E}_q$ is finite then $\forall v : q.\varphi \equiv \Big(\bigwedge_{t \in \mathcal{E}_q} \varphi\{t/v\}\Big)$.

The second equivalence is obvious (recall that we assume a constant symbol for any element in the domain), and the first can be explained as follows:

$$\begin{aligned} \forall v : q.\varphi &\equiv \Big(\bigwedge_{t \in \mathcal{V}_\varphi} \varphi\{t/v\}\Big) \wedge \forall v : q.\Big(\Big(\bigwedge_{t \in \mathcal{V}_\varphi} v \not\approx t\Big) \to \varphi\Big) \\ &\equiv \Big(\bigwedge_{t \in \mathcal{V}_\varphi} \varphi\{t/v\}\Big) \wedge \forall v : q.\Big(\Big(\bigwedge_{t \in \mathcal{V}_\varphi} v \not\approx t\Big) \to \varphi^{\not\approx v}\Big) \\ &\equiv \Big(\bigwedge_{t \in \mathcal{V}_\varphi} \varphi\{t/v\}\Big) \wedge \Big((\exists v : q. \bigwedge_{t \in \mathcal{V}_\varphi} v \not\approx t) \to \varphi^{\not\approx v}\Big) \\ &\equiv \Big(\bigwedge_{t \in \mathcal{V}_\varphi} \varphi\{t/v\}\Big) \wedge \varphi^{\not\approx v} \end{aligned}$$

where the last equivalence is due to the fact that $\mathcal{E}_q$ is infinite.

By repeatedly applying these equivalences from the inner-most to the outer-most quantifier, one obtains an equivalent quantifier-free formula. Hence, the second step amounts to performing this simple quantifier elimination on all formulas in the set, producing $\{\varphi^a_{I(H)} \mid a \in \mathcal{A}\}$ as a result.

To summarize, if we are given Φ, an $\mathcal{L}_1$ specification of a policy F, we showed in Theorem 6 that we can automatically construct Φ^{opt}, an $\mathcal{L}$ specification of an optimal event handler generator I^{opt}_F. Now we have given an algorithm to go from Φ^{opt} and a history H to the quantifier-free specification of forwarding rules that implement $I^{\text{opt}}_F(H)$. In conclusion, we have shown the following:

Theorem 7. *Given an $\mathcal{L}_1$-specification of a policy F, we can automatically produce a program which takes a history H and produces the quantifier-free specification of forwarding rules that implement an optimal implementation of F.*

Remark 7. The running time of our solution to the decentralization problem is determined by the complexity of the static part and the complexity of the dynamic part. For a policy formula ψ expressed in $\mathcal{L}_1$ with causality bound k, the static translation produces an

$\mathcal{L}$-formula for I^{opt}_{ψ} of the length $\mathcal{O}(|\psi| \cdot k^2)$ with $k+1$ nested universal quantifiers, where $|\psi|$ is the length of formula ψ. Denote the resulting formula by ψ'. In the dynamic part, for the current history H of length $h = |H|$, Step 1 eliminates guarded quantifiers from ψ' and produces a formula ψ'' of size $\mathcal{O}(|\psi'| \cdot h^{2k})$ with the same nesting depth of universal quantifiers as ψ'. Step 2 eliminates (unguarded) universal quantifiers from ψ'' and returns a formula of size $\mathcal{O}(|\psi''| \cdot m^{k+1})$ where m is a maximum of k, h, the number of constants occurring in ψ and of all $|\mathcal{E}_q|$, for finite $\mathcal{E}_q$. In total, the running time of the algorithm of Theorem 7 is in $\mathcal{O}(|\psi| \cdot k^2 \cdot h^{2k} \cdot m^{k+1})$. In all the examples in this paper, the algorithm is linear in $|\psi|$, as k can be taken to be 0 according to Remark 5.

It is important to note that (1) the complexity is measured with respect to $|H|$ where H is the log kept by the controller, and not the full history. This is especially important as the optimal event handler generator minimizes the size of this log; and (2) the running time analysis refers to the processing of events handled by the controller, whose number is minimized, and not of events handled by the switches.

7.4 Examples

We demonstrate our technique on two examples: the firewall and the learning switch that supports migrations.

Firewall Recall the policy for the firewall, given by:

$$\varphi^1 = (\bar{x}.in \approx 2) \wedge \exists \bar{z} \in \mathbb{H}.(\bar{z}.in \approx 1) \wedge (\bar{z}.dst \approx \bar{x}.src)$$
$$\varphi^2 = (\bar{x}.in \approx 1)$$
$$\varphi^{\texttt{drop}} = \neg\varphi^1 \wedge \neg\varphi^2$$

In the above we renamed $\bar{y}$ to $\bar{z}$ in order to avoid confusion. For the simplicity of the presentation, we focus here on just one switch. To obtain the $\mathcal{L}$-description of the optimal decentralizing event handler generator, we use Definition 13. For each $a \in \{1, 2, \texttt{drop}\}$, we get $\varphi^a_I = \varphi^a \wedge \tau$ where

$$\tau = \forall \bar{y}, \bar{y}_1, \ldots, \bar{y}_{k_\Phi}.$$
$$\bigwedge_{a' \in \mathcal{A}} \varphi^{a'}\{\bar{y}/\bar{x}\}[\bar{x}, \bar{y}_1, \ldots, \bar{y}_{k_\Phi}] \iff \varphi^{a'}\{\bar{y}/\bar{x}\}[\bar{y}_1, \ldots, \bar{y}_{k_\Phi}]$$

Since the policy formulas have only one history-guarded quantifier, we can take $k_\Phi = 0$, as explained in Remark 5. Then we can compute an equivalent formula for τ as follows.

$$\tau \equiv \forall \bar{y}. \bigwedge_{a' \in \mathcal{A}} \varphi^{a'}\{\bar{y}/\bar{x}\}[\bar{x}] \iff \varphi^{a'}\{\bar{y}/\bar{x}\}$$
$$\overset{(1)}{\equiv} \forall \bar{y}. (\varphi^1\{\bar{y}/\bar{x}\}[\bar{x}] \iff \varphi^1\{\bar{y}/\bar{x}\}) \wedge$$
$$(\varphi^2\{\bar{y}/\bar{x}\}[\bar{x}] \iff \varphi^2\{\bar{y}/\bar{x}\}) \wedge$$
$$(\varphi^{\texttt{drop}}\{\bar{y}/\bar{x}\}[\bar{x}] \iff \varphi^{\texttt{drop}}\{\bar{y}/\bar{x}\})$$
$$\overset{(2)}{\equiv} \forall \bar{y}. (\varphi^1\{\bar{y}/\bar{x}\}[\bar{x}] \iff \varphi^1\{\bar{y}/\bar{x}\})$$
$$\overset{(3)}{\equiv} \forall \bar{y}. (\bar{y}.in \approx 2) \wedge (\bar{x}.in \approx 1) \wedge (\bar{x}.dst \approx \bar{y}.src) \implies$$
$$\exists \bar{z} \in \mathbb{H}. (\bar{z}.in \approx 1) \wedge (\bar{z}.dst \approx \bar{y}.src)$$

In step (2), we omit the conjunct for $\varphi^{\texttt{drop}}$, since $\varphi^{\texttt{drop}}$ is a Boolean combination of φ^1 and φ^2. Furthermore, the conjunct for φ^2 can be omitted as tautological. Step (3) is by applying standard equivalences of propositional logic. By applying rules of FOL with equality, it is also easy to see that the last formula can be further simplified to:

$$\tau \equiv (\bar{x}.in \approx 1) \implies \exists \bar{z} \in \mathbb{H}.(\bar{z}.in \approx 1) \wedge (\bar{z}.dst \approx \bar{x}.dst)$$

In particular, for the empty history one obtains:

$$\varphi^{\texttt{drop}}_I \equiv \neg\varphi^1 \wedge \neg\varphi^2 \wedge (\bar{x}.in \not\approx 1) \equiv \bar{x}.in \not\approx 1$$

which is equivalent to $(\bar{x}.in \approx 2)$. This means that the synthesized rule installation policy will initially install a rule to drop all packets arriving on port 2. This actually improves the SDN program implementing a stateful firewall policy from [8] by proactively installing more rules.

Learning switch with migrations Recall the policy for the learning switch that supports migrations of hosts, given by:

$$\varphi^p = \exists \bar{z} = Last_{(\bar{x}.sw \approx \bar{z}.sw) \wedge (\bar{x}.dst \approx \bar{z}.src)}. (\bar{z}.in \approx p)$$
$$\varphi^{\texttt{flood}} = \bigwedge_{p \in \texttt{Ports}} \neg\varphi^p$$

Again, we renamed $\bar{y}$ to $\bar{z}$ in order to avoid confusion.

For each $a \in \texttt{Ports} \cup \{\texttt{flood}\}$, we have $\varphi^a_I = \varphi^a \wedge \tau$ where

$$\tau \equiv \forall \bar{y}. \bigwedge_{a' \in \mathcal{A}} \varphi^{a'}\{\bar{y}/\bar{x}\}[\bar{x}] \iff \varphi^{a'}\{\bar{y}/\bar{x}\}$$
$$\overset{(1)}{\equiv} \forall \bar{y}. \bigwedge_{p \in \texttt{Ports}} (\varphi^p\{\bar{y}/\bar{x}\}[\bar{x}] \iff \varphi^p\{\bar{y}/\bar{x}\}) \wedge$$
$$(\varphi^{\texttt{flood}}\{\bar{y}/\bar{x}\}[\bar{x}] \iff \varphi^{\texttt{flood}}\{\bar{y}/\bar{x}\})$$
$$\overset{(2)}{\equiv} \forall \bar{y}. \bigwedge_{p \in \texttt{Ports}} (\varphi^p\{\bar{y}/\bar{x}\}[\bar{x}] \iff \varphi^p\{\bar{y}/\bar{x}\})$$
$$\overset{(3)}{\equiv} \forall \bar{y}. \bigwedge_{p \in \texttt{Ports}} [((\exists \bar{z} = Last_{\bar{y}.sw \approx \bar{z}.sw \wedge \bar{y}.dst \approx \bar{z}.src}. \bar{z}.in \approx p) \wedge$$
$$\neg(\bar{y}.sw \approx \bar{x}.sw \wedge \bar{y}.dst \approx \bar{x}.src)) \vee$$
$$(\bar{x}.in \approx p \wedge \bar{y}.sw \approx \bar{x}.sw \wedge \bar{y}.dst \approx \bar{x}.src) \iff$$
$$\exists \bar{z} = Last_{\bar{y}.sw \approx \bar{z}.sw \wedge \bar{y}.dst \approx \bar{z}.src}. \bar{z}.in \approx p]$$
$$\overset{(4)}{\equiv} \bigwedge_{p \in \texttt{Ports}} \forall \bar{y}. [\bar{y}.sw \approx \bar{x}.sw \wedge \bar{y}.dst \approx \bar{x}.src \implies$$
$$((\exists \bar{z} = Last_{\bar{y}.sw \approx \bar{z}.sw \wedge \bar{y}.dst \approx \bar{z}.src}. \bar{z}.in \approx p) \iff \bar{x}.in \approx p)]$$
$$\overset{(5)}{\equiv} \bigwedge_{p \in \texttt{Ports}} [(\exists \bar{z} = Last_{\bar{x}.sw \approx \bar{z}.sw \wedge \bar{x}.src \approx \bar{z}.src}. \bar{z}.in \approx p) \iff$$
$$\bar{x}.in \approx p]$$
$$\overset{(6)}{\equiv} \exists \bar{z} = Last_{\bar{x}.sw \approx \bar{z}.sw \wedge \bar{x}.src \approx \bar{z}.src}. \bar{z}.in \approx \bar{x}.in$$

In the initial description of τ we used the property that the policy formulas have only last quantifiers. Therefore, we can take $k_\Phi = 0$, as explained in Remark 5. We can omit the conjunct for $\varphi^{\texttt{flood}}$ in step (2), since $\varphi^{\texttt{flood}}$ is a Boolean combination of φ^p, $p \in \texttt{Ports}$. In step (4), we used the propositional equivalence

$$((A \wedge \neg B) \vee (C \wedge B) \iff A) \equiv (B \implies (A \iff C))$$

and the first-order equivalence $\forall x. \bigwedge A_i \equiv \bigwedge(\forall x. A_i)$. In steps (5) and (6), we used rules of FOL with equality.

We demonstrate the computation done at the runtime for the topology from Figure 1 and scenario from Table 4. As the current history we consider

$$H = \langle s, 1, A \to B\rangle \cdot \langle s, 2, B \to A\rangle$$

Consider the current event $e = \langle s, 3, A \to B\rangle$, which means that the host A has migrated to port 3. After eliminating the last-quantifier with respect to the current history and omitting the conjuncts for equalities of switch components, we get for τ:

$$1 \approx \bar{x}.in \wedge \bar{x}.src \approx A \wedge \bar{x}.src \not\approx B \vee 2 \approx \bar{x}.in \wedge \bar{x}.src \approx B$$

We see that for the current event, the above formula is equivalent to $\bot$, which means that the current event e must be forwarded to the controller.

Consider as a current history

$$H = \langle s, 1, A \to B\rangle \cdot \langle s, 2, B \to A\rangle \cdot \langle s, 3, A \to B\rangle$$

after the migration of host A to port 3. Consider the current packet $e = \langle s, 2, B \to A\rangle$. After applying Step 1, we get for τ

$$\begin{aligned} &1 \approx \bar{x}.in \wedge \bar{x}.src \approx A \wedge \bar{x}.src \not\approx B \wedge \bar{x}.src \not\approx A \vee \\ &2 \approx \bar{x}.in \wedge \bar{x}.src \approx B \wedge \bar{x}.src \not\approx A \vee \\ &3 \approx \bar{x}.in \wedge \bar{x}.src \approx A \\ \equiv\ &2 \approx \bar{x}.in \wedge \bar{x}.src \approx B \vee 3 \approx \bar{x}.in \wedge \bar{x}.src \approx A \end{aligned}$$

Since e satisfies the formula $\varphi_I^3 = \varphi^3 \wedge \tau$, the current event is forwarded to port 3, as intended.

8. Related Work

Verification and Synthesis of Correct SDN Programs Verifying the correctness of SDN programs has recently received a lot of attention [3–5, 9, 12, 16]. These works aim to verify a given controller program which implements a forwarding rule installation policy. In contrast, this paper identifies a specific correctness criterion. We also automatically infer correct and optimal forwarding rule installation policies for forwarding policies expressed in a useful fragment of first-order logic.

Atomicity of Rule Installation This paper assumes that events are executed atomically, ignoring out-of-order rule installations. Consistently updating a software-defined network is an important challenge in SDN (see [15]). One can enforce the atomicity of the inferred forwarding rules using barriers. In the future, it may be possible also infer the minimal barriers for a given forwarding policy using techniques similar to [11].

Language Abstractions for SDNs. [6, 10, 14, 18] introduce abstractions for programming controllers in order to simplify the task of programming controllers and to enable more efficient implementations. Our work can be combined with these abstractions in order to further simplify the task or SDN programming and to reduce the number of errors for limited policies expressible in our language.

[18] transforms algorithmic policies into forwarding rules which drastically simplifies SDN programming. However, unlike our work, they do not analyze the dependencies between the forwarding policies and the packet histories. Therefore that cannot guarantee correctness for forwarding policies which depend on the histories.

[6] provides a high level level language called *Pyretic* for programming SDN controllers. Pyretic provides a mechanism to define functions which update the forwarding rules in an abstract way. These functions are called when new packets arrive to the controller, according to filters defined by the programmer. Note that the programmer has to write the update code, and also to explicitly define the filter stating which packets should trigger an update. The programmer is also responsible for assuring that the filter definition is consistent with the update code which requires non-trivial temporal reasoning. In contrast we provide a higher level of abstraction for expressing the forwarding policy and automatically infer correct and optimal installation of forwarding rules.

Figure 2 shows the learning switch (w/o host migration) expressed in Pyretic. In lines 2–7 the programmer defines the update and in line 8 the programmer defines the filter. Maintaining consistency between the filter and the update code is error prone and subtle. For instance, the consistency between lines 2–7 and line 8 depends on the assumption that there is no host migration, and this assumption is not explicit anywhere in the code. Therefore, it may

```
1: def learn(self):
2:     def update(pkt):
3:         self.P =
4:         if_(match(dstmac=pkt['srcmac'],
5:                   switch=pkt['switch']),
6:             fwd(pkt['inport']),
7:             self.P)
8:     q = packets(1,['srcmac','switch'])
9:     q.when(update)
```

Figure 2. A learning switch in Pyretic.

```
1: ON packet_in(p):
2:   DO forward(new) WHERE
3:     learned(p.locSw, new.locPt, p.dlDst);
4:
5: ON packet_in(p):
6:   INSERT (p.locSw, p.locPt, p.dlSrc)
7:     INTO learned WHERE
8:     NOT learned(p.locSw, p.locPt, p.dlSrc);
```

Figure 3. A learning switch in FlowLog.

be worthwhile to integrate our inference mechanism into Pyretic by limiting the expressive power of the update code.

FlowLog [14] is an innovative tierless programming language for SDNs. It allows the programmer to express the forwarding policy in an SQL like language.

Figure 3 shows the learning switch expressed in FlowLog. Lines 1–3 express the forwarding behavior using the relation `learned`. Lines 5–8 express updating of the `learned` relation in response to new packets. Line 8 (the `WHERE`-clause), defines for which packets the update should take place. At first glance, it looks as if the programmer can declare the forwarding policy w/o worrying about rule installation. However, the installation of rules is determined by the `WHERE`-clause, which has to be specified by the programmer. Requiring that the programmer manually specifies the `WHERE`-clause may lead to subtle performance and correctness issues. For example, the programmer can naively interpret these rules according to SQL semantics and replace the condition in the `WHERE`-clause by `true`. In SQL semantics, both variants are equivalent since adding an element to a relation does nothing if the element is already in the relation. However, in FlowLog, changing the `WHERE`-clause to `true` results in sending all packets to the controller since they will all satisfy the `WHERE`-clause, leading to a performance issue. Similarly, if the `WHERE`-clause is inconsistent with the update that takes place (e.g., over-restrictive), then relevant packets will not be sent to the controller, and the correctness of the system may be compromised.

Indeed, FlowLog and Pyretic both split the task of updating the controller state into two interleaved mechanisms, and it is up to the programmer to make sure they are used consistently. In contrast, we propose a way to automatically infer one from the other and guarantee correctness and optimality.

NetCore [13] is a novel framework for programming SDN controllers. It includes a mechanism to transform Boolean combinations of conditions on events into OpenFlow rules, which can be used to translate the installation rules inferred by this paper into OpenFlow. Because NetCore is Turing complete, it requires that the programmers specify the conditions under which rules need to be installed in the form of an auxiliary hand-written function. As argued in this paper, this approach is error prone and requires non-trivial temporal reasoning on all possible sequences of events which may occur. Our approach in this paper limits the expressive power of the policy language to avoid the need to supply unchecked conditions.

Datalog as a Specification Language Datalog is another good candidate for expressing network forwarding policies [7]. This means expressing the forwarding policy as a Datalog query over the history. For policies expressed in Datalog with recursion (with or without allowing stratified negation), k-bounded causality does not necessarily hold. This is also the case for policies expressed in Datalog without recursion and with negation. As for Datalog without recursion and without negation, policies expressed in it can also be expressed using $\mathcal{L}_1$-formulas, and therefore have the k-bounded causality for some k. However, not every policy expressible using $\mathcal{L}_1$ formulas can be expressed in Datalog without recursion and without negation, including simple policies such as the learning switch (since the forwarding behavior is non-monotonic in the history).

9. Conclusion

In this work we have investigated the decentralization problem of SDN. We characterized correctness and optimality of forwarding rule installation policies. While the problem of finding an optimal correct installation policy is undecidable in general, we show how to construct such installation policies for forwarding policies expressed in a natural restricted variant of first order logic. As a future work, it is worthwhile to study the applicability of our approach in a broader setting of distributed systems and distributed databases.

Extensions Our approach can be straightforwardly extended in several ways.

Our specification language of policies, $\mathcal{L}_1$, includes two kinds of guarded quantifiers: one which examines existence of some event in the history, and one which refers to the last event in the history that has some property. As discussed in Remark 4, our results also apply if we extend $\mathcal{L}_1$ by additional guarded quantifiers, such as "last k" or "first k".

We consider a finite set of actions $\mathcal{A}$, which allows us to describe functions from $\mathcal{E}$ to $\mathcal{A}$ using a finite set of formulas, one for each action. In order to handle packet rewriting (performed for example in NAT boxes and supported by OpenFlow) one needs to consider an infinite set of actions since in this context, an action may be "forward to port 2 and change the source field to `1.2.3.4`". This means that the (infinite) set of addresses will be added to the set of actions. Our approach can be adapted to handle this as well by adding another sort, representing an action, to be used in $\mathcal{L}_1$ formulas, such that actions will be specified internally in $\mathcal{L}_1$. A policy will then be defined by a single formula with an additional free variable describing the action to be performed. The conjunction over all actions that is used when defining the formulas of the optimal event handler generator (Definition 13) will simply be replaced by a universal quantifier which will later be eliminated.

Our approach can also handle multiple types of events, by adding an attribute that describes the event type, and any other type specific attributes needed.

It is also possible to develop synthesis methods which are applied incrementally each time a history is changed and to produce only the new rules to be installed or removed.

Acknowledgments

Padon, Immerman, Karbyshev, Lahav and Sagiv were funded by the European Research Council under the European Union's Seventh Framework Program (FP7/2007–2013) / ERC grant agreement no. [321174-VSSC], and by a grant from the Israel Science Foundation (652/11). Immerman was partially supported by NSF grant CCF 1115448. Shoham was partially supported by BSF grant no. 2012259. We thank Nate Foster, Daniel Jackson, Tova Milo, Aurojit Panda, Michael Schapira and the anonymous referees for insightful comments which improved this paper.

References

[1] The Open Networking Foundation. `http://opennetworking.org`.

[2] *OpenFlow Switch Specification*, Oct. 2013. Version 1.4.0.

[3] ANDERSON, C. J., FOSTER, N., GUHA, A., JEANNIN, J.-B., KOZEN, D., SCHLESINGER, C., AND WALKER, D. NetKAT: Semantic foundations for networks. In *POPL* (2014), S. Jagannathan and P. Sewell, Eds., ACM, pp. 113–126.

[4] BALL, T., BJØRNER, N., GEMBER, A., ITZHAKY, S., KARBYSHEV, A., SAGIV, M., SCHAPIRA, M., AND VALADARSKY, A. Vericon: Towards verifying controller programs in software-defined networks. In *PLDI* (June 2014), SIGPLAN, ACM.

[5] CANINI, M., VENZANO, D., PERES, P., KOSTIC, D., AND REXFORD, J. A NICE Way to Test OpenFlow Applications. In *NSDI* (2012).

[6] FOSTER, N., GUHA, A., REITBLATT, M., STORY, A., FREEDMAN, M. J., KATTA, N. P., MONSANTO, C., REICH, J., REXFORD, J., SCHLESINGER, C., WALKER, D., AND HARRISON, R. Languages for software-defined networks. *IEEE Communications Magazine 51*, 2 (2013), 128–134.

[7] HUANG, S. S., GREEN, T. J., AND LOO, B. T. Datalog and emerging applications: an interactive tutorial. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data* (2011), ACM, pp. 1213–1216.

[8] KATTA, N. P., REXFORD, J., AND WALKER, D. Logic programming for software-defined networks. In *ACM SIGPLAN Workshop on Cross-model Language Design and Implementation* (Sept. 2012).

[9] KAZEMIAN, P., VARGHESE, G., AND MCKEOWN, N. Header Space Analysis: Static Checking For Networks. In *NSDI* (2012).

[10] KOPONEN, T., AMIDON, K., BALLAND, P., CASADO, M., CHANDA, A., FULTON, B., GANICHEV, I., GROSS, J., GUDE, N., INGRAM, P., JACKSON, E., LAMBETH, A., LENGLET, R., LI, S.-H., PADMANABHAN, A., PETTIT, J., PFAFF, B., RAMANATHAN, R., SHENKER, S., SHIEH, A., STRIBLING, J., THAKKAR, P., WENDLANDT, D., YIP, A., AND ZHANG, R. Network virtualization in multi-tenant datacenters. In *NSDI* (2014).

[11] KUPERSTEIN, M., VECHEV, M. T., AND YAHAV, E. Automatic inference of memory fences. *SIGACT News 43*, 2 (2012), 108–123.

[12] KUZNIAR, M., PERESINI, P., CANINI, M., VENZANO, D., AND KOSTIC, D. A SOFT Way for OpenFlow Switch Interoperability Testing. In *CoNEXT* (2012), pp. 265–276.

[13] MONSANTO, C., FOSTER, N., HARRISON, R., AND WALKER, D. A compiler and run-time system for network programming languages. *SIGPLAN Not. 47*, 1 (Jan. 2012), 217–230.

[14] NELSON, T., FERGUSON, A. D., SCHEER, M. J. G., AND KRISHNAMURTHI, S. Tierless programming and reasoning for software-defined networks. In *NSDI* (2014), USENIX Association, pp. 519–531.

[15] REITBLATT, M., FOSTER, N., REXFORD, J., SCHLESINGER, C., AND WALKER, D. Abstractions for network update. In *ACM SIGCOMM* (2012), pp. 323–334.

[16] SKOWYRA, R., LAPETS, A., BESTAVROS, A., AND KFOURY, A. A verification platform for sdn-enabled applications. In *HiCoNS* (2013).

[17] THE COQ DEVELOPMENT TEAM. *The Coq proof assistant reference manual*. TypiCal Project (formerly LogiCal), 2012. Version 8.4.

[18] VOELLMY, A., WANG, J., YANG, Y. R., FORD, B., AND HUDAK, P. Maple: simplifying SDN programming using algorithmic policies. In *ACM SIGCOMM* (2013), pp. 87–98.