VeriCon: Towards Verifying Controller Programs in Software-Defined Networks

Thomas Ball and Nikolaj Bjørner
Microsoft Research
tball,nbjorner}@microsoft.com

Aaron Gember
University of Madison-Wisconsin
agember@cs.wisc.edu

Shachar Itzhaky
Tel Aviv University
shachar@cs.tau.ac.il

Aleksandr Karbyshev
Technische Universität München
aleksandr.karbyshev@in.tum.de

Mooly Sagiv
Tel Aviv University
msagiv@acm.org

Michael Schapira and
Asaf Valadarsky
Hebrew University
{schapiram,asaf.valadarsky}@huji.ac.il

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 from the network hardware, and can be viewed as the network operating system. Network operators can run both inhouse and third-party SDN programs (often called applications) on top of the controller, e.g., to specify routing and access control policies. SDN opens up the possibility of applying formal methods to prove the correctness of computer networks. Indeed, recently much effort has been invested in applying finite state model checking to check that SDN programs behave correctly. However, in general, scaling these methods to large networks is challenging and, moreover, they cannot guarantee the absence of errors.

We present VeriCon, the first system for verifying that an SDN program is correct on all admissible topologies and for all possible (infinite) sequences of network events. VeriCon either confirms the correctness of the controller program on all admissible network topologies or outputs a concrete counterexample. VeriCon uses first-order logic to specify admissible network topologies and desired network-wide invariants, and then implements classical Floyd-Hoare-Dijkstra deductive verification using Z3. Our preliminary experience indicates that VeriCon is able to rapidly verify correctness, or identify bugs, for a large repertoire of simple core SDN programs. VeriCon is compositional, in the sense that it verifies the correctness of execution of any single network event w.r.t. the specified invariant, and can thus scale to handle large programs. To relieve the burden of specifying inductive invariants from the programmer, VeriCon includes a separate procedure for inferring invariants, which is shown to be effective on simple controller programs. We view VeriCon as a first step en route to practical mechanisms for verifying network-wide invariants of SDN programs.

1. Introduction

Software defined networking (SDN) is an emerging architecture for operating and managing computer networks, fueled by adoption by major technology companies [11]. SDN allows network administrators to program a (logically) centralized software-based network “controller” that maintains a global view of the network rather than manage tens of thousands of lines of configuration scattered among thousands of network devices (e.g., routers). By centralizing network control and separating it from network hardware, SDN enables network administrators to run (in-house and third-party) programs on top of the controller, e.g., for route computation and to enforce access control policies, without having to wait for features to be embedded in router/switch vendors’ proprietary and closed software environments.

The controller configures the network devices, called “switches”, through a simple API in the form of a flow table: each flow table entry contains a set of packet fields to match, and an action to be taken, such as “send packet out of port X” or “drop packet”. When a switch receives a packet it seeks a matching flow entry in its table; if the table has no matching flow entries, the switch sends the packet to the controller, which can then add a flow entry directing the switch how to handle similar packets in the future. The controller maintains a global view of the network by gathering information from the switches (e.g., traffic flow statistics and switch/link failures) and can change the flow table in response to changes in the network and network traffic. The controller effectively plays the role of the network’s operating system; programs running on top of it (often called “applications”) can use the controller to specify forwarding rules, access control policies, etc., without directly interacting with network hardware or with other SDN programs.
Operating large networks is a complex task that is highly prone to error. This problem is only expected to be exacerbated as network configuration shifts from today’s human time scale to event-driven automated configuration with SDN. Guaranteeing network-wide invariants (e.g., enforcing access control) is hence of great importance. SDN opens up the possibilities of applying formal methods to SDN programs to prove the correctness of computer networks. Indeed, much effort has recently been invested in applying finite state model checking to check that SDN programs behave correctly. However, generally speaking, finite-state model checking of SDN programs suffers from two main problems: (1) scaling these methods to large networks is highly nontrivial; and (2) finite-state model checking might identify errors, but cannot guarantee the absence of errors. We present VeriCon, a tool for provably verifying network-wide invariants of SDN programs at compile time.

VeriCon symbolically reasons about (potentially infinite) network states to verify that network-wide invariants are preserved for any sequence of “events” (e.g., the controller receives a packet header from a switch, or a link fails) and on all admissible topologies, where invariants and topologies are expressed via first-order logic. VeriCon is sound in the sense that if it outputs “no errors” then the preservation of the specified invariants is guaranteed. When verification fails, VeriCon displays a concrete scenario that violates the invariants, in the form of an admissible topology and an event, and it is therefore a useful tool for debugging controller programs. Notice that VeriCon reasons about both the controller and switch correctness.

We designed a simple imperative programming language, called Core SDN (CSDN), for writing SDN programs. CSDN is but a means to an end; we use it to illustrate that Hoare style verification of SDN programs is feasible in many programming languages. VeriCon models the semantics of both controller events, such as the receipt of a packet from a switch at the controller, and switch events, such as executing a rule entry in a switch’s flow table and forwarding an incoming packet to a certain port (or dropping it). Thus, sequences of events in VeriCon capture both controller-to-switch interaction and switch-to-switch interaction, making it possible to reason about SDN programs’ behavior for arbitrary sequences of network events (at both the controller and the switches). In fact, we treat the network as just a guarded command program where guards model controller and switch events. This allows us to naturally handle the non-deterministic aspect of networks where switch events can be triggered if the switch includes a matching rule in the flow table, and otherwise controller events occur. Also, these events interact in a non-trivial way. The handler of controller events can change the content of the flow table and then a switch event can cause a violation of an invariant. A premature installation of a flow table rule can break an invariant if the state of the controller is not correctly updated after subsequent switch event. VeriCon can catch these kinds of errors before the program is executed, provided that the programmer can specify the desired invariants.

States in CSDN consist of relations representing the network topology and the flow tables of individual switches, and also of auxiliary relations that maintain the controller’s information about the global network state. The state — of the controller and switches — is updated by adding or removing tuples from relations (e.g., the switches’ flow tables). CSDN was designed in the spirit of the OpenFlow standard [1].

VeriCon takes as input a CSDN program as well as formulas in first-order logic that specify constraints on the network topology and desired safety properties (network-wide invariants). VeriCon employs the standard weakest preconditions [5] in order to generate verification condition (VC). The VC is a first order formula which holds if and only if: (i) the initial network state satisfies the invariant and (ii) the invariant is inductive, i.e., the execution of arbitrary controller and switch events maintain the invariant. The semantics of the controller events is taken from the CSDN program. The semantics of the switch is dictated by the OpenFlow standard [1]. The soundness of completeness of such approaches is completely standard (e.g., see [7]).

An automated theorem prover (Z3 [4]) checks the verification condition, to either verify the correctness of the controller program or to generate a concrete network topology and event that violate the safety properties. It is well known that tools like Z3 are not guaranteed to terminate in general. However, we notice that the generated VCs for many network protocols belong to a class of formulas that are easy to verify by SAT solvers.

We show that VeriCon is practical for a large repertoire of controller programs, ranging from simple but well-studied examples, e.g., MAC-learning switches and firewalls, to simplified versions of recently proposed SDN-based systems: Resonance [19] for dynamic access control and Stratos [8] for orchestrating the steering of traffic through middlebox sequences. VeriCon is compositional, in the sense that it checks the correctness of each network event independently against the specified invariant, and so it can scale to handle complex systems well beyond other approaches, e.g., finite state model checking, given appropriate inductive invariants.

**Organization** Section 2 provides an informal overview of VeriCon. Section 3 shows how admissible topologies and network-wide invariants in SDNs can be formulated in first-order logic. Section 4 discusses how to generate verification conditions for controller programs. To simplify exposition, we limit the discussion in this section to the small core language CSDN. Section 5 presents the implementation of VeriCon and our experience with verifying SDN controllers. Section 6 reviews related work.

## 2. Overview

This section provides an overview of the VeriCon system.

### 2.1 Programming Controllers

Fig. 1 shows a simple SDN program implementing a stateful firewall, inspired by [12]. The idea here is that end-hosts in the corporate domain can send traffic to the outside world but, for security reasons, traffic from an end-host outside the domain can only enter the domain if that end-host has previously received traffic from some end-host within the domain. This is realized as follows. Two types of hosts are connected to a switch: (i) trusted hosts (within the organization) via port 1; and (ii) untrusted hosts (outside the organization) via port 2. Packets from trusted hosts are always forwarded to trusted hosts. Packets from untrusted hosts are forwarded to trusted hosts only if the source host has previously received a packet from a trusted host. The auxiliary relation pktln records the trusted hosts for each switch. We use bold font to denote OpenFlow commands. The program is actually executed in an infinite loop with two types of events: pktln events that are annotated with commands, and pktFlow events whose semantics is determined by the current content of the flow table.

Fig. 2 shows a typical topology and Table 1 depicts a particular scenario of network events and their effects.

### 2.2 Correctness Checking

VeriCon receives three inputs: (i) a SDN program, (ii) a first-order formula describing constraints on the topology, and (iii) a correctness condition expressed as invariants in first-order logic. It verifies that for every event executed in an arbitrary topology the program satisfies the required invariants by means of a theorem prover. In the firewall example, the requirement is that for every packet sent from an untrusted host to a trusted host there exists a packet sent to that untrusted host from some trusted host. This is
rel \( \tau r(SW, HO) = \{ \} \)
@while new events occur do {
    pktIn\(s, src \rightarrow dst, prt(1)\) ⇒
    s.forward\((src \rightarrow dst, prt(1)) \rightarrow prt(2)\)
    \(\tau r\).insert\((s, dst)\)
    pktIn\((s, src \rightarrow dst, prt(1) \rightarrow prt(2))\)
    @pktFlow\((s, src \rightarrow dst, prt(1))\)
    @forward according to the table
    pktIn\((s, src \rightarrow dst, prt(2))\) ⇒
    if \( \tau r(s, src) \) then
        s.forward\((src \rightarrow dst, prt(2) \rightarrow prt(1))\)
    pktIn\((s, src \rightarrow dst, prt(2))\) ⇒
    @forward according to the table
@}

Figure 1. A simple stateful firewall monitoring the traffic from untrusted hosts connected to \( prt(2) \) to trusted hosts connected to \( prt(1) \). Statements beginning with @ are added for expository purposes.

<table>
<thead>
<tr>
<th>event</th>
<th>action</th>
<th>new-rule</th>
<th>new-trusted</th>
</tr>
</thead>
<tbody>
<tr>
<td>pktIn((s, c \rightarrow b, prt(2)))</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>pktIn((s, a \rightarrow c, prt(1)))</td>
<td>s.forward((a \rightarrow c, prt(1) \rightarrow prt(2)))</td>
<td>(\langle s, a \rightarrow c, prt(1) \rightarrow prt(2)\rangle)</td>
<td>(\langle s, c\rangle)</td>
</tr>
<tr>
<td>pktIn((s, c \rightarrow b, prt(2)))</td>
<td>s.forward((c \rightarrow b, prt(2) \rightarrow prt(1)))</td>
<td>(\langle s, c \rightarrow b, prt(2) \rightarrow prt(1)\rangle)</td>
<td>none</td>
</tr>
<tr>
<td>pktFlow((s, c \rightarrow b, prt(2) \rightarrow prt(1)))</td>
<td>s.forward((c \rightarrow b, prt(2) \rightarrow prt(1)))</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>

Table 1. A scenario of a packet transmissions for the SDN program shown in Fig. 1 using the topology shown in Fig. 2.

captured by the following safety invariant
\[ I_1 = S.\text{sent}(Src \rightarrow Dst, prt(2) \rightarrow prt(1)) \Rightarrow \exists Src' \in HO : S.\text{sent}(Src' \rightarrow Src, prt(1) \rightarrow prt(2)) \]

To improve readability, free variables in formulas (such as \( S \) and \( Src, Dst \)) are implicitly universally quantified. \( S \) denotes an arbitrary switch; \( Src, Dst \), and \( Src' \) denote arbitrary hosts. Finally, \( S.\text{sent}(Src \rightarrow Dst, I \rightarrow O) \) denotes the fact that a packet from source \( Src \) to destination \( Dst \) was forwarded from input port \( I \) to output port \( O \) of the switch \( S \).

2.2.1 Unbounded Symbolic Topologies

VeriCon, unlike finite-state model checking (e.g., [3]), verifies that the invariants hold under any admissible network topology of any size. By default, the admissible topologies are all the possible network graphs, but the programmer may choose to force a certain class of admissible topologies; e.g., one can enforce a star shape by requiring that there exists a switch to which all the other switches are connected by a link:

\[ \exists S \in SW : \forall S_1, S_2 \in SW : S_1 \neq S_2 \Rightarrow (I_1, I_2 \in PR : link(S_1, I_1, I_2, S_2)) \leftrightarrow S_1 = S \lor S_2 = S \]

Here \( link(S_1, I_1, I_2, S_2) \) denotes the fact that port \( I_1 \) of switch \( S_1 \) is connected to port \( I_2 \) of switch \( S_2 \).

VeriCon first checks that the safety invariant and topology constraints are consistent. In the example, \( I_1 \) can be trivially satisfied by an empty topology; conjoined with the star constraint, it can be satisfied by a topology of size 1. Note that we can conjon different requirements and let the theorem prover infer a common topology.

VeriCon then continues to check that the invariants are preserved by executions of arbitrary switch and controller events on an arbitrary network subject to the topology constraints. VeriCon provides a compositional way to check complex code under arbitrary event sequences. However, this requires the specification of inductive invariants\(^1\). In the firewall example above, \( I_1 \), though correct, is not inductive; VeriCon displays a counterexample for each event that violates the invariant. For example, VeriCon generated the counterexample shown in Fig. 3 for the switch event (pktFlow). It describes a configuration in which the switch’s flow table is unconstrained and permits forwarding from untrusted hosts to trusted ones. In this case, it is not a bug in the code—this situation cannot occur at runtime, but \( I_3 \) is not strong enough to exclude it. Another counterexample, which VeriCon produces (Fig. 4), corresponds to situations in which the trusted relation contains superfluous entries when entering a controller event (pktIn). We strengthen \( I_1 \) to be inductive by adding the following two safety invariants that exclude the above situations:

\[ I_2 = S.flt(Src \rightarrow Dst, prt(2) \rightarrow prt(1)) \Rightarrow \exists Src' \in HO : S.\text{sent}(Src' \rightarrow Src, prt(1) \rightarrow prt(2)) \]

\[ I_3 = \tau r(S, H) \Rightarrow \exists Src \in HO : S.\text{sent}(Src \rightarrow H, prt(1) \rightarrow prt(2)) \]

Safety invariant \( I_2 \) uses the relation symbol \( flt \) to refer to a potentially infinite relation describing the flow tables of the individual switches. It states that flow table entries only contain forwarding rules from trusted hosts. \( I_3 \) states that the controller data structure \( \tau r \) records the correct hosts. These kind of invariants are common in many SDN programs.

VeriCon reports that \( I_1 \land I_2 \land I_3 \) is inductive, and the controller program is verified. Notice that when VeriCon proves that an invariant holds, that invariant is guaranteed to hold for an arbitrary sequence of network events. For example, \( I_3 \) also guarantees that packets forwarded by the switch are sent by certified hosts.

Bug Finding VeriCon can identify subtle bugs in an SDN program. When VeriCon is applied to an incorrect SDN program, it produces a concrete counterexample in a readable manner. VeriCon shows the event that violates the safety invariant and the error configuration,

\(^1\) An invariant is inductive if it is preserved by executions starting from an arbitrary state satisfying the invariant.
2.2.2 Inferring Inductive Invariants

In general, writing inductive invariants by hand is very tricky since the programmer needs to specify the set of states after an arbitrary sequence of events. Therefore, VeriCon includes a separate simple utility for inferring invariants using iterated weakest preconditions [5]. The main idea is to strengthen the goal invariants by arbitrary executions of controller and switch events. We start with the goal invariant and perform backward analysis on the event handler code (including flow events). For example, \( I_2 \) is the weakest precondition which guarantees that \( I_1 \) holds after executing the semantics of the switch event \( \text{pktFlow}(s, \text{src} \to \text{dst}, \text{prt}(2)) \). Also, \( I_2 \) is the weakest precondition which guarantees that \( I_1 \) holds after executing the controller command associated with the \( \text{pktIn}(s, \text{src} \to \text{dst}, \text{prt}(2)) \) event. Therefore, VeriCon can infer the inductive invariant from the \( I_2 \) specification. As shown in Section 5, inductive invariants can often be inferred for simple SDN programs using few strengthening iterations. However, for more complicated programs it may be necessary to apply more advanced invariant inference techniques.

2.3 Limitations

Our current verification methodology is limited in two ways:

- We focus on safety properties. We leave the verification of liveness properties, e.g., that packets must eventually reach their destinations, for future research.
- We assume that events are executed atomically, ignoring out-of-order rule installations. Consistently updating a software-defined network is an important challenge in SDN (see [22]). We plan to address this issue in the future by considering interleavings of rule installations without barriers.

3. Symbolically Modeling SDN Properties with First-Order Logic

We now show that many interesting properties of software-defined networks can be naturally expressed in first-order logic. We use relations to model the network topology, the flow tables of the switches, the SDN program’s internal state, as well as histories of transmitted packets. Each of the relations is potentially infinite. CSDN commands manipulate relations by inserting and removing tuples from relations. Queries over network states are defined using first-order formulas.

3.1 Predefined Relations

Table 2 describes the predefined relations supported by VeriCon. In addition, the programmer also can define her own relation symbols. The relations \( \text{link}(S, O, H) \), \( \text{link}(S_1, I_1, I_2, S_2) \), and \( \text{path}(S, O, H) \) represent the physical network topology. SDN programs generally do not explicitly manipulate these relations, other than populating them based on link-level discovery protocol (LLDP) information reported by SDN switches.

For clarity, a packet header is represented as a pair \( \text{Src} \to \text{Dst} \). Our implementation supports different packet header fields as functions \( PK \to \text{Values} \). The most interesting built-in relation involving packet headers is \( S, f_t(Src \to Dst, I \to O) \), which represents the switches’ forwarding tables. This relation denotes the semantics of the switch and not its concrete storage. For example, the SDN program command \( s.\text{install}(h \to s, i \to o) \) updates switch \( s \) to include a general forwarding rule for all packets from host \( h \) coming from ingress port \( i \) to be forwarded to port \( o \). This is implemented in OpenFlow by installing a general matching rule based on the source of the packets. The symbolic effect on the \( ft \) relation is to add all the tuples \( S, f_t(Src \to Dst, I \to O) \) where \( S = s, SRC = h, I = i, \) and \( O = o \) (\( Dst \) is unconstrained).
Table 2. Built-in relations describing network states.

<table>
<thead>
<tr>
<th>Relation</th>
<th>Attributes</th>
<th>Intended Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>link(S,O,H)</td>
<td>$S \in SW, O \in PR, H \in HO$</td>
<td>Host $H$ is directly connected to switch $S$ via port $O$</td>
</tr>
<tr>
<td>link(S1,I1,I2,S2)</td>
<td>$S_1 \in SW, I_1, I_2 \in PR, S_2 \in SW$</td>
<td>Port $I_1$ of switch $S_1$ is directly connected to port $I_2$ of switch $S_2$</td>
</tr>
<tr>
<td>path(S,O,H)</td>
<td>$S \in SW, O \in PR, H \in HO$</td>
<td>There is a path from switch $S$ via port $O$ to a host $H$</td>
</tr>
<tr>
<td>path(S1,I1,I2,S2)</td>
<td>$S_1, S_2 \in SW, I_1, I_2 \in PR$</td>
<td>There is a path from port $I_1$ of switch $S_1$ to port $I_2$ of switch $S_2$</td>
</tr>
<tr>
<td>S,ft(I → O)</td>
<td>$S \in SW, Src, Dst \in HO, I, O \in PR$</td>
<td>Switch $S$ has a rule to forward packets $Src \to Dst$ arriving from port $I$ to port $O$</td>
</tr>
<tr>
<td>S,pktSent(Src → Dst, I → O)</td>
<td>$S \in SW, Src, Dst \in HO, I, O \in PR$</td>
<td>Packet $Src \to Dst$ arrived at ingress $I$ is forwarded to egress $O$</td>
</tr>
<tr>
<td>rcv$\alpha$(S, Src → Dst, I)</td>
<td>$S \in SW, Src, Dst \in HO, I, O \in PR$</td>
<td>Packet $Src \to Dst$ is received at ingress port $I$</td>
</tr>
</tbody>
</table>

The effect is handled in the generated verification condition using a first-order formula expressing the weakest precondition of this command by substituting the ft relation symbol. Sending to egress port $O = null$ models dropping packets.

The history relation $S$.sent($p$, $I$ → $O$) records packets whose header is $p$ forwarded by a switch $S$ from ingress port $I$ to egress port $O$. This can happen either by executing a forwarding rule at a switch or by sending a packet to the controller which then instructs the switch to forward the packet from $I$ to $O$. This relation records the history of forwarding and is used for reasoning. For example, it occurs in the invariant $I_2$ defined in eq. (2.2).

Note that “$S$.r($\bar{x}$)” is just syntactic sugar for the first-order formula “$r(S, \bar{x})$”, used to enhance readability.

The predicate $rcv^{\alpha}(S, P, I)$ allows assertions to refer to the packet currently being handled by the switch or the controller code. The examples of such assertions include transition invariants, defined below. For a controller event pktFlow($s, p, i$) (respectively, for a switch event pktFlow($s, p, i$ → $o$)), $rcv^{\alpha}(S, P, I)$ holds if and only if $S = s$, $P = p$ and $I = i$.

3.2 Invariants

Fig. 5 shows the syntax of standard (typed) first-order formulas which are used to describe invariants of SDN programs and topology constraints. In the atomic formulas, $Rid$ is either a predefined or a user-defined relation. For readability, we write atomic formulas $r(S, (Src, Dst), I)$ as $S$($Src \to Dst, I$ → $O$), where $S$ denotes an arbitrary switch, $Src$ denotes a source host, $Dst$ denotes a destination host, $I$ is an input port, and $O$ is an output port.

VeriCon supports three kinds of invariants:

(i) The topo invariants define the admissible topologies. These are assumed to hold in the initial state. VeriCon checks that these invariants are consistent with safety and trans invariants and that together they form an inductive invariant that is preserved under the execution of switch and controller events.

(ii) The safety invariants are supposed to hold at the initial state and be preserved for any execution of switch and controller event sequence.

(iii) The trans invariants are checked after the execution of every event. They describe the properties of transitions caused by the event in a similar way to postconditions in procedures.

VeriCon simplifies the verification task by assuming that both switch and controller events are executed atomically. In particular, when the controller executes a sequence of commands the invariant is checked before and after the whole sequence and not after individual commands. It is straightforward to check that the invariant holds after every command. However, this will lead to many false alarms as the code usually assumes atomically.

\[ F ::= \begin{array}{ll}
   True & \text{true} \\
   Trm = Trm & \text{equality} \\
   \neg Trm & \text{negation} \\
   \forall \alpha: \text{Fid,F} & \text{universal quantification} \\
   \exists \alpha: \text{Fid,F} & \text{existential quantification} \\
   F \land F & \text{conjunction} \\
   F \lor F & \text{disjunction} \\
   Trm ::= \alpha & \text{logical variable} \\
   Fid(Trm) & \text{uninterpreted functions}
\end{array} \]

Table 3. Examples of interesting topology invariants for SDNs.

<table>
<thead>
<tr>
<th>Formula</th>
<th>Intended Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$I_1$</td>
<td>no self-loops</td>
</tr>
<tr>
<td>$\neg \text{link}(S_1, I_1, I_2, S_2)$</td>
<td>symmetry of links</td>
</tr>
<tr>
<td>$\text{link}(S_1, I_1, I_2, S_2) \Rightarrow \text{link}(S_2, I_2, I_1, S_1)$</td>
<td>packets arrive from reachable hosts</td>
</tr>
<tr>
<td>$rcv^{\alpha}(S, P, I)$</td>
<td></td>
</tr>
<tr>
<td>$\text{pr}(M) = \text{pr}(N)$</td>
<td></td>
</tr>
<tr>
<td>$\rightarrow \Rightarrow M \equiv N$</td>
<td></td>
</tr>
</tbody>
</table>

3.2.1 Topology Invariants

First-order logic can express many topology invariants naturally, as shown in Table 3. These invariants are crucial for VeriCon to precisely reason about networks. For example, the invariant $T_3$ asserts that packets cannot be received from disconnected hosts. Without this invariant the theorem prover may issue false messages. Notice that some of these invariants may not be relevant or need not hold for some topologies. The current implementation provides a library of invariants which can optionally be included in the controller code.

3.2.2 Safety Invariants

VeriCon permits safety invariants that define the required consistency of network-wide states. VeriCon checks that every event preserves all the topology and safety invariants. In the firewall example, we check that $\mathcal{I}_1 \land \mathcal{I}_2 \land \mathcal{I}_3$ is preserved by the execution of flow and controller events.

3.2.3 Transition Invariants

VeriCon also permits the programmer to define transition invariants, which describe the effect of executing event handlers. VeriCon checks that all the transition invariants are satisfied after the execution of every switch and controller event.
rel connected (SW, PR, HO) = { } // new relation with 3 args
pktn (s, src → dst, i) ⇒ // packet from src into dst
var o : PR
// var. for egress ports
connected.insert (s, i, src) // learn a new connection
if connected (s, o, dst) then // if dest. is already known
s.forward (src → dst, i → o) // forward the packet
s.install (src → dst, i → o) // install a new rule
else s.flood (src → dst, i) // flood if dest. is unknown

Figure 6. A simple learning switch controller code

A simple example of a transition invariant is the absence of “black holes”, i.e., packets are never dropped. This is expressed as

rvcup (S, Src → Dst, I) ⇒ ∀O ∈ PR : S.sent (Src → Dst, I → O)

The latter invariant does not hold in the firewall example since it drops packets from untrusted hosts. However, the invariant does hold for the simple learning switch shown in Fig. 6. This SDN program learns connected hosts as soon as new packets from them appear. When a packet arrives, it is forwarded to the port to which the destination host is connected or, if this port is unknown, the packet is sent via flood to all the ports excluding the input port.

Interestingly, with VeriCon one can even state (and prove) that the learning switch SDN program correctly forwards packets. One way to express this using the transition invariant

trans : rvcup (S, Src → Dst, I) ∧ O ̸= I ∧ path (S, O, Dst) ⇒ S.sent (Src → Dst, I → O)

However, VeriCon reports that this invariant is violated in network topologies where multiple ports can be used to reach hosts. We can restrict the topology by adding an extra topology invariant

topo : path (S, I1, H) ∧ path (S, I2, H) ⇒ I1 = I2

Alternatively, it is possible to state the correctness of the learning switch in networks with multiple outgoing ports using the formula L4 shown in Table 4. Here, O1 quantifies over the existence of path and O2 denotes the port chosen by the flood command or via learning. This invariant is a bit complicated since it deals with situations in which there are multiple ports leading to the destination host. In fact, it is possible to show that the learning switch also correctly learns the connections using the invariants shown in Table 4.

4. Verifying Controller Programs

This section defines the CSDN language, a simple imperative language for writing SDN programs, with an eye towards making verification tractable. The only data structures in CSDN are relations, which model both the internal state of the controller and the flow tables. CSDN commands can query and update the relations. As shown later, updates to relations are expressible using Boolean operations, a fact which greatly simplifies the verification task. As a result, Z3 is able to precisely reason about many interesting properties, as discussed in Section 4.3.

4.1 The CSDN Language

Fig. 7 describes the abstract syntax of CSDN. The SDN program first declares external relations, initializes these relations, and specifies constraints on the network topology. Unless the programmer restricts the set of admissible topologies (using the keyword topo), all topologies are admissible. The programmer can initialize user-defined relations and specify controller-specific invariants. The built-in relations sent, rece, and ft are initially empty by default. Events have attributes that include the switch where they occur (denoted by s), the sending host (src), the receiving host (dst), and the ingress port at which the packet arrives at the switch (i).

The controller code reacts to events by performing a sequence of (conditionally executed) commands. The command assume F instructs the verifier to assume that F holds in the sequel, whereas the command assert F causes the verifier to produce an error if F does not hold. The insert and remove commands are used to update a given relation with a set of tuples. The flood command forwards a packet to all switch ports except the packet’s ingress port. For readability, we define an install command as shorthand for updating the flow table of the relevant switch and a forward command to encode sending the current packet, i.e.,

S.install (P, I → O) ⇔ ft.insert (S, P, I → O)

S.forward (P, I → O) ⇔ sent.insert (S, P, I → O)

4.2 From Programs to Formulas

We now show how to convert CSDN programs into first-order formulas, which can be fed into the theorem prover. We use the standard Dijkstra’s weakest (liberal) precondition calculus, originally invented for specifying the meaning of guarded commands [5]. The main idea is to compute a weakest formula in first-order logic that ensures that execution of a command c leads to the state satisfying a postcondition Q. Such formula is called the weakest liberal precondition wp[c] (Q). The formula wp[c] (Q) can be used to check the correctness of c for a given precondition P by asserting that P ⇒ wp[c] (Q). Alternatively, models of P ∧ ¬ wp[c] (Q) are counterexamples to the fact that Q holds when c is executed on states satisfying P. Finally, I is an inductive invariant for a command c if and only if I ⇒ wp[c] (I).

Notice that these rules only apply to prove the safety of the networks.

Table 5 contains syntax directed rules for computing the weakest (liberal) preconditions of CSDN commands. These rules compute a formula representing the largest set of states from which a command executes without failure, thus defining the axiomatic meaning of CSDN commands. The first section defines the meaning of atomic commands. The second section defines the meaning of controller and switch events. For brevity, we omit the rules for the while-loops (e.g., see [7]), which assert that (i) the loop invariant initially holds, (ii) that the loop invariant is preserved by the loop body, and (iii) that the loop invariant and the negation of program condition imply the postcondition.

It is interesting to note that destructive updates to relations (insertions and removals) are simply handled using Boolean operations.

An alternative is to use a version of McCarthy’s store functions specialized to updating relations in the way Table 5 prescribes. However, in our case, this just introduces more overhead than savings because relations are never passed as first-class objects and there are no nested access patterns, where McCarthy stores are known to provide a more succinct representation.

Controller events pktn (s, p, i) are assumed to be triggered when a packet arrives at a switch and there is no entry in the flow table for handling this packet. Switch events pktnflow (s, p, i → o) represent a new packet arriving at a switch and being handled according to an existing entry in the flow table. Observe that this existing packet-handling rule can use the output port mu11 to drop the packet. We will slightly abuse the notation, as flow events are implicit and do not include commands. We call events and their commands guarded commands.

Priorities The OpenFlow standard supports priorities that allow a programmer to install some rules without removing existing rules. Only the flow rule with the maximal priority is executed. We implement flow tables with priorities by including an extra column in ft. Accordingly, the semantics of the flow event from Table 5 is
<table>
<thead>
<tr>
<th>I</th>
<th>Formula</th>
<th>Intended Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$L_1$</td>
<td>$S.f(t(Src \rightarrow Dst, I \rightarrow O) \Rightarrow path(S, O, Dst)$</td>
<td>Correctly learned connections</td>
</tr>
<tr>
<td>$L_2$</td>
<td>connected$(S, I, H) \Rightarrow path(S, I, H)$</td>
<td>Consistent SDN program data structure</td>
</tr>
<tr>
<td>$L_3$</td>
<td>$S.f(t(Src \rightarrow Dst, I \rightarrow O) \Rightarrow connected(S, I, Src) \land connected(S, O, Dst)$</td>
<td>Consistent learning</td>
</tr>
<tr>
<td>$L_4$</td>
<td>$rcv_{\text{out}}(S, Src \rightarrow Dst, I) \land (3O_1 \in PR : O_1 \neq I \land path(S, O_1, Dst)) \Rightarrow \exists O_2 \in PR : path(S, O_2, Dst) \land S.sent(Src \rightarrow Dst, I \rightarrow O_2)$</td>
<td>Guaranteed forwarding</td>
</tr>
</tbody>
</table>

Table 4. Invariants for the learning switch. $L_1$, $L_2$, $L_3$ are safety invariants and $L_4$ is a transition invariant.

| Ctrl ::= | Init $(E \Rightarrow Cmd)^*$ | controller |
| Init ::= | rel $Rid(Tid^*)$ | declare relation |
| rel $Rid(Tid^*) = (Pred^*)^*$ | with initialization |
| var $Id : Tid$ | new variable |
| topo $Fid : F$ | topology invariant |
| inv $Fid : F$ | safety invariant |
| trans $Fid : F$ | transition invariant |
| Event ::= | $\text{packet-in event}$ |
| Exp ::= | $\text{const } Id$ |
| Cond ::= | True | boolean values |
| Cond ::= | False | variable assignment |
| Cond ::= | $\neg Cond$ | sequence |
| Cond ::= | $\land Cond$ | block of commands |
| Cond ::= | $\lor Cond$ | restrict to values |
| Var ::= | $\text{var } Id : Tid$ | local variable |
| Cmd ::= | $\text{skip}$ | do nothing |
| Expr ::= | $\text{Id.flood } (Exp^*)$ | flood to all ports |
| Cond ::= | $\text{assert } F$ | assume that $F$ holds |
| Cond ::= | $\text{Rid.insert } (Pred^*)$ | relation insertion |
| Cond ::= | $\text{Rid.remove } (Pred^*)$ | relation removal |
| Exp ::= | $\text{Cmd}$ | conditional |

Figure 7. The abstract syntax of CSDN. $F$ is a first-order formula defined in Fig. 5.

$$\text{wp}[skip](Q) \equiv Q$$
$$\text{wp}[assume](F) \equiv F \Rightarrow Q$$
$$\text{wp}[assert](F) \equiv F \land Q$$
$$\text{wp}[r.insert](P)(Q) \equiv Q[r(x) \lor [P]_{FO}(r(x))]$$
$$\text{wp}[r.remove](P)(Q) \equiv Q[r(x) \land \neg[P]_{FO}(r(x))]$$
$$\text{wp}[s.flood](P, I, O) \equiv Q[S.sent(P, I, O) \lor (S = s \land P = p \land I = i \land O \neq o \land O \neq null)/S.sent(P, I, O)]$$
$$\text{wp}[if b then c_1 else c_2](Q) \equiv (b \Rightarrow \text{wp}[c_1](Q)) \land \neg b \Rightarrow \text{wp}[c_2](Q))$$
$$\text{wp}[wp](Q) \equiv \text{wp}[wp](Q)$$

Table 5. Rules for computing weakest (liberal) preconditions for CSDN programs. $Q[\psi/\varphi]$ denotes the substitution of all occurrences of $\varphi$ in $Q$ by $\psi$. The meaning of predicates $[P]_{FO}$ is a first-order formula over $r$’s columns defined in Table 6.

$$[exp]_{FO}(t) \equiv exp = t$$
$$[s]_{FO}(t) \equiv True$$
$$[P_1 \land P_2]_{FO}(t) \equiv [P_1]_{FO}(t) \land [P_2]_{FO}(t)$$
$$[P_1, P_2, \ldots, P_k]_{FO}(t_1, t_2, \ldots, t_k) \equiv \bigwedge_{i=1}^{k}[P_i]_{FO}(t_i)$$

Table 6. Converting predicates into first-order formulas. The meaning of predicates $[P]_{FO}$ is a first-order formula over the columns of the relation.

modified as

$$\text{wp}[wp\text{Flow}](s, p, i, o) \Rightarrow \text{“forward”}[Q] \equiv (rcv_{\text{out}}(s, p, i) \land \exists \alpha : PRI. \text{maxft}(\alpha, s, p, i, o)) \Rightarrow \text{wp}[s.foward](p, i, o)](Q)$$

with $PRI = \text{Nat}$. Here, predicate $\text{maxft}(\alpha, s, p, i, o)$ denotes the fact that $\alpha$ is the maximal priority for the matching flow rules, i.e., $s.f(t(\alpha, p, i \rightarrow o))$ and

$$\forall \alpha' : PRI, O' : PR. s.f(t(\alpha', p, i \rightarrow O')) \Rightarrow \alpha' \leq \alpha.$$

4.3 From Formulas to Theorems and Models

Our evaluation in Section 5 shows that the verification conditions can be solved by Z3. The invariants we encountered so far were all restricted to formulas with a quantifier prefix $\exists \forall$. Proving such invariants could easily be highly intractable and we explain the apparent ease based on the following observation:

Observation: Instantiation dependencies are shallow. By inspecting verification conditions from CSDN programs we observed that the formulas could be proved or disproved using relatively few instantiations. The reason is that the formulas do not contain opportunities for pumping. In other words, instantiations do not produce
We implemented the pseudocode shown in Fig. 8 in Python using programs annotated with: (i) topology invariants; (ii) safety invariants; and (iii) transition invariants. The VC generator uses the rules of Table 5 to compute verification conditions.

The tool accepts as input a program code and the invariants for both correct and incorrect programs and to programs with incorrect invariants. Section 5.3 describes our experience applying VeriCon to incorrect programs and to programs with incorrect invariants.

All experiments were performed on an Intel i5 1.3GHz, 4GB, MacBook Air running OS 10.8.5.

5.1 Implementation

We implemented the pseudocode shown in Fig. 8 in Python using the Z3 Python API. We use PLY (Python Lex-Yacc) to parse CSDN programs annotated with: (i) topology invariants; (ii) safety invariants; and (iii) transition invariants. The VC generator uses the rules of Table 5 to compute verification conditions.

The tool accepts as input a CSDN program $Prog$ and a number $n_{\text{max}}$ limiting the depth of invariant strengthening (default value is 0). First, we check that the topology constraints are consistent with the initial states of the network. Then, for each value $n$ from 0 to $n_{\text{max}}$, we proceed as follows: We strengthen the safety invariants with $n$ applications of $wp$ for all events of $Prog$. We check that the strengthened safety invariants hold for the initial states under the topological assumptions. We then check that the topology, the strengthened safety invariants, and the transition invariants are preserved by the execution of arbitrary events executed on every

VeriCon($Prog$, $n_{\text{max}}$)

Let $\text{Topo}$ be the set of topology invariants in $Prog$
Let $\text{Inv}$ be the set of safety invariants in $Prog$
Let $\text{Trans}$ be the set of transition invariants in $Prog$
Let $\text{Event}$ be the set of events of $Prog$
Let $\text{Init}$ be the formula describing the initial states of $Prog$
if not SAT($\text{Init} \land (\bigwedge \text{Topo})$)
  then return topology and initial conditions are incompatible
for $n = 0$ to $n_{\text{max}}$ do
  Let $\text{Inv}^\# = \{\text{Str}^{(n)}(I, \text{Event}) \mid I \in \text{Inv}\}$ // strengthened invariants
  Let $\text{Ind} = \bigwedge (\text{Inv}^\# \cup \text{Topo})$ // candidate inductive formula
  if there exists $I \in \text{Inv}^\#$ s.t. SAT($\text{Ind} \land (\bigwedge \text{Topo}) \land \neg I$)
    then report $I$ does not hold on initial states
  if there exist $ev \in \text{Event}$ and $I \in \text{Inv}^\# \cup \text{Topo} \cup \text{Trans}$ s.t. SAT($\text{Ind} \land \neg wp[ev](I)$)
    then report $I$ is not provable on event $ev$ using Ind
  else return all proved

Figure 8. A pseudocode for the VeriCon tool.
Table 7. Running times for VeriCon on correct SDN controller programs. LOC — the number of code lines where MAX is the maximal number of lines per event and TOT is the total number. Rel — the number of user provided relations in the controller code. Inv — specification safety and transition invariants: the number of goal invariants, the number of auxiliary invariants that make goal invariants inductive, and the number of auxiliary invariants automatically inferred by the tool, respectively. VC — the tool generated verification conditions: # — total number of sub-formulas, ∀ — quantifier nesting. Time — the running times for Z3.

<table>
<thead>
<tr>
<th>Program</th>
<th>Description</th>
<th>LOC</th>
<th>Rel</th>
<th>Inv</th>
<th>VC</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firewall</td>
<td>Simple stateful firewall, Fig. 1.</td>
<td>8</td>
<td>3</td>
<td>1</td>
<td>2</td>
<td>998</td>
</tr>
<tr>
<td>StatelessFirewall</td>
<td>Simple stateless firewall, Fig. 9.</td>
<td>4</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>446</td>
</tr>
<tr>
<td>FirewallMigration</td>
<td>Firewall with migrating hosts, Fig. 10.</td>
<td>9</td>
<td>4</td>
<td>1</td>
<td>2</td>
<td>1186</td>
</tr>
<tr>
<td>Learning</td>
<td>Simple learning switch, Fig. 6</td>
<td>8</td>
<td>7</td>
<td>1</td>
<td>2</td>
<td>1251</td>
</tr>
<tr>
<td>Auth</td>
<td>Authentication on the network with a learning controller switch, Section 5.2.3</td>
<td>15</td>
<td>14</td>
<td>4</td>
<td>6</td>
<td>2284</td>
</tr>
<tr>
<td>Resonance</td>
<td>Learning switch with authentication from [19], Section 5.2.4.</td>
<td>93</td>
<td>92</td>
<td>16</td>
<td>5</td>
<td>6319</td>
</tr>
<tr>
<td>Stratos</td>
<td>Forwarding traffic through a sequence of middleboxes [8, 21], Section 5.2.5</td>
<td>29</td>
<td>28</td>
<td>4</td>
<td>3</td>
<td>1493</td>
</tr>
</tbody>
</table>

 pktIn(s, src → dst, prt(1)) ⇒ // packet from a trusted host
 s.forward(src → dst, prt(1) → prt(2)) ⇒ // forward the packet to untrusted hosts
 s.install(s → dst, prt(1) → prt(2)) ⇒ // insert a rule to forward future packets targeted at dst
 s.install(dst → s, prt(2) → prt(1)) ⇒ // insert a rule to forward future packets coming from dst

Figure 9. A simple stateless firewall monitoring the traffic from untrusted to trusted hosts.

rel tr(HO) = {} // declare a relation of trusted hosts (initially empty)
pktIn(s, src → dst, prt(1)) ⇒ // packet from a trusted host
s.forward(src → dst, prt(1) → prt(2)) ⇒ // forward the packet to untrusted hosts
tr.insert(dst) // insert dst into trusted controller memory
tr.insert(src) // insert src into trusted controller memory
s.install(src → dst, prt(1) → prt(2)) ⇒ // insert a per-flow rule to forward future packets
pktIn(s, src → dst, prt(2)) ⇒ // packet from a presumably untrusted host
if tr(src) then
 s.forward(src → dst, prt(2) → prt(1)) ⇒ // forward the packet to trusted hosts
 s.install(src → dst, prt(2) → prt(1)) ⇒ // insert a per-flow rule to forward future packets

Figure 10. A simple stateful firewall monitoring the traffic from untrusted to trusted hosts with migrating hosts.

var authServ : HO // a designated host is an authentication server
rel auth(HO) = {authServ} // declare a relation of authenticated hosts (initially, contains authServ only)
rel connected(SW, PR, HO) = {} // declare a relation with three arguments to store learned connections
pktIn(s, src → dst, i) ⇒ // unknown packets
connected.insert(s, i, src) // learn a new connection
if src = authServ then
 auth.insert(dst) // received a message from the authentication server
auth.insert(dst) // destination is now authenticated
if auth(src) ∧ auth(dst) then // got a packet from an authenticated host to an authenticated host
 var o : PR // a local variable for egress port
if connected(s, o, dst) then // destination of the packet is already learned
 s.forward(src → dst, i → o) // forward the packet
 s.install(src → dst, i → o) // install a new rule
else s.flood(src → dst, i) // otherwise flood the packet
else
if dst = authServ then // the sender is not authenticated (hence, it is not authServ)
s.flood(src → dst, i) // flood the packet

Figure 11. A learning switch controller code with authentication.
5.2.4 The Resonance Example
We implemented a simplified version of Resonance [19], an access control approach for host authentication in enterprises. Unlike the original work, we do not model redirection of the web-traffic.

In Resonance, a host could be in one of four states: (i) Registered, (ii) Authenticated, (iii) Operational, or (iv) Quarantined. For each state there are dedicated management servers, and a host is only allowed to communicate with the servers responsible for its current state. Once a host is marked as operational, we allow it to communicate with other operational hosts.

Transition between host states is controlled by the network management servers, which notify the controller on changes in the authentication procedure. The host becomes authenticated if the authentication servers approve its registration, and operational when it gets scanned, by some scanning server, and is found to be free of vulnerabilities. As the scanning servers perform random scans on hosts, a host may become quarantined at any given time (if it was authenticated before).

The key invariants we verified are (1) the installed flow rules satisfy the access policy, and (2) all packet flows in the network respect the policy, i.e., a packet is dropped if and only if it violates the policy.

5.2.5 The Middlebox Composition Example
We implemented a simple application to forward traffic through a sequence of middleboxes, similar to the traffic steering performed by Stratos [8] and SIMPLE [21]. However, unlike these more advanced frameworks, our implementation does not handle middleboxes that modify packet headers or terminate connections, nor does our application perform any weighted selection of middlebox instances. Furthermore, we reactively install rules for each middlebox when the first packet of each flow (where a flow is defined by a pair of source and destination IP addresses) is emitted by the previous middlebox in the sequence. Currently, we have only considered a simple case of one switch in the network. We have verified that the flow table is consistent with the specification of Stratos’ chains which implies that (i) all packets of a flow traverse an instance of each middlebox in the sequence, and (ii) all packets of a flow (in both the forward and backward directions) traverse the same set of instances throughout the lifetime of the flow.

5.3 Buggy Examples
We also applied the tool to erroneous programs and to programs with incorrect assertions. The results, including run-time statistics, formula sizes, and topology sizes, are reported in Table 8.

A simple kind of bug which can occur in the learning switch (Program Learning-NoSend) is when the SDN program forgets to send a message when the destination is known, i.e., the send command is omitted. VeriCon detects that the controller event violates the \( L_4 \) transition invariant and generates the counterexample shown in Fig. 12.

Another type of bug that can occur in SDN programs is that the data structure of the SDN program is inconsistent with the forwarding tables of the switches. For example, in program AuthNoFlowRemoval, we enhanced the network authentication controller with the ability to remove hosts. VeriCon uncovered a bug of a \( \text{pktIn}(s, src \rightarrow dst, in) \) event violating the authentication protocol. This bug occurred because the forwarding rules were not removed from the switch forwarding tables, rendering re-authentication impossible.

6. Related Work
The past few years have witnessed a surge of interest in SDNs. We now discuss the work most relevant to ours along these lines.

Figure 12. A counterexample which shows a scenario where the programmer forgot the line \texttt{forward}(...) of Fig. 6, causing a black hole — a packet may be lost.

Language abstractions. [6, 25] introduce abstractions for programming controllers in order to simplify the task of programming controllers. [10] shows that the compiler from a high-level language, called NetCore, to OpenFlow generates semantically equivalent code. [20] defines a nice declarative language to ease the task of programming and verifying SDN programs. NetKAT [2] presents an equational calculus for imperative, finite state, SDN programs. Declarative programming is also successfully used for updating multi-tenant networks [16].

In contrast, our focus is on the orthogonal problem of verifying the safety of infinite state SDN programs. The ultimate goal is to verify SDN programs in stylized Java or Python, but we focus on an imperative language like CSDN which captures the essence of imperative SDN programming. In the future it is worthwhile to apply VeriCon to declarative programs.

Finite-state model checking of SDN programs. NICE [3] was the first system to use finite-state model checking to verify the correctness of SDN controllers. The SDN program is modeled as a state-transition system with events similar to those in VeriCon. The concrete network topology is also explicitly modeled. Finite state model checking has many advantages over Hoare style verification: it does not require inductive invariants, and it can employ simpler verification technology. It can be easily applied to arbitrary programs. However, it is unsound in the sense that it can never prove the absence of errors in the infinite state SDN program. Also, it is hard to scale finite state model checking to realistic networks. In contrast, we use first-order logic to model the admissible topology and network-wide invariants. Consequently, VeriCon is able to verify the absence of errors and can also potentially handle huge topologies. We note, however, that VeriCon relies on user-provided invariants and a first-order (potentially non-terminating) theorem prover. Our preliminary experience with Z3 is fairly positive.

Finite-state model checking has also been applied to verify SDN programs on large networks [23]. Two examples, the learning switch and the stateful firewall from this work, use manual abstractions. Our results establish that verification with VeriCon (with infinite states) is orders of magnitude faster than the approach in [23] (0.13s vs. 6.835s for the finite-state abstraction).

Verificare [24] also uses finite state CTL model checking. The FlowLog [20] system also employs finite-state model checking by limiting the number of packets sent. It also shows that for some specifications this bound suffices to obtain sound results.

In [17], NICE was extended to perform concolic testing [9] and thus reduce the number of missed bugs.

Checking invariants by analyzing snapshots of the network. [14] suggests a novel method for checking certain network proper-
ties by analyzing packet headers. As with the above schemes, the approach described in [14] can establish the existence of bugs but not their absence.

**Dynamically checking SDN programs.** [13, 15] show how to monitor the correctness of certain properties of SDN programs in real time. The main challenge is to guarantee that this can be accomplished without harming network performance. VeriCon runs at compile-time, and so it verifies correctness or, alternatively, exhibits errors before the code is actually executed. We view controller code verification ala VeriCon and dynamic checking as in [13, 15] as two complementary approaches.

### 7. Conclusion

We presented VeriCon, the first verification tool for (infinite-state) SDN programs. VeriCon reflects two fundamental choices: (i) express controller programs as imperative event-driven programs that manipulate relations; and (ii) express network-wide invariants as first-order logic formulas. From a verification perspective, these two choices fit together well, since they guarantee that the generated verification conditions are simple enough to be expressible in a weak form of first-order logic, which enables complete verification via SMT solvers. Indeed, our results establish that Z3 is able to rapidly verify network-wide invariants of interesting controller programs or, alternatively, to quickly compute a concrete counterexample. VeriCon’s encouraging ability to prove the functional correctness of controller programs of interest motivates further research along these lines.

### Acknowledgments

We thank Ras Bodik, Ahmed Bouajjani, Nate Foster, Oded Padon, Brandon Heller, Ori Lahav, Aurojit Panda, Hossein Hojjat, Peyman Kazemian, Shriram Krishnamurthi, Teemu Koponen, Ratul Mahajan, Tim Nelson, Mark Reitblatt, Vyas Sekar and Sharon Shoham, for their insightful comments on earlier versions of this paper. The research of Itzhaky, and Sagiv has received funding from the European Research Council under the European Union’s Seventh Framework Program (FP7/2007–2013) / ERC grant agreement n° [321174-VSSC]. The research of Karbyshev was funded by Technical University of Munich. Karbyshev thanks Prof. Sagiv for inviting him to visit Tel Aviv University. Part of this work was done while Sagiv was visiting Microsoft Research.

### References


