Event-driven Network Programming

Jedidiah McClurg
CU Boulder, USA
jedidiah.mcclurg@colorado.edu

Hossein Hojjat
Cornell University, USA
hojjat@cornell.edu

Nate Foster
Cornell University, USA
jnfoster@cs.cornell.edu

Pavol Černý
CU Boulder, USA
pavol.cerny@colorado.edu

Abstract

Software-defined networking (SDN) programs must simultaneously describe static forwarding behavior and dynamic updates in response to events. Event-driven updates are critical to get right, but difficult to implement correctly due to the high degree of concurrency in networks. Existing SDN platforms offer weak guarantees that often break application invariants, leading to problems such as dropped packets, degraded performance, security violations, etc. This paper introduces *event-driven consistent updates* that are guaranteed to preserve well-defined behaviors when transitioning between configurations in response to events. We propose *network event structures* (NESs) to model constraints on updates, such as which events can be enabled simultaneously and causal dependencies between events. We define an extension of the NetKAT language with mutable state, and give semantics to stateful programs using NESs. We discuss strategies for implementing NESs using SDN switches, and prove them correct. Finally, we evaluate our approach empirically, demonstrating that it gives well-defined consistency guarantees while avoiding expensive synchronization and packet buffering.

1. Introduction

Software-defined networking (SDN) allows network behavior to be specified using logically-centralized programs that execute on general-purpose machines. These programs react to events such as topology changes, traffic statistics, receipt of packets at a switch, etc. by modifying the set of forwarding rules installed on switches. SDN programs can implement a wide range of advanced network functionality including fine-grained access control [7], network virtualization [21], traffic engineering [14, 15], and many others.

Although the basic SDN model is simple, building sophisticated applications is challenging in practice. Programmers must keep track of numerous low-level details such as encoding configurations into prioritized forwarding rules, processing concurrent events, managing asynchronous events, dealing with unexpected failures, etc. To address these challenges, a number of domain-specific network programming languages have been proposed [2, 9, 18, 20, 28, 29, 33, 34, 36]. The details of these languages vary, but they all offer higher-level abstractions for specifying behavior—e.g., using mathematical functions, boolean predicates, relational operators, etc.—and rely on a compiler and run-time system to generate and manage the underlying network state.

Unfortunately, the languages that have been proposed so far lack critical features that are needed to implement dynamic, event-driven applications. Static languages such as NetKAT [2] offer rich constructs for describing network configurations, but lack features for responding to events and maintaining internal state. Instead, programmers must write a stateful program in a general-purpose language that generates a stream of NetKAT programs. Dynamic languages such as FlowLog and Kinetic [20, 29] offer stateful programming models, but they do not specify how the network behaves while it is being reconfigured in response to state changes. Abstractions such as consistent updates do provide strong guarantees during periods of reconfiguration [25, 31], but current realizations are limited to properties involving a single packet (or set of related packets, such as a unidirectional flow). To implement dynamic SDN applications today, the most effective option is often to use low-level APIs, forgoing the benefits of higher-level languages entirely.

Example: Stateful Firewall. As an example to illustrate the challenges that arise when implementing dynamic applications, consider a simple topology where an internal host H_1 is connected to switch s_1, an external host H_4 is connected to a switch s_4, and switches s_1 and s_4 are connected to each other (see Figure 7(a)). Now suppose we wish to implement a stateful firewall: at all times the internal host H_1 should be allowed to communicate with the external host H_4, but H_4 should only be allowed to communicate with H_1 if H_1 previously initiated a connection. Even implementing this simple application turns out to be quite difficult, because it involves coordinating behavior across multiple devices and packets. For example, suppose that upon receiving a packet from H_1 at s_4, the program might issue a command to install a forwarding rule on s_4 allowing traffic to flow from H_4 back to H_1. While the forwarding rule is being installed, the packet will be forwarded to H_4. Hence, it is likely that the response from H_4 will be dropped at s_4, which is incorrect from the perspective of the application. The root cause of this error is that the network *fails to provide strong semantic guarantees during periods of transition between configurations in response to network events*. An eventual guarantee is not sufficiently strong to implement the stateful firewall correctly, and even a consistent update would not help—it only says what must happen to each individual packet, whereas the application requires propagating the update to the switch instantly. Clearly, a different approach is needed.

Event-driven Consistent Updates. This paper proposes a new semantic correctness condition that gives clear guarantees about updates triggered by events. This condition provides a specification of how the network should behave during updates, and enables—for the first time—precise formal reasoning about stateful network programs.

Formally, an *event-driven consistent update* is a triple $C_i \xrightarrow{e} C_f$, where C_i and C_f are configurations and e is an event. Intuitively, the configurations describe the forwarding behavior of the network before and after the update, while the event represents an occurrence such as the receipt of a packet at a particular switch that triggers the update itself. Semantically, the update ensures that:

1. *Packets are forwarded consistently.* Each packet should be processed by C_i or C_f, but not a mixture of the two.
2. *The network does not update too early.* If every switch traversed by the packet has not heard about the event, then the packet must be processed by C_i.
3. *The network does not update too late.* If every switch the packet traverses has heard about the event, then the packet must be processed by C_f.

The first criterion requires that updates are consistent, which is analogous to the condition proposed previously by Reitblatt et al. [31]. However, consistent updates alone would not provide the guarantees necessary for the stateful firewall example, as they apply only to a single packet, and not to multiple packets in a bidirectional flow. The last two criteria relate the packet-processing behavior on each switch to the events it has "heard about." Note that these criteria leave substantial flexibility for implementations: packets that do not satisfy the second or third condition can be processed by either the C_i or C_f configuration. It remains to define what it means for a switch s to have "heard about" an event e that occurred at switch t (assuming $s \neq t$). We use a causal model and say that s hears about e when a packet, which was processed by t after e occurred, is received at s. This notion can be formalized using the "happens-before" relation.

Returning to the stateful firewall, it is not hard to see that the semantic guarantees offered by event-driven consistent updates are sufficient to ensure the correctness of the overall application. Let us consider an update $C_i \xrightarrow{e} C_f$. In C_i, H_1 can send packet to H_4, but not vice-versa. In C_f, additionally H_4 can send packets to H_1. The event e is the arrival at s_4 of a packet from H_1 to H_4. Now imagine the event e occurs, the and the host H_4 wants to send a packet to H_1 afterwards. Can s_4 drop the new packet as it would have done in the initial configuration C_i? No, because the only switch the packet would traverse is s_4, and s_4 has heard about the event. It follows that the only possible implementation processes the new packet in C_f.

Note that while event-driven updates require immediate responses to local events, importantly, they do not require immediate reactions to events "at a distance." For example, the receipt of a packet in New York does not immediately affect the behavior of switches in London. Intuitively, this makes sense: requiring immediate reaction to remote events would force expensive synchronization between switches and buffering of packets, leading to unacceptable performance penalties. These issues are an instance of the well-known tension between consistency and availability in distributed systems. Existing SDN languages prioritize availability (no expensive synchronization and packet buffering) over consistency (weak guarantees when state changes).

This paper demonstrates that it is possible to provide the same level of availability, while providing a natural consistency condition that is sufficiently powerful to build many applications. We also show that in general, strengthening the condition would force us to weaken availability. Overall, we believe that an abstraction based on (i) a notion of causal consistency that requires that events must be propagated between nodes, and (ii) per-packet consistency that governs how packets are forwarded through the network is a powerful combination that is a natural fit for many applications.

Event-Driven Transition Systems. To specify an event-driven consistent update, we use labeled transition systems called event-driven transition systems (ETSs). In an ETS, each node is annotated with a network configuration and each edge is annotated with an event. For example, the simple stateful firewall application would be described as a two-state event-driven transition system, one representing the initial configuration before H_1 has communicated with H_4 and another representing the configuration after communication has occurred. There would also be a transition between the states upon receipt of a packet from H_1 to H_4 at s_4. This model is similar to the finite state machines used in Kinetic [20]. However, we stipulate that every transition $C_i \xrightarrow{e} C_f$ must be implemented as an event-driven consistent update whereas Kinetic uses best-effort updates. For simplicity, we focus on finite state systems and events corresponding to the packet delivery in this paper. However, these are not fundamental assumptions—our design extends naturally to other notions of event and infinite state systems.

Network Event Structures. The key challenge in implementing event-driven consistent updates stems from the fact that at any time, different switches may have different views of the global set of events that have occurred in the network. Hence, for a given ETS, several different updates may be enabled at a particular moment of time, and we need a way to resolve conflicts. To do this, we use the well-studied model of event structures [37] that constrains transitions in two ways: (1) *causal dependency* which requires that an event e_1 happens before another event e_2 may occur, and (2) *compatibility* which forbids pairs of events that are in some sense incompatible with each other from occurring in the same execution. As an example to illustrate why compatibility constraints are needed, suppose that New York sends packets to London and Paris, but the ETS only allows the first one to receive a packet to respond. Clearly, it would be impossible to implement this behavior without significant coordination. On the other hand, suppose New York and Philadelphia are sending packets to London, and London is allowed to respond only to the one from whom it first received a packet. Clearly, this behavior is easily implementable since the choice is local to London. We use the event structures to rule out non-local incompatible events—i.e., events that are incompatible must concern the same switch.

Implementation. Network event structures also provide a natural formalism for guiding an implementation of updates. Intuitively, we need switches that can record the set of events that have been heard locally, make decisions based on those events, and also transmit events to other switches. Fortunately, in the networking industry there is a trend toward more programmable data planes: mutable state is already supported in most switch ASICs (e.g. MAC learning table) and is also being exposed to SDN programmers in next-generation platforms such as OpenState [4] and P4 [5]. Using these features, we can implement network event structures as follows:

1. Encode the sets of events contained in the network event structure as flat tags that can be carried by packets and tested on switches.
2. Compile the configurations contained in the network event structure to a collection of forwarding tables.
3. Add predicates to the forwarding rules in each configuration that explicitly test for the tag(s) that enable that configuration.
4. Add rules to stamp incoming packets with the tag corresponding to current set of events at ingress (as in as in two-phase updates [31]).
5. Add rules to "learn" which events have happened by reading tags on incoming packets and adding the tags in the local state to outgoing packets, as required to implement the happens-before relation.

We prove that a system implemented in this way correctly implements a network event structure.

Evaluation. To evaluate our design for event-driven consistent updates, we have used our prototype implementation to build a number of event-driven network applications: (a) a stateful firewall, which we have already described; (b) a learning switch that floods packets going to unknown hosts along a spanning tree, but uses point-to-point forwarding for packets going to known hosts; (c) a port knocking gateway that blocks incoming traffic, but allows hosts to gain access to the internal network by sending packet probes to a predefined sequence of ports. Additionally, we have built a synthetic application that forwards around a ring topology to evaluate update scalability. We developed these applications in Stateful NetKAT (code extracts can be found in §5). Our experiments show that our implementation provides competitive performance on several important metrics while ensuring strong consistency properties. We draw several conclusions: First, event-driven consistent updates allow programmers to easily write real-world network applications and get the correct behavior, whereas approaches relying only on best-effort consistency guarantees do not. Second, our design provides high availability, since it does not require any buffering of packets. Third, the performance overhead of maintaining state and manipulating tags (measured in terms of bandwidth) is within 6% of an implementation that provides only best-effort consistency. We also discuss an optimization that exploits common structure in rules across many states to reduce the number of rules that must be installed on switches. In our experiments, a basic heuristic version of this optimization resulted in 32-37% reduction in the number of rules required on average.

Summary. Our main contributions are as follows.

- We propose a new semantic correctness condition called *event-driven consistent updates* that balances the need for immediate response with the need to avoid costly synchronization and buffering of packets.
- We propose *network event structures* as a semantic structure that captures causal dependencies and compatibility between events.
- We describe an implementation based on a stateful extension of NetKAT, and present optimizations that reduce the overhead of implementing stateful programs.
- We conduct experiments showing that our approach gives well-defined consistency guarantees to realistic applications, while avoiding expensive synchronization such as packet buffering.

The rest of this paper is structured as follows: §2 formalizes event-driven consistent updates; §3 defines event transition systems, network event structures, and stateful NetKAT; §4 describes our implementation; §5 presents experiments.

2. Event-driven Network Behavior

This section presents our new consistency model for stateful network programs: event-driven consistent updates.

Preliminaries. A *packet* is a pair (pid, ph) containing a unique ID pid and a record of fields ph, where fields include source and destination addresses, protocol type etc. IDs are only used to streamline the formal definitions below and are not fundamental—in particular, our implementation does not use them. A *switch* sw is a node in the network with one or more *ports* pt. A *host* is a switch that can be a source or a sink of packets. A *location* is a switch-port pair $n{:}m$. Locations may be connected by the links in the topology.

Local forwarding at each switch is dictated by a *network configuration* C. A *located packet* $lp = (pkt, sw, pt)$ is a tuple consisting of a packet, a switch, and a port. Formally, we model the configuration C as a relation on located packets: if $C(lp, lp')$, then the network maps lp to lp', possibly changing its location and rewriting some of its fields. We assume that C respects the links in the topology and does not depend on or modify the ID field. Note that as C is a relation, it may create multiple output packets from a single input.

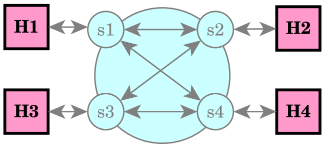


Figure 1: Example topology with four switches and hosts.

A *packet trace* is a sequence of located packets with the same ID—i.e., a trace corresponds to a single packet moving through the network. The set of packet traces that start at a host and can be produced by the network in a single configuration C is denoted by $\mathit{Traces}(C)$.

A *network trace* ntr is a sequence of located packets. Intuitively, a correct network trace will be defined to be an interleaving of packet traces from $\mathit{Traces}(C)$, possibly for different configurations C. Given a network trace ntr and an ID pid, we define a packet trace $ntr{\downarrow}pid$ by removing from ntr all located packets whose ID is not equal to pid.

We now define how the network changes its configuration in response to events. An *event* e is a tuple (eid, φ, sw, pt), where eid is an event identifier and φ is a formula over fields of a packet header. Events model the arrival of a packet with headers ph satisfying φ (denoted $ph \models \varphi$) at location $sw{:}pt$. Note that we could have other events (links coming up or going down, etc.)—anything that a switch can detect could be an event—but for simplicity, we focus on packet events. A located packet $lp_i = ((pid, ph), sw', pt')$ *matches* an event $e = (eid, \varphi, sw, pt)$ (denoted by $lp \models e$) iff $sw = switch' \wedge pt = port' \wedge ph \models \varphi$. Given a network trace $ntr = lp_0 lp_1 \ldots$, the k-th *event occurrence* $eo(ntr, e, k)$ in ntr of an event $e = (eid, \varphi, sw, pt)$ is a located packet lp_j where the index j is the k-th smallest element in the set $\{i \mid lp_i = ((pid, ph), sw, pt) \wedge lp_i \models e\}$ (and $eo(ntr, e, k)$ is undefined for k larger than the size of the set).

Definition 1 (Happens-before relation $\prec_{ntr}$)**.** *Given a network trace $ntr = lp_0 lp_1 \ldots$, the happens-before relation $\prec_{ntr}$ is the least partial order on located packets in ntr that*

- *respects the total order induced by ntr at switches, i.e., $\forall i, j : lp_i \prec lp_j \Leftarrow i < j \wedge lp_i = (pkt, sw, pt) \wedge lp_j = (pkt', sw, pt')$, and*
- *respects the total order induced by ntr for each packet, i.e., $\forall i, j : lp_i \prec lp_j \Leftarrow i < j \wedge lp_i = (pkt, sw, pt) \wedge lp_j = (pkt', sw', pt') \wedge pkt = (pid, ph) \wedge pkt' = (pid, ph')$.*

Event-Driven Consistent Updates. An *event-driven consistent update* of a network is a triple consisting of initial configuration C_i, event e, and final configuration C_f. We denote such an update by $C_i \xrightarrow{e} C_f$. We now define a correctness condition on network traces.

Definition 2 (Event-driven consistent update)**.** *A network trace $ntr = lp_0 lp_1 \ldots$ is correct w.r.t. an event-driven consistent update $C_i \xrightarrow{e} C_f$ iff the following condition holds. Let lp_e be the first occurrence of e in ntr (i.e., $lp_e = eo(ntr, e, 1)$). For all IDs pid occurring in ntr, let $ntr{\downarrow}pid = lp_0^{pid} lp_1^{pid} \ldots$. For all pid occurring in ntr we require:*

- *if $\forall j : lp_j^{pid} \prec lp_e$, then $ntr{\downarrow}pid$ is in $\mathit{Traces}(C_i)$, (i.e., the packet is processed in C_i), and*
- *if $\forall j : lp_e \prec lp_j^{pid}$, then $ntr{\downarrow}pid$ is in $\mathit{Traces}(C_f)$ (i.e., the packet is processed in C_f), and*
- *otherwise, $ntr{\downarrow}pid$ is in $\mathit{Traces}(C_i) \cup \mathit{Traces}(C_f)$ (i.e., the packet is processed in one of C_i or C_f).*

To illustrate, consider Figure 1. We describe an update $C_i \xrightarrow{e} C_f$. In the initial configuration C_i, the host H_1 can send packets to H_2, but not vice-versa. In the final configuration C_f, traffic from H_2 to H_1 is allowed. Event e models the arrival to s_4 of a packet from H_1 (imagine s_4 is part of a distributed firewall). Assume that e occurs, and immediately afterwards H_2 wants to send a packet to s_1. Can s_2 drop the packet (as it would do in configuration C_i)? Event-driven consistent updates allow this, as otherwise we would require s_2 to react immediately to the event at s_4, which would be an example of action at a distance. Formally, the occurrence of e is not in a happens-before relation with the arrival of the new packet to s_2. On the other hand, if e.g. s_4 forwards some packets to s_1 and s_2 before the new packet from H_2 arrives, s_1 and s_2 would be required to change their configurations, and the packet would be allowed to reach H_1.

Network Event Structures. Event-driven consistent updates specify how the network should behave during a single update triggered by an event, but we also want to specify behavior when multiple events occur, and capture constraints between the events. For example, we might want to say that e_2 can only happen after e_1 has occurred, or that e_2 and e_3 cannot both occur in the same network trace.

To model such constraints, we turn to the *event structures* model introduced by Winskel [37]. Intuitively, an event structure endows a set of events $\mathcal{E}$ with (a) a *consistency predicate* (con) specifying which events are allowed to occur in the same sequence, and (b) an *enabling relation* ($\vdash$) specifying a (partial) order in which events can occur.

Definition 3 (Event structure)**.** *An event structure is a tuple $(\mathcal{E}, con, \vdash, g)$ where:*

- *$\mathcal{E}$ is a set of events,*
- *$con : \mathit{fin}(\mathcal{P}(\mathcal{E})) \to$ Boolean is a consistency predicate that satisfies $con(X) \wedge Y \subseteq X \implies con(Y)$,*
- *$\vdash : \mathcal{P}(\mathcal{E}) \times \mathcal{E} \to$ Boolean is an enabling relation that satisfies $(X \vdash e) \wedge X \subseteq Y \implies (Y \vdash e)$.*

Each event structure can be seen as defining a transition system whose states are the finite subsets of $\mathcal{E}$ that are consistent and reachable via the enabling relation. We refer to such a subset an as an *event-set* (called “configuration” in [37]).

Definition 4 (Event-set of an event structure)**.** *Given an event structure $M = (\mathcal{E}, con, \vdash)$, an* event-set *of M is any subset $X \subseteq \mathcal{E}$ which is: (a) consistent: $\forall Y \subseteq_{\mathit{fin}} X$, $con(Y)$ is true, and (b) reachable via the enabling relation: for each*

$e \in X$, there is a sequence $e_0, e_1, \cdots, e_n \in X$ where $e_n = e$ and $\{e_0, \cdots, e_{i-1}\} \vdash e_i$ for all $i \leq n$.

We want to be able to specify which network configuration should be active at each event-set of the event structure. Thus, we need the following extension of event structures:

Definition 5 (Network event structure (NES))**.** *A* network event structure *is a tuple $(\mathcal{E}, con, \vdash, g)$ where $(\mathcal{E}, con, \vdash)$ is an event structure, and $g : \mathcal{P}(\mathcal{E}) \to \mathcal{C}$ maps each event-set of the event structure to a network configuration.*

Correct Network Traces. Now we define what it means for a network trace to be correct w.r.t. a given network event structure M. Intuitively, if ntr is a network trace, we consider the unique sequence S of events that is both allowed by M, and matches the trace ntr.

To start, note that the first event of S is uniquely determined, if it exists. Let Cnd_0 be a set of events e such that $\emptyset \vdash e$ and $con(\{e\})$. Let e_0 be the event in Cnd_0 which has the first occurrence in ntr among the events in Cnd_0. The event e_0 is then the first event in the sequence S. (If Cnd_0 is empty, or no event in Cnd_0 has an occurrence in ntr, we return an empty sequence S). Let k_0 be the index of the first occurrence of e_0 in ntr. Similarly, after finding first $i-1$ events, the i-th event is uniquely determined, if it exists. For $1 \leq i$, let Cnd_i be the set of events e such that $\{e_0, e_1 \ldots e_{i-1}\} \vdash e$ and $con(\{e_0, e_1 \ldots e_{i-1}, e\})$. Let e_i be the event in Cnd_i which has the first occurrence in ntr (among the events in Cnd_i) which is a located packet $lp = ((pid, ph), sw)$ such that $ntr{\downarrow}pid$ is in $Traces(g(\{e_0, e_1 \ldots e_{i-1}\}))$. When no such event exists, we return the sequence $S = e_0 \ldots e_{i-1}$. Let k_i be the index of the corresponding occurrence of e_i in ntr. Let N be the length of the sequence of events e_i and corresponding indices k_i. We have thus obtained a sequence of events $S = e_0 e_1 \ldots e_N$ and their corresponding occurrences (for the event e_i, the relevant occurrence in ntr is k_i).

We now check whether the trace ntr satisfies the sequence S according to the event-driven consistent update condition extended to sequences of events. For all IDs pid occurring in ntr, let $ntr{\downarrow}pid = lp_0^{pid} lp_1^{pid} \ldots$. For all pid occurring in ntr, we require that $ntr{\downarrow}pid$ is in $\bigcup_{j=0}^{N} Traces(g(\{e_0, e_1 \ldots e_j\}))$ (i.e., each packet is processed entirely by one configuration, not by a mixture of configurations), and furthermore:

- if $\forall j : lp_j^{pid} \prec lp_{k_i}$, then $ntr{\downarrow}pid$ is an element of $\bigcup_{j=0}^{i-1} Traces(g(\{e_0, e_1 \ldots e_j\}))$, that is, the packet is processed by one of the configurations preceding e_i, and
- if $\forall j : lp_{k_i} \prec lp_j^{pid}$, then $ntr{\downarrow}pid$ is an element of $\bigcup_{j=i+1}^{N} Traces(g(\{e_0, e_1 \ldots e_i e_{i+1} \ldots e_j\}))$, that is, the packet is processed by a configuration that follows e_i.

Locality Restrictions for Incompatible Events. We now show how NESs can be used to impose reasonable locality restrictions on updates. A set of events E is called *inconsistent* if $con(E)$ does not hold. It is called *minimally-inconsistent* iff all of its proper subsets are consistent. An NES M is called *locally-determined* iff for all of its minimally-inconsistent sets E, we have that all events in E happen at the same switch (i.e., there exists a switch sw such that each $e_i \in E$ is of the form (eid_i, φ_i, sw)).

To illustrate the need for the locally-determined property, recall that two events are inconsistent if either of them can happen, but both cannot happen in the same execution. Consider the topology shown in Figure 1 and suppose the program requires that H_2 and H_4 can both receive packets from H_1, but only the first one to receive a packet is allowed to respond. There will be two events e_1 and e_2, with e_1 the arrival of a packet from H_1 at s_2, and e_2 the arrival of a packet from H_1 at s_4. The two events are always enabled, but the set $\{e_1, e_2\}$ is not consistent ($con(\{e_1, e_2\})$ does not hold). This models the fact that at most one of the events can take effect. These events are on different switches – making sure that at most one of the events takes effect would necessitate information to be propagated instantaneously "at a distance." In implementations, this would lead to synchronization and buffering of packets. This is where the locality restriction comes into play–it gives us a clean condition that ensure that the NES semantics is efficiently implementable.

On the other hand, consider the requirement that H_2 can send traffic to one of the two hosts (H_1, H_3) that sends it a packet first. The two events (a packet from H_1 arriving at s_2, and a packet from H_3 arriving at s_2) are still inconsistent, but inconsistency does not cause problems in this case, because both events happen at the same switch. The switch can properly determine which one was the first.

Strengthening Consistency. We now show that strengthening the consistency conditions imposed by NESs would lead to lower availability, as it would lead to the need for expensive synchronization (packet buffering, etc.). First, let us now try to remove the locally-determined condition, and obtain a strengthened consistency condition. The proof of the following theorem is an adaptation of the proof of the CAP theorem [6] as presented in [12]. The idea is that in asynchronous network communication, a switch might need to wait arbitrarily long to hear about an event.

Lemma 1. *It is impossible to faithfully implement an NES that does not satisfy the locally-determined condition while guaranteeing that every packet will be processed by each switch within an given a priori time bound.*

We now ask whether we can strengthen the event-driven consistent update definition. We define *strong update* as an update $C_1 \xrightarrow{e} C_2$ such that immediately after the event e occurred, the network processes all incoming packets in C_2. We obtain the following lemma by the same reasoning as the previous one.

Lemma 2. *It is impossible to implement strong updates and guarantee that every packet is processed by a switch within an a priory given time bound.*

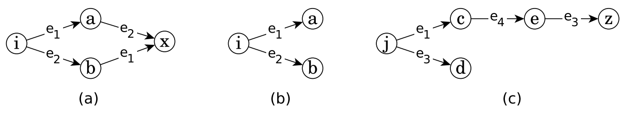


Figure 2: Event-driven transition systems.

3. Programming with Events

This section introduces an intuitive method for building NESs using transition systems where nodes correspond to configurations and edges correspond to events. We also present a network programming language based on NetKAT that provides a compact notation for specifying both transition systems and the configurations at each node.

3.1 Event-driven Transition Systems

Definition 6 (Event-driven Transition System)**.** *An* event-driven transition system (ETS) *is a graph* (V, D, v_0)*, where* V *is a set of vertices, where each vertex is labeled by a configuration* C*;* $D \subseteq V \times V$ *is a set of edges, where each edge is labeled by an event* e*; and* v_0 *is the initial vertex.*

Consider the ETSs shown in Figure 2 (a-b). In (a), the two events are intuitively compatible—they can happen in any order, so we obtain a correct execution if both happen in different parts of the network and different switches can have a different view of the order in which they happened. In (b), the two events are intuitively incompatible—only one of them can happen in any particular execution. Therefore even if they happen nearly simultaneously, only one of them should take an effect. To implement this, we need the locality restriction—we need to check whether the two events happen at the same switch. We thus need to distinguish between the two ETSs in Figure 2, parts (a) and (b) to determine where locality restrictions must be imposed in the conversion from an ETS to an NES.

From ETSs to NESs. To convert an ETS to an NES, we first form the event sets (as in Definition 4) and then construct the enabling relation and consistency predicate. We assume that the ETS is loop-free. Given an ETS T, consider the set $W_p(T)$ of paths in T from the initial node to any vertex. For each path $p \in W_p(T)$, let $S(p)$ be the set of events collected along the path. The set $F(T) = \{S(p) \mid p \in W_p(T)\}$ is our candidate collection of event sets. We now define conditions under which $F(T)$ gives rise to a network event structure. These conditions can be checked for T using straightforward graph algorithms, and any problematic vertices or edges in the ETS can be indicated to the programmer.

1. We require that each set S in $F(T)$ must correspond to exactly one network configuration. This holds if all paths in $W_p(T)$ corresponding to S end at states labeled with the same configuration.
2. We require that $F(T)$ is *finite-complete*, i.e. for any sets $S_1, S_2, \cdots, S_n$ where $S_i \in F(T)$, if there is a set $S' \in F(T)$ which contains every S_i (an upper bound for the sets S_i), then the set $S_{lub} = \cup_i S_i$ (the least upper bound for the S_i) is also in $F(T)$. For example, consider the ETS in Figure 2(c), which violates this condition since the event-sets $S_1 = \{e_1\}$ and $S_2 = \{e_3\}$ are both subsets of $\{e_1, e_4, e_3\}$, but there is no event-set of the form $S_1 \cup S_2 = \{e_1, e_3\}$.

In [37], such a collection $F(T)$ is called a *family of configurations*. Our condition #2 is condition (i) in Theorem 1.1.9 in [37] (conditions (ii)-(iii) are already satisfied by the ETS).

We build the con and $\vdash$ relations of an event structure from the family $F(T)$, using Theorem 1.1.12. of [37]. Predicate con can be defined by declaring all sets in $F(T)$ as consistent. For $\vdash$, we take the smallest relation satisfying the constraints: $\emptyset \vdash e \iff \{e\} \in con$ and $X' \vdash e \iff (X' \in con) \wedge ((X' \cup \{e\}) \in F \vee (\exists X \subseteq X' : X \vdash e))$.

After obtaining an NES, deciding where to check the locality restriction is easy: we check whether the NES is locally determined (see Section 2), and verify for each minimally-inconsistent set that the locality restriction holds.

Loops in ETSs. If there are loops in the ETS T, the previous definition needs to be slightly modified because we need to rename events encountered multiple times in the same execution. This gives rise to an NES where each event-set is finite, but the NES itself might be infinite (and thus can only be computed lazily). If we have the ability to store/communicate unbounded (but finite) event-sets in the network run-time, then no modifications are needed to handle infinite NESs in the implementation. Otherwise, we can handle these by computing the strongly-connected components (SCCs) of the ETS, enforcing the locality restriction on events in each (non-singleton) SCC, and requiring the implementation to attach timestamps on occurrences of events in those SCCs.

3.2 Stateful NetKAT

NetKAT [2] is a domain-specific language for specifying network behavior. It has a semantics based on Kleene Algebra with Tests (KAT), and a sound and complete equational theory that enables formal reasoning about programs. Operationally, a NetKAT program can be modeled as a function takes as input a single packet, and uses tests, field-assignments, sequencing, and union to produce a set of "histories" corresponding to packet traces.

Standard NetKAT does not support mutable *state*. Every packet is processed in isolation using the static function described by the program. Thus, we can use a standard NetKAT program for specifying individual network configurations, but not event-driven configuration changes. We describe a stateful variant of NetKAT which allows us to compactly specify a *collection* of network configurations, as well as the event-driven relationships between them. This variant preserves the existing equational theory of individual configurations (though it is not a KAT itself), but also allows packets to affect processing of future packets in the network via assignments to (and tests of) a global *state*.

The syntax of Stateful NetKAT is shown in Figure 4. A Stateful NetKAT program is a *command*, which can be

$$
\begin{array}{rcl}
\llparenthesis f ⊜ n \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & (\{\}, \{\varphi \wedge f ⊜ n\}) \\
\llparenthesis \mathbf{sw} ⊜ n \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & \llparenthesis \mathbf{true} \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis \mathbf{port} ⊜ n \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & \llparenthesis \mathbf{true} \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis state(m) ⊜ n \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & \begin{cases} \llparenthesis \mathbf{true} \rrparenthesis_{\vec{k}}\ \varphi & \text{if } \vec{k}(m) ⊜ n \\ \llparenthesis \mathbf{false} \rrparenthesis_{\vec{k}}\ \varphi & \text{otherwise} \end{cases} \\
\llparenthesis f \leftarrow n \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & (\{\}, \{(\exists f : \varphi) \wedge f{=}n\}) \\
\llparenthesis p + q \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & (\llparenthesis p \rrparenthesis_{\vec{k}}\ \varphi) \sqcup (\llparenthesis q \rrparenthesis_{\vec{k}}\ \varphi) \\
\llparenthesis p\ ;\ q \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & (\llparenthesis p \rrparenthesis_{\vec{k}} \blacksquare \llparenthesis q \rrparenthesis_{\vec{k}})\ \varphi \\
\llparenthesis p* \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & \bigsqcup_j F_p^j\ (\varphi, \vec{k})
\end{array}
\qquad
\begin{array}{rcl}
\llparenthesis a \wedge b \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & \llparenthesis a\ ;\ b \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis a \vee b \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & \llparenthesis a + b \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis \mathbf{true} \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & (\{\}, \{\varphi\}) \\
\llparenthesis \mathbf{false} \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & (\{\}, \{\}) \\
\llparenthesis \neg\mathbf{true} \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & \llparenthesis \mathbf{false} \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis \neg\mathbf{false} \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & \llparenthesis \mathbf{true} \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis \neg(v ⊜ n) \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & \llparenthesis v \not⊜ n \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis \neg\neg a \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & \llparenthesis a \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis \neg(a \wedge b) \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & \llparenthesis \neg a \vee \neg b \rrparenthesis_{\vec{k}}\ \varphi \\
\llparenthesis \neg(a \vee b) \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & \llparenthesis \neg a \wedge \neg b \rrparenthesis_{\vec{k}}\ \varphi
\end{array}
$$

$$
\begin{array}{rcl}
\llparenthesis (s_1{:}p_1) \rightarrowtriangle (s_2{:}p_2) \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & (\{\}, \{\varphi\}) \\
\llparenthesis (s_1{:}p_1) \rightarrowtriangle (s_2{:}p_2) \rightarrowtriangle \langle \mathbf{state}(m) \leftarrow n \rangle \rrparenthesis_{\vec{k}}\ \varphi & \triangleq & (\{(\vec{k}, (\varphi, s_2, p_2), \vec{k}[m \mapsto n])\}, \{\varphi\})
\end{array}
$$

Figure 3: Stateful NetKAT: Extracting event-edges from state $\vec{k}$.

$$
\begin{array}{rcll}
f & \in & \mathit{Field} & \textit{(header-field name)} \\
x & ::= & f \mid \mathbf{pt} & \textit{(scalar variable)} \\
a, b & ::= & \mathbf{true} \mid \mathbf{false} \mid x = n \mid \mathbf{sw}{=}n \mid \mathbf{state}(n) = n & \textit{(test)} \\
 & \mid & a \vee b \mid a \wedge b \mid \neg a & \\
p, q & ::= & a \mid x \leftarrow n \mid p + q \mid p\ ;\ q \mid p* \mid (n{:}n) \rightarrowtriangle (n{:}n) & \textit{(command)} \\
 & \mid & (n{:}n) \rightarrowtriangle (n{:}n) \rightarrowtriangle \langle \mathbf{state}(n) \leftarrow n \rangle &
\end{array}
$$

Figure 4: Stateful NetKAT: Syntax.

$$
\begin{array}{rcl}
[\![\mathbf{state}(m){=}n]\!]_{\vec{k}} & \triangleq & \begin{cases} [\![\mathbf{true}]\!]_{\vec{k}} & \text{if } \vec{k}(m){=}n \\ [\![\mathbf{false}]\!]_{\vec{k}} & \text{otherwise} \end{cases} \\
[\![(a{:}b) \rightarrowtriangle (c{:}d) \rightarrowtriangle \langle \mathbf{state}(m) \leftarrow n \rangle]\!]_{\vec{k}} & \triangleq & [\![(a{:}b) \rightarrowtriangle (c{:}d)]\!]_{\vec{k}}
\end{array}
$$

Figure 5: Stateful NetKAT: Extracting NetKAT Program for state $\vec{k}$.

- a *test*, which is a formula over packet header fields (there are special fields **sw** and **pt** which test the switch- and port-location of the packet respectively),
- a *field assignment* $x \leftarrow n$, which changes a packet's header-field value,
- a *union* of commands $p + q$, which unions together the packet-processing behavior of commands p and q,
- a command *sequence* $p\,;q$, which runs packet-processing program q on the result of p,
- an *iteration* $p*$, which is equivalent to $\mathbf{true} + p + (p\,;p) + (p\,;p\,;p) + \cdots$,
- or *conditional forwarding* $(n_1{:}m_1) \rightarrowtriangle (n_2{:}m_2)$, which forwards a packet from port m_1 at switch n_1 to port m_2 at switch n_2.

The key distinguishing feature of our Stateful NetKAT is a special global vector-valued variable called **state**, which allows the programmer to represent a collection of NetKAT programs. The function shown in Figure 5 gives the standard NetKAT program $[\![p]\!]_{\vec{k}}$ corresponding to each value $\vec{k}$ of the state vector (for conciseness, we only show the non-trivial cases). We can use the NetKAT compiler [32] to generate forwarding tables (i.e. configurations) corresponding to these, which we denote $C([\![p]\!]_{\vec{k}})$.

3.3 Converting Stateful NetKAT Programs to ETSs

It is clear how the $[\![\cdot]\!]_{\vec{k}}$ function can produce the vertices of an ETS. In Figure 3, we define another function $\llparenthesis \cdot \rrparenthesis_{\vec{k}}$, which produces the event-edges. This collects (using parameter φ) the conjunction of all guards seen up to a given program location, and records a corresponding event-edge when a state assignment command is encountered. In the figure, the $\blacksquare$ operator denotes (pointwise) Kleisli composition, i.e. $(f \blacksquare g) \triangleq \bigsqcup\{g\ y : y \in f\ x\}$, and function F is

$$
\begin{array}{rcl}
F_p^0\ (\varphi, \vec{k}) & \triangleq & (\{\}, \{\varphi\}) \\
F_p^{j+1}\ (\varphi, \vec{k}) & \triangleq & (\llparenthesis p \rrparenthesis_{\vec{k}} \blacksquare F_p^j)\ \varphi
\end{array}
$$

The symbol variable $⊜$ can be either equality "$=$" or inequality "$\neq$", and $\not⊜$ is the opposite symbol w.r.t. $⊜$. Given any conjunction φ and a header field f, the formula $(\exists f : \varphi)$ strips all predicates of the form $(f ⊜ n)$ from φ.

Using *fst* to denote obtaining the first element of a tuple, we can now produce the event-driven transition system for a Stateful NetKAT program p with the initial state $\vec{k}_0$:

$$
\begin{array}{l}
ETS(p) \triangleq \quad (V, D, v_0) \\
\quad \text{where } V \triangleq \bigcup_{\vec{k}} \{(\vec{k}, C([\![p]\!]_{\vec{k}}))\} \\
\quad \quad \text{and } D \triangleq \mathit{fst}\ \big(\bigsqcup_{\vec{k}} \llparenthesis p \rrparenthesis_{\vec{k}}\ \mathbf{true}\big) \\
\quad \quad \text{and } v_0 \triangleq (\vec{k}_0, C([\![p]\!]_{\vec{k}_0}))
\end{array}
$$

4. Implementing Event-Driven Programs

Next we show how to implement NESs in a real SDN and prove that it is correct—i.e. all traces followed by actual packets in the network should be correct with respect to the Section 2 definition. To a first approximation, our implementation can be understood as follows. We assume that the switches in the network provide mutable state that can be read and written as packets are processed. Given an NES, we assign a tag to each event-set and compile a collection of configurations whose rules are predicated on the appropriate tags. We then add logic which (i) updates the mutable state to record local events, (ii) stamps incoming packets with the tag for the current event-set upon ingress, and (iii) reads the tags carried by packets and updates the event-set at subsequent switches.

4.1 Building Blocks

Static Configurations. The NES contains a set of network configurations that need to be installed as flow tables on

$$\begin{array}{llll}
\textit{Switch ID} & n & \in & \mathbb{N} \\
\textit{Port ID} & m & \in & \mathbb{N} \\
\textit{Host ID} & h & \in & \mathbb{N} \\
\textit{Counter} & z & \in & \mathbb{N} \\
\textit{Identifier} & i & \in & \mathbb{N}
\end{array}
\quad
\begin{array}{llll}
\textit{Packet} & pkt & ::= & (pid, \{f_1; \cdots ; f_k\}) \\
\textit{Location} & l & ::= & n : m \\
\textit{Queue Map} & qm & ::= & n \mapsto pkts \\
\textit{Link} & lk & ::= & (l, l) \\
\textit{Links} & L & ::= & \{lk, \cdots\} \\
\textit{Event} & e & ::= & (n, \varphi)
\end{array}
\quad
\begin{array}{llll}
\textit{Configuration} & C & ::= & (pkt, n{:}m) \mapsto \{(pkt, m)\} \\
\textit{Event-set} & E & ::= & \{e, \cdots\} \\
\textit{Enabling Relation} & \vdash & ::= & E \mapsto E \\
\textit{Configuration Map} & g & ::= & E \mapsto C \\
\textit{Switch} & sw & ::= & (n, qm, E, qm) \\
\textit{Queue, Controller} & Q, R & ::= & E \\
\textit{Switches} & S & ::= & \{sw, \cdots\}
\end{array}$$

$$\frac{(h, n{:}m) \in L \quad S = S' \cup \{(n, qm[m \mapsto pkts], E, qm_2)\}}{(Q, R, S) \to (Q, R, S' \cup \{(n, qm[m \mapsto pkts@[pkt_{g(E)}]], E, qm_2)\})}\ \textsc{In}
\qquad
\frac{(n{:}m, h) \in L \quad S = S' \cup \{(n, q_1, E, qm[m \mapsto pkt{::}pkts])\}}{(Q, R, S) \to (Q, R, S' \cup \{(n, q_1, E, qm[m \mapsto pkts])\})}\ \textsc{Out}$$

$$\frac{C(pkt_C, n{:}m){=}\{(m_1, pkt_1), \cdots\} \quad E'{=}\{e{:}((E{\cup}pkt.d){\vdash}e){\wedge}(pkt, n{:}m){\models}e\} \quad S{=}S'{\cup}\{(n, qm[m{\mapsto}pkt_C{::}pkts], E, qm_2[m_1{\mapsto}pkts_1, \cdots])\}}{(Q, R, S) \to (Q{\cup}E', R, S'{\cup}\{(n, qm[m{\mapsto}pkts], E{\cup}E'{\cup}pkt.d, qm_2[m_1{\mapsto}pkts_1@[pkt_1[d{\mapsto}pkt_1.d{\cup}E{\cup}E']], \cdots])\})}\ \textsc{Sw}$$

$$\frac{(n_1{:}m_1, n_2{:}m_2) \in L \quad S = S' \cup \{(n_1, q_1, E_1, qm_2[m_1 \mapsto pkt{::}pkts]), (n_2, qm_3[m_2 \mapsto pkts'], E_2, qm_4)\}}{(Q, R, S) \to (Q, R, S' \cup \{(n_1, q_1, E_1, qm_2[m_1 \mapsto pkts]), (n_2, qm_3[m_2 \mapsto pkts'@[pkt]], E_2, qm_4)\})}\ \textsc{Link}$$

$$\frac{Q = Q' \cup \{o\}}{(Q, R, S) \to (Q', R \cup \{o\}), S)}\ \textsc{CtrlRecv}
\qquad
\frac{R = R' \cup \{o\} \quad S = S' \cup \{(n, qm, E, qm_2)\}}{(Q, R, S) \to (Q, R, S' \cup \{(n, qm, E \cup \{o\}, qm_2)\})}\ \textsc{CtrlSend}$$

Figure 6: Implemented Program Semantics

switches. In addition, we may need to transition to a new configuration in response to a local event. We do this *proactively*, installing all of the needed rules on switches in advance, with each rule guarded by its configuration's ID. This has a disadvantage of being less efficient in terms of rule-space usage, but an advantage of allowing very quick configuration changes. In Section 5.3, we discuss an approach to addressing the space-usage issue.

Stateful Switches. New switch-programming languages such as P4 [5] and OpenState [4] are being actively developed, adding support for advanced functionality such as customizable parsing, and arbitrary stateful registers. We assume that the switches support reading and writing stateful register(s) while processing each packet. This allows each switch to maintain a local view of the global state. Specifically, the register records the set of events the device knows have occurred. At any time, the device can receive a packet (from the controller or another device) informing it of new event occurrences, which are then added to the local register.

Packet Processing. Each packet entering the network is admitted from a host to a port at the edge. The configuration number j corresponding to the device's view of the global state is assigned to the packet's *version number* field. The packet will processed only by j-guarded rules throughout its lifetime. Packets also carry a *digest* encoding the set of events seen so far (i.e. the packet's view of the global state). If the packet passes through an edge device which has seen additional events, the packet's digest is updated accordingly. Similarly, if the packet's digest contains events not yet seen by the device, the latter adds them to its view of the state. When a packet triggers an event, that event is immediately added to the packet's digest, as well as to the state of the device where the event was detected. The controller is then notified about the event. Optionally, the controller can periodically broadcast its view of the global state to all switches, in order to speed up dissemination of the state.

4.2 Operational Model

We formalize the above via an operational semantics for the global behavior of the network as it executes a NES.

Each state in Figure 6 has the form (Q, R, S), with a controller queue Q, a controller R, and set of switches S. Both the controller queue and controller are a set of events. Each switch $s \in S$ is a tuple (n, q_{in}, E, qm_{out}), where n is the switch ID, qm_{in}, qm_{out} are the input/output queue maps (mapping port IDs to packet queues). The event-set E represents this switch's view of what events have occurred.

- IN/OUT: move a packet between a host and edge port;
- SW: process a packet by first adding new events from the packet's digest to the local state, then checking if the packet's arrival matches an event e enabled by the NES and updating the state and packet digest if so, and finally updating the digest with other local events;
- LINK: move packets across a link;
- CTRLRECV: bring an event occurrence from the controller queue into the controller;
- CTRLSEND: update local state of the switches.

4.3 Correctness of the Implementation

We now prove the correctness of our implementation. Formally we show that the traces allowed by the operational semantics are correct traces, as defined in Section 2.

Lemma 3 (Global Consistency)**.** *Given network event structure N, if the locality restriction is satisfied, then given an execution of the implementation $t = (Q_1, R_1, S_1)(Q_2, R_2, S_2) \cdots (Q_m, R_m, S_m)$, the event-set $X = Q_i \cup R_i$ is consistent for all $1 \leq i \leq m$.*

Proof Sketch. We first show (by contradiction) that if there is no nonderminism in the N (i.e. a case such as Figure 2(a)), then $X \cup \{e\}$ is consistent for all e and any $X \in N$.

Otherwise, we assume that the implementation adds an e_0 to some consistent event-set X, producing an inconsistent set. We look at the minimally-inconsistent set $Y \subseteq (X \cup$

$\{e_0\}$), and notice that the locality restriction requires all $e \in Y$ be detected at the same switch, resulting in $|Y| \leq 1$, and generating a contradiction, since Y is then consistent. □

Traces of the Implementation. Note that we can readily follow the path of a single packet throughout an implementation trace (at most one packet moves at each step of Figure 6). Given an implementation trace t, we overload the notation $Traces(t)$ to denote the set of all such single-packet traces in t. We now present the main result of this section—*executions of the implementation allow only correct traces*.

Theorem 1 (Implementation Correctness). *For NES N, and execution $t = (Q_1, R_1, S_1)(Q_2, R_2, S_2) \cdots (Q_i, R_i, S_i)$ of the implementation, each packet trace in $Traces(t)$ is correct with respect to N.*

Proof Sketch. The proof is by induction over the length of the execution t. In the induction step, we show that (1) the SW rule can only produce consistent event-sets (this follows directly from Lemma 3), and (2) when the IN rule tags a packet pkt based on the local event-set E, that E consists of exactly the events that happened before pkt arrived (as ordered by the happens-before relation). □

5. Implementation and Evaluation

We built a full-featured prototype implementation:

- We implemented the compiler described in Section 3. This tool accepts a Stateful NetKAT program, and produces the corresponding NES, with a standard NetKAT program representing the configuration at each node. We interface with Frenetic's NetKAT compiler to produce flow-table rules for each of these NetKAT programs.
- We have modified the OpenFlow 1.0 reference implementation to support the custom switch/controller needed to realize the runtime described in Section 4.
- We have built tools to automatically generate custom Mininet [22] scripts to bring up the programmer-specified network topology, using switches/controller running the compiled NES. We can then realistically simulate the whole system using real network traffic.

Research questions. To evaluate our approach, we wanted to obtain answers to the following questions:

1. How useful is our approach? Does it allow programmers to easily write real-world network programs, and get the behavior they want?
2. What is the performance of the tools (compiler, etc.)?
3. How much does our correctness guarantee help? For instance, how do the running network programs compare with "best-effort" event-driven strategies?
4. How efficient are the implementations generated by our approach? For instance, what about message overhead? State-change convergence time? Number of rules used?

We address these through case studies on real-world programming examples, and quantitative performance measurements on simple automatically-generated programs. For the experiments, we assume that the programmer has first confirmed that the program satisfies the conditions allowing proper compilation to an NES, and we assume the ETS has no loops. Our experimental platform was an Ubuntu machine with 20GB RAM and a quad-core Intel i5-4570 (3.2 GHz).

5.1 Case Studies

Stateful Firewall. The example in Figures 7-11(a) is a simplified stateful firewall. It always allows "outgoing" traffic (from H1 to H4), but only allows "incoming" traffic (from H4 to H1) after the outside network has been contacted (outgoing forwarded to H4).

Program p corresponds to configurations $C_{[0]} = [\![p]\!]_{[0]}$ and $C_{[1]} = [\![p]\!]_{[1]}$. In the former, only outgoing traffic is allowed, and in the latter, both outgoing and incoming are allowed. The ETS has the form $\{\langle[0]\rangle \xrightarrow{(dst=H4,\, 4:1)} \langle[1]\rangle\}$. The NES has the form $\{E_0{=}\emptyset \rightarrow E_1{=}\{(dst{=}H4,\ 4{:}1)\}\}$, where the g is given by $g(E_0) = C_{[0]}, g(E_1) = C_{[1]}$.

The Stateful Firewall example took 0.013s to compile, and produced a total of 18 flow-table rules. In Figure 8(a), we show that the running firewall has the expected behavior. We first try to ping H1 from H4 (the "H4-H1"/red points), which fails. Then we ping H4 from H1 (the "H1-H4"/orange points), which succeeds. Again we try H4-H1, and now this succeeds, since the event-triggered state change occurred.

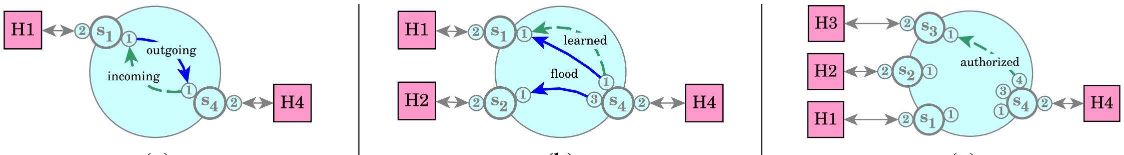
We compare this correct behavior with that of a "best-effort" update strategy, in which events are sent to the controller, which pushes updates to the switches at its convenience. The results are in Figure 8(b), showing that some of the H1-H4 pings get dropped (i.e. H1 doesn't hear back from H4), meaning the state change did not behave as if it was caused immediately upon arrival of a packet at S4.

Learning Switch. The example in Figures 7-11(b) is a simple learning switch. Traffic from H4 to H1 is flooded (sent to both H1 and H2), until H4 receives a packet from H1, at which point it "learns" the address of H1, and future traffic from H4 to H1 is sent only to H1.

This program p corresponds to two configurations $C_{[0]} = [\![p]\!]_{[0]}$ and $C_{[1]} = [\![p]\!]_{[1]}$. In the former, flooding occurs from H4, and in the latter, packets from H4 are forwarded directly to H1. The ETS has the form $\{\langle[0]\rangle \xrightarrow{(dst=H4,\, 4:1)} \langle[1]\rangle\}$. The NES has the form $\{E_0{=}\emptyset \rightarrow E_1{=}\{(dst{=}H4,\ 4{:}1)\}\}$, where the g is given by $g(E_0) = C_{[0]}, g(E_1) = C_{[1]}$.

This only allows learning a single host (H1), but we could easily add learning for H2 by using a different index in the vector-valued *state* field: we could replace **state** in Figure 11(b) with **state**(1), and union the program (using the NetKAT $+$ operator) with another instance of Figure 11(b) which learns for H2 and uses **state**(2).

The Learning Switch example took 0.015s to compile, and produced a total of 43 flow-table rules. We again compare the behavior of our correct implementation with that of an implementation which uses a best-effort update strategy. In Figure 9, we ping H1 from H4. The expected behavior is



(a) (b) (c)

Figure 7: Topologies: (a) firewall, (b) learning switch, (c) port knocking.

(a)

```
pt=2 ∧ ip_dst=H4; pt←1; (state=[0];
      (1:1)→(4:1)→⟨state←[1]⟩ + state≠[0];
      (1:1)→(4:1)); pt←2
+ pt=2 ∧ ip_dst=H1; state=[1]; pt←1;
      (4:1)→(1:1); pt←2
```

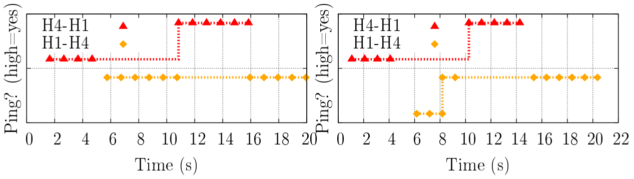
(b)

```
pt=2 ∧ ip_dst=H1; (pt←1; (4:1)→(1:1) +
      state=[0]; pt←3; (4:3)→(2:1)); pt←2
+ pt=2 ∧ ip_dst=H4; pt←1; (1:1)→(4:1)→⟨
      state←[1]⟩; pt←2
+ pt=2; pt←1; (2:1)→(4:3); pt←2
```

(c)

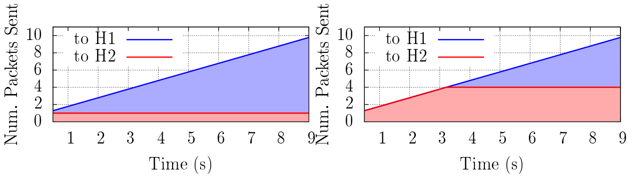
```
state=[0] ∧ pt=2 ∧ ip_dst=H1; pt←1;
      (4:1)→(1:1)→⟨state←[1]⟩; pt←2
+ state=[1] ∧ pt=2 ∧ ip_dst=H2; pt←3;
      (4:3)→(2:1)→⟨state←[2]⟩; pt←2
+ state=[2] ∧ pt=2 ∧ ip_dst=H3; pt←4;
      (4:4)→(3:1); pt←2
+ pt=2; pt←1; ((1:1)→(4:1) + (2:1)→(4:3) +
      (3:1)→(4:4)); pt←2
```

Figure 11: Programs: (a) firewall, (b) learning switch, (c) port knocking.



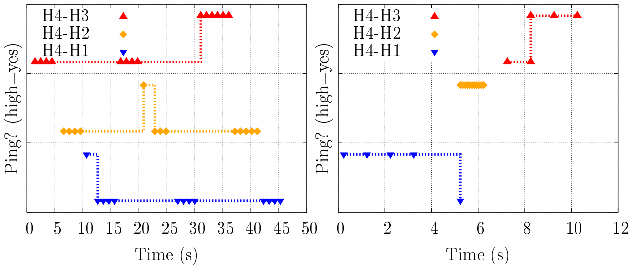
(a) (b)

Figure 8: Stateful Firewall—Correct (left) vs. Incorrect (right)



(a) (b)

Figure 9: Learning Switch: Correct (left) vs. Incorrect (right)



(a) (b)

Figure 10: Port Knocking: Correct (left) vs. Incorrect (right)

shown on the left, where the first packet is flooded to both H1 and H2, but then H4 hears a reply from H1, causing the state change (i.e. learning the address of H1), and all subsequent packets are sent only to H1. On the right, however, if the state change is delayed, multiple packets are sent to H2, even after H4 has seen a reply from H1.

Port Knocking. In this example, shown in Figures 7-11(c), the untrusted host H4 wishes to contact H3, but can only do so after contacting H1 and then H2, in that order.

This program p corresponds to three configurations: $C_{[0]} = [\![p]\!]_{[0]}$ in which only H4-H1 traffic is enabled, $C_{[1]} = [\![p]\!]_{[1]}$ in which only H4-H2 traffic is enabled, and $C_{[2]} = [\![p]\!]_{[2]}$ which finally allows H4 to communicate with H3. The ETS has the form $\{\langle[0]\rangle \xrightarrow{(dst=H1,\ 1:1)} \langle[1]\rangle \xrightarrow{(dst=H2,\ 2:1)} \langle[2]\rangle\}$. The NES has the form $\{E_0{=}\emptyset \rightarrow E_1{=}\{(dst{=}H1,\ 1{:}1)\} \rightarrow E_2{=}\{(dst{=}H1,\ 1{:}1),(dst{=}H2,\ 2{:}1)\}\}$, where the g function is given by $g(E_0) = C_{[0]}, g(E_1) = C_{[1]}, g(E_2) = C_{[2]}$.

The Port Knocking example took 0.017s to compile, and produced a total of 72 flow-table rules. In Figure 10 we demonstrate the correct behavior of the program, by first trying (and failing) to ping H3 and H2 from H4, then successfully pinging H1, again failing to ping H3 (and H1), and finally succeeding in pinging H3. The incorrect (best-effort) implementation on the left allows an incorrect behavior where we can successfully ping H1 and then H2, but then fail to ping H3 (at least temporarily).

5.2 Quantitative Results

In this experiment, we automatically generated some event-driven programs which specify that two hosts H1 and H2 are connected to opposite sides of a ring of switches. Initially, traffic is forwarded clockwise, but when one of the switches detects a (packet) event, the configuration changes to forward counterclockwise. We increased the "diameter" of the ring (distance from H1 to H2) up to 8, as shown in Figure 12, and performed two experiments:

1. We used `iperf` to measure H1-H2 TCP/UDP bandwidth, and compared the performance of our running event-driven program, versus that of the initial (static) configuration of the program running on un-modified OpenFlow 1.0 reference switches/controller. The left side of the figure shows that our performance (solid line) is very close to performance of a system which does not to packet tagging, event detection, etc. (dashed line)—we see around 6% performance degradation on average.
2. We measured maximum and average time needed for a switch to learn about the event. The "Max." and "Avg." bars in the right of the figure are these numbers when the controller does not assist in disseminating the events (i.e. only the packet digest is used), and the other columns are the maximum and average when the controller does so.

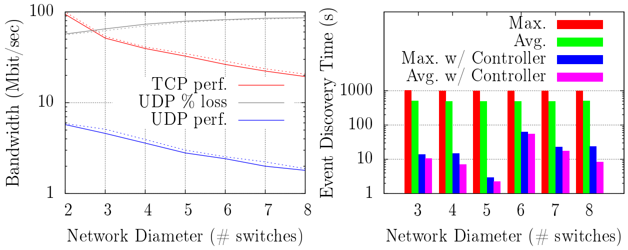


Figure 12: Circular Example (solid line is ours, dotted line is reference implementation)

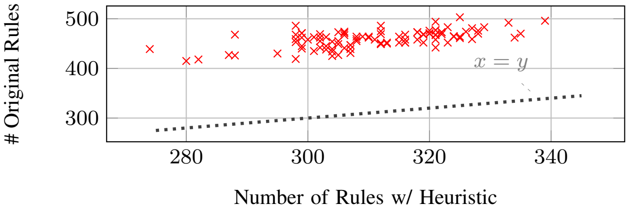


Figure 13: Polynomial heuristic to reduce the number of inserted rules

5.3 Optimizations

We provide an effective approach for reducing rule space usage: for example, if the same rule r is to be used in two different configurations with state IDs 2 (binary 10) and 3 (binary 11), we can simply wildcard the lowest bit $(1*)$ for use as a guard, instead of keeping r in both states, with the "10" and "11" guards. When IDs are being assigned to different configurations, it is non-obvious which assignment gives maximal sharing of this form. We implemented a polynomial heuristic to provide (re-)assignments that give good sharing, as indicated by the Figure 13 (64 randomly-generate configurations w/ 20 rules) results. On average, rule savings was about 32% of the original number of rules. We also ran this on the previously-discussed Firewall, Learning Switch, and Port Knocking examples, and got rule reductions of 18 $\rightarrow$ 16, 43 $\rightarrow$ 27, and 72 $\rightarrow$ 46 respectively.

6. Related Work

Network updates. We have already briefly mentioned an early approach known as consistent updates [31]. This work was followed by update techniques that respect other correctness properties [24] [16] [39] [25]. These approaches for expressing/verifying correctness of network updates work in terms of *individual* packets.

In event-driven network programs, it is necessary to check properties which talk about interactions between *multiple* packets. There are several current directions which seek to do network updates in the context of multi-packet properties, e.g. [11] and [23]. There are also approaches for synthesizing SDN controller program from multi-packet examples [38], and from specifications in first-order logic [30].

Network programs can often be constructed using high-level languages. The Frenetic project [9] [26] [10] allows higher-level specification of network policies. Other related projects such as Merlin [33] and NetKAT [32] [3] provide high-level languages and tools to compile such programs to network configurations. Works such as Nettle, [34], Procera [35], Maple [36], and FlowLog [29] seek to address the dynamic aspect of network programming.

None of these systems and languages provide both (1) event-based constructs, and (2) strong semantic guarantees about consistency during updates, while our framework enables both. Our approach is unique in that we enable correct-by-construction event-based behavior, providing a dynamic correctness property that is strong enough for easy reasoning, yet flexible enough to enable efficient implementations.

Routing. The trade-off between consistency and availability is of interest in routing outside of the SDN context as well. In [17], a solution called consensus routing, based on a notion of causality between triggers (related to our events). However, the solution is different in many aspects: for instance it allows a transient phase without safety guarantees.

Implementing High-level Network Functionality. Some recent work has proposed putting even more powerful features into the network itself, such as fabrics [8], intents [1], and other virtualization functionality [21]. Pyretic [27] and projects built on top of it such as PyResonance [19], SDX [13], and Kinetic [20] provide high-level operations on which network programs can be built. These projects do not guarantee consistency during updates, and thus could be profitably combined with our approach.

7. Discussion and Future Work

This research opens up several directions for future work that can address limitations of our current system. First, we assume that the set of (potential) hosts is known in advance, and use this information to generate the corresponding flow tables for each switch. This may not be the right choice in settings where hosts join and leave. The approach can be extended to represent hosts *symbolically*. Second, we currently store all configurations at switches, so they are immediately available during updates. Our optimizations allow this to be done in a space-efficient way, but there may be situations when it would be better for the controller to reactively push new configurations to switches. This is an interesting problem due to interleavings of events and controller commands. Third, we leave issues such as formal reasoning and automated verification for Stateful NetKAT for future work.

8. Conclusion

This paper presents a full framework for correct event-driven programming. Our approach provides a way of rigorously defining correct event-driven behavior without the need for specifying logical formulas. We detail a programming language and compiler which allow the user to write high-level network programs and produce correct and efficient SDN implementations, and we demonstrate the benefits of our approach using real-world examples. This paper considers the challenging problem of distributing an event-based network program, and solves it in a principled way.

References

[1] ONOS Intent Framework. 2014. URL `http://onos.wpengine.com/wp-content/uploads/2014/11/ONOS-Intent-Framework.pdf`.

[2] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic Foundations for Networks. *POPL*, 2014.

[3] R. Beckett, M. Greenberg, and D. Walker. Temporal NetKAT. *PLVNET*, 2015.

[4] G. Bianchi, M. Bonola, A. Capone, and C. Cascone. OpenState: Programming Platform-independent Stateful Openflow Applications Inside the Switch. *ACM SIGCOMM CCR*, 2014.

[5] P. Bosshart, D. Daly, G. Gibb, M. Izzard, N. McKeown, J. Rexford, C. Schlesinger, D. Talayco, A. Vahdat, G. Varghese, et al. P4: Programming Protocol-independent Packet Processors. *ACM SIGCOMM CCR*, 2014.

[6] E. Brewer. Towards robust distributed systems (abstract). *PODC*, page 7, 2000.

[7] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking Control of the Enterprise. *SIGCOMM*, 2007.

[8] M. Casado, T. Koponen, S. Shenker, and A. Tootoonchian. Fabric: A Retrospective on Evolving SDN. *HotSDN*, 2012.

[9] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. *ICFP*, 2011.

[10] N. Foster, A. Guha, M. Reitblatt, A. Story, M. J. Freedman, N. P. Katta, C. Monsanto, J. Reich, J. Rexford, C. Schlesinger, et al. Languages for Software-Defined Networks. *Communications Magazine, IEEE*, 51(2):128–134, 2013.

[11] S. Ghorbani and B. Godfrey. Towards Correct Network Virtualization. *HotSDN*, 2014.

[12] S. Gilbert and N. Lynch. Perspectives on the CAP Theorem. *IEEE Computer*, 45(2):30–36, 2012.

[13] A. Gupta, L. Vanbever, M. Shahbaz, S. P. Donovan, B. Schlinker, N. Feamster, J. Rexford, S. Shenker, R. Clark, and E. Katz-Bassett. SDX: A Software Defined Internet Exchange. *SIGCOMM*, 2014.

[14] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving High Utilization with Software-driven WAN. *SIGCOMM*, 2013.

[15] S. Jain et al. B4: Experience with a Globally-Deployed Software Defined WAN. *SIGCOMM*, 2013.

[16] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic Scheduling of Network Updates. *SIGCOMM*, 2014.

[17] J. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. Consensus Routing: The Internet as a Distributed System. *NSDI*, 2008.

[18] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the One Big Switch Abstraction in Software-Defined Networks. *CoNEXT*, 2013.

[19] H. Kim, A. Gupta, M. Shahbaz, J. Reich, N. Feamster, and R. Clark. Simpler Network Configuration with State-Based Network Policies. Technical report, Georgia Institute of Technology, 2013.

[20] H. Kim, J. Reich, A. Gupta, M. Shahbaz, N. Feamster, and R. Clark. Kinetic: Verifiable Dynamic Network Control. *NSDI*, 2015.

[21] T. Koponen, K. Amidon, P. Balland, M. Casado, A. Chanda, B. Fulton, I. Ganichev, J. Gross, N. Gude, P. Ingram, et al. Network Virtualization in Multi-tenant Datacenters. *NSDI*, 2014.

[22] B. Lantz, B. Heller, and N. McKeown. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. *HotNets*, 2010.

[23] W. Liu, R. B. Bobba, S. Mohan, and R. H. Campbell. Inter-Flow Consistency: Novel SDN Update Abstraction for Supporting Inter-Flow Constraints. *NDSS*, 2015.

[24] A. Ludwig, M. Rost, D. Foucard, and S. Schmid. Good Network Updates for Bad Packets: Waypoint Enforcement Beyond Destination-based Routing Policies. *HotNets*, 2014.

[25] J. McClurg, H. Hojjat, P. Cerny, and N. Foster. Efficient Synthesis of Network Updates. *PLDI*, 2015.

[26] C. Monsanto, N. Foster, R. Harrison, and D. Walker. A Compiler and Run-time System for Network Programming Languages. *POPL*, 2012.

[27] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing Software Defined Networks. *NSDI*, 2013.

[28] M. Moshref, M. Yu, A. B. Sharma, and R. Govindan. Scalable Rule Management for Data Centers. *NSDI*, 2013.

[29] T. Nelson, A. D. Ferguson, M. Scheer, and S. Krishnamurthi. Tierless Programming and Reasoning for Software-Defined Networks. *NSDI*, 2014.

[30] O. Padon, N. Immerman, A. Karbyshev, O. Lahav, M. Sagiv, and S. Shoham. Decentralizing SDN Policies. *POPL*, 2015.

[31] M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for Network Update. *SIGCOMM*, 2012.

[32] S. Smolka, S. Eliopoulos, N. Foster, and A. Guha. A Fast Compiler for NetKAT. *ICFP*, 2015.

[33] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A Language for Provisioning Network Resources. *CoNEXT*, 2014.

[34] A. Voellmy and P. Hudak. Nettle: Taking the Sting Out of Programming Network Routers. *PADL*, 2011.

[35] A. Voellmy, H. Kim, and N. Feamster. Procera: A Language for High-level Reactive Network Control. *HotSDN*, 2012.

[36] A. Voellmy, J. Wang, Y. R. Yang, B. Ford, and P. Hudak. Maple: Simplifying SDN Programming Using Algorithmic Policies. *SIGCOMM*, 2013.

[37] G. Winskel. *Event Structures*. Springer, 1987.

[38] Y. Yuan, R. Alur, and B. T. Loo. NetEgg: Programming Network Policies by Examples. *HotNets*, 2014.

[39] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey. Enforcing Generalized Consistency Properties in Software-Defined Networks. *NSDI*, 2015.