# Kuai: A Model Checker for Software-defined Networks

Rupak Majumdar
MPI-SWS
Germany

Sai Deep Tetali
UC Los Angeles
USA

Zilong Wang
MPI-SWS
Germany

***Abstract*—In software-defined networking (SDN), a software controller manages a distributed collection of switches by installing and uninstalling packet-forwarding rules in the switches. SDNs allow flexible implementations for expressive and sophisticated network management policies.**

**We consider the problem of verifying that an SDN satisfies a given safety property. We describe Kuai, a distributed enumerative model checker for SDNs. Kuai takes as input a controller implementation written in Murphi, a description of the network topology (switches and connections), and a safety property, and performs a distributed enumerative reachability analysis on a cluster of machines. Kuai uses a set of partial order reduction techniques specific to the SDN domain that help reduce the state space dramatically. In addition, Kuai performs an automatic abstraction to handle unboundedly many packets traversing the network at a given time and unboundedly many control messages between the controller and the switches.**

**We demonstrate the scalability and coverage of Kuai on standard SDN benchmarks. We show that our set of partial order reduction techniques significantly reduces the state spaces of these benchmarks by many orders of magnitude. In addition, Kuai exploits large-scale distribution to quickly search the reduced state space.**

## I. Introduction

Software-defined networking (SDN) is a novel networking architecture in which a centralized software controller dynamically updates the packet processing policies in network switches based on observing the flow of packets in the network [9], [5]. SDNs have been used to implement sophisticated packet processing policies in networks, and there is increasing industrial adoption [12], [9].

We consider the problem of verifying that an SDN satisfies a network-wide safety property. Since the controller code in an SDN can dynamically change how packets flow in the network, a bug in the controller code can lead to hard-to-analyze network errors at run time. We describe the design of Kuai, a distributed enumerative model checker for SDNs. The input to Kuai is a model of an SDN consisting of two parts. The first part is the controller, written in a simplified guarded-command language similar to Murphi. The second part is the description of a network, consisting of a fixed finite set of switches, a fixed set of client nodes, and the topology of the network (i.e., the connections between the ports of the clients and the switches). Given a safety property of the network, Kuai explores the state space of the SDN to check if the property holds on all executions.

Figure 1 shows a simple SDN. It consists of two switches $sw_1$ and $sw_2$ connected to two clients $c_1$ and $c_2$. Each client has a port and each switch has two ports to send and receive packets, and the figure shows how the ports are connected to each other. Each connection between ports represents a bi-directional communication channel that may reorder packets.

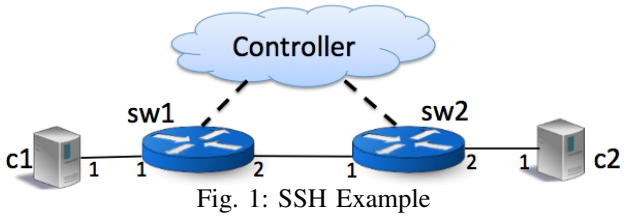


Fig. 1: SSH Example

```
def pktIn(pkt)
  (sw,pt) = pkt.loc
  if pkt.prot = SSH:
    drop(pkt)
  else:
    dest = 2 if pt = 1 else 1
    fwd(pkt, [|dest|], sw)
  rule r1 = (5,{prot=SSH},[||])
  rule r2 = (1,{port=1},[|2|])
  rule r3 = (1,{port=2},[|1|])
  message cm1 = add(r1)
  message cm2 = add(r2)
  message cm3 = add(r3)
  for sw in [sw1, sw2]:
    send_message(cm1, sw)
    send_message(cm2, sw)
    send_message(cm3, sw)
```

Listing 1: Controller for SSH

Moreover, the switches are connected to a controller through dedicated links. Packets are routed in the network using *flow tables* in switches. A flow table is a collection of prioritized forwarding *rules*. A rule consists of a priority, a pattern on packet headers, and a list of ports. A switch processes an incoming packet based on its flow table. It looks at the highest priority rule whose pattern matches the packet and forwards the packet to the list of ports specified in the rule, and drops the packet if the list of ports in the rule is empty. In case no rule matches a packet, the switch forwards the packet to the controller using a request queue and waits for a reply from the controller on a forward queue. The controller replies with a list of ports to which the packet should be forwarded, and optionally sends *control messages* to the control queue of one or more switches to update their flow tables. A control message can add or delete a rule in a switch.

By specifying the rules to be added or deleted, a controller can dynamically control the behaviors of all switches in an SDN network. For example, suppose we want to implement the policy that all SSH packets are dropped. The controller can update the switches with a rule that states that no SSH packets are forwarded, and another that states all non-SSH packets are forwarded. List 1 shows a possible controller that implements this policy. Essentially, the controller drops SSH packets, and adds three rules on the switches: $r1$ to drop SSH packets, $r2$ to forward packets from port 1 to port 2, and $r3$ to forward packets from port 2 to port 1. Since dropping

ISBN: 978-0-9835678-4-4.  

SSH packets (rule $r1$) has higher priority, it will match SSH packets, and rules $r2$ and $r3$ will only match (and forward) non-SSH packets. The controller has a subtle bug. It turns out that a switch can implement rules in arbitrary order. Thus, the switches may end up adding rules $r2$ and $r3$ before adding $r1$, thus violating the policy. Our model checker confirms the bug A possible fix in this case is to implement a *barrier* after line 15, to ensure that rule $r1$ is added before the other rules. Our model checker confirms the policy holds in the fixed version.

The verification of SDNs is challenging due to several reasons. First, even when the topology is fixed with a finite set of clients and switches, the state space is still unbounded, as clients may generate unboundedly many packets and these packets could be simultaneously progressing through the network. For example, client $c_1$ may send a packet to $sw_1$ at any point, and an unbounded number of packets can be in the network before $sw_1$ processes them. Similarly, there may be an unbounded number of control messages (i.e., messages sent from the controller to a switch) between the controller and the switches. While there may be a physical limit on the number of packets and control messages imposed by packet buffers in the switches, the sizes of these buffers can be large (of the order of megabytes) and precise modeling of buffers will blow up the state space.

Second, the packets may be processed in arbitrary interleaved orders, and the processing of one packet may influence the processing of subsequent ones because the controller may update flow tables based on the first packet. Similarly, control messages between the controller and the switches may be processed in arbitrary order and this may lead to potential bugs, including the bug pointed to above.

Kuai handles these challenges in the following way. First, instead of modeling unbounded multisets for packet queues, we implement a *counter abstraction* where we track, for each possible packet, whether zero or arbitrarily many instances of the packet are waiting in a multiset. This abstraction enables us to apply finite-state model checking approaches.

Second, we implement a set of partial-order reduction techniques that are specific to the SDN domain. For example, we note that while in principle a switch only processes one packet at a time, we do not lose behaviors by processing all packets at the packet queue of a switch atomically. Similarly, using the semantics of the barrier message [12], we show that a switch can atomically execute all control messages up to the last barrier in its control queue. Specifically, this optimization enables the model checker to bound the size of control queues. Additionally, we show that whenever there is a packet in a client's packet queue, the client can receive and process it immediately, so that sends from switches can be atomically processed with receives at clients. Finally, we show that we can eagerly serve requests to the controller, that is, we do not lose behaviors if we restrict the controller's request queue to size one and service these requests as soon as they appear.

We empirically demonstrate that our set of partial order reduction techniques significantly reduces the state spaces of SDN benchmarks, often by many orders of magnitude. For the simple SSH example, the number of explored states is approximately 2 million without partial order reductions, but only 13 with reductions!

To handle large state spaces, our model checker Kuai distributes the model checking over a number of nodes in a cluster, using the PReach distributed model checker [2] (based on Murphi [4]) as its back end. The large-scale distribution enables Kuai to model check large state spaces quickly.

**Related Work.** There is a lot of systems and networking interest in SDNs [9], [5] and standards such as Openflow [12]. From the formal methods perspective, research has focused on verified programming language frameworks for writing SDN controllers [6], [8]. Here, verification refers to correct compilation from Frenetic to executable code, or to checking composability of programs, not the correctness of invariants.

Previous model checking attempts for SDNs mostly focused either on proving a static snapshot of the network [10] or on model checking or symbolic simulation techniques for a fixed number of packets [3], [14]. Recent work extended to controller updates and arbitrary number of packets [17], but used a manual process to add non-interference lemmas. In contrast, our technique automatically deals with unboundedly many packets and, thanks to the partial-order techniques, scales to much larger configurations than reported in [17]. Program verification for SDN controllers using loop invariants and SMT solving has been proposed recently [1]. While the invariants can quantify over the network (and therefore not limited to finite topologies), the model of the network ignores asynchronous interleavings of packet and control message processing that we handle here.

Our work builds on top of distributed enumerative model checking and the PReach tool [2]. Our contribution is identifying domain specific state space reduction heuristics that enable us to explore large configurations.

## II. Software-defined Networks

**Preliminaries**. A *multiset* $m$ over a set $\Sigma$ is a function $\Sigma \to \mathbb{N}$ with finite support (i.e., $m(\sigma) \neq 0$ for finitely many $\sigma \in \Sigma$). By $\mathbb{M}[\Sigma]$ we denote the set of all multisets over $\Sigma$. We shall write $m = [\![\sigma_1^2, \sigma_3]\!]$ for the multiset $m \in \mathbb{M}[\{\sigma_1,\sigma_2,\sigma_3\}]$ with $m(\sigma_1) = 2, m(\sigma_2) = 0$, and $m(\sigma_3) = 1$. We write $\emptyset$ for an empty multiset, mapping each $\sigma \in \Sigma$ to 0. We write $\{\}$ for an empty set. Two multisets are ordered by $m_1 \leq m_2$ if for all $\sigma \in \Sigma$, we have $m_1(\sigma) \leq m_2(\sigma)$. Let $m_1 \oplus m_2$ (resp. $m_1 \ominus m_2$) be the multiset that maps every element $\sigma \in \Sigma$ to $m_1(\sigma) + m_2(\sigma)$ (resp. $\max\{0, m_1(\sigma) - m_2(\sigma)\}$).

Given a set of states, a *(guarded) action* $\alpha$ is a pair $(g, c)$ where $g$ is a *guard* that evaluates the states to a boolean and $c$ is a *command*. A action $\alpha$ is *enabled* in a state $s$ if the guard of $\alpha$ evaluates $s$ to true. If $\alpha$ is enabled in $s$, the command of $\alpha$ can execute and lead to a new state $s'$, denoted by $s \xrightarrow{\alpha} s'$. We write $\alpha(s) = s'$ if $s \xrightarrow{\alpha} s'$. A *transition system* $TS$ is a tuple $(S, A, \to, s_0, AP, L)$ where $S$ is a set of states, $A$ is a set of actions, $\to \subseteq S \times A \times S$ is a transition relation, $s_0 \in S$ is the initial state, $AP$ is a set of atomic propositions, and $L : S \to 2^{AP}$ is a labeling function. We write $\to^*$ for the reflexive transitive closure of $\to$. A state $s'$ is *reachable* from $s$ if $s \to^* s'$. We write $s \to^+ s'$ if there is a state $t$ such that $s \to t \to^* s'$. For a state $s$, let $A(s)$ be the set of actions enabled in $s$; we assume $A(s) \neq \emptyset$ for each $s \in S$. The *trace* of an infinite execution $\rho = s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \dots$ is defined as $trace(\rho) = L(s)L(s_1)\dots$. The trace of a finite execution $\rho = s \xrightarrow{\alpha_1} s_1 \xrightarrow{\alpha_2} \dots \xrightarrow{\alpha_n} s_n$ is defined as $trace(\rho) = L(s)L(s_1)\dots L(s_n)$. An execution is *initial* if it starts in $s_0$. Let $Traces(TS)$ be the set of traces of initial executions in $TS$. We define invariants and invariant satisfaction in the usual way.

**Syntax of Software-defined Networks** We model an SDN as a network consisting of *nodes*, *connections*, and a *controller*

program. Nodes come from a finite set *Clients* of *clients* and a (disjoint) finite set *Switches* of *switches*. Each node $n$ has a finite set of *ports* $Port(n) \subseteq \mathbb{N}$ which are connected to ports of other nodes. A *location* $(n, pt)$ is a pair of a node and a port $pt \in Port(n)$. Let *Loc* be the set of locations. A connection is a pair of locations. A network is well-formed if there is a bijective function $\lambda : Loc \to Loc$, called the *topology function*, such that $\{((n, pt), \lambda(n, pt)) \mid (n, pt) \in Loc\}$ is the set of connections and no two clients are connected directly.

We model a *packet* $pkt$ in the network as a tuple $(a_1, \ldots, a_k, loc)$, where $(a_1, \ldots, a_k) \in \{0, 1\}^k$ models an abstraction of the packet data and $loc \in Loc$ indicates the location of $pkt$. Let *Packet* be the set of all packets.

Each switch contains a set of rules that determine how packets are forwarded. A *rule* is a tuple $(priority, pattern, ports)$, where $priority \in \mathbb{N}$ determines the priority of the rule, $pattern$ is a proposition over *Packet*, and $ports$ is a multiset of ports. We write *Rule* to denote the set of all rules. Intuitively, a packet matches a rule if it satisfies $pattern$. A switch forwards a packet along $ports$ for the highest priority rule that matches.

Rules are added or deleted on a switch by the controller through a set of *control messages* $CM = \{\mathsf{add}(r), \mathsf{del}(r) \mid r \in Rule\}$. Additionally, the controller uses a *barrier* message $b$ to synchronize.

```
type client {
  Port : set of nat
  pq : multiset of packets
}
rule "send(c, pkt)"
  true ==> send(c, pkt)
end
rule "recv(c,pkt,pkts)"
  exist(pkt:c.pq, true) ==> recv(c,pkt,pkts)
end
```

Listing 2: Client

A client $c \in Clients$ is modeled as in List 2. It consists of a finite set *Port* of ports and a *packet queue* $pq \in \mathbb{M}[Packet]$ containing a multiset of packets which have arrived at the client. We use (guarded) actions to model behaviors of clients. An action is written as "**rule** *name guard* $\Longrightarrow$ *command* **end**." Predicate $exist(i : X, \varphi)$ asserts that there is an element $i$ in the set (or multiset) $X$ such that the predicate $\varphi$ holds. Additionally, if $exist(i : X, \varphi)$ holds, then the variable $i$ is bound to an element of $X$ that satisfies $\varphi$ and can be used later in the command part. In each step, a client $c$ can (1) send a non-deterministically chosen packet $pkt$ along some ports (rule send), or (2) receive a packet $pkt$ from its packet queue and (optionally) send a multiset of packets $pkts$ on some ports (rule recv).

A switch $sw$ is modeled as in List 3. It consists of a set of ports, a *flow table* $ft \subseteq Rule$, a packet queue $pq$ containing packets arriving from neighboring nodes, a *control queue* $cq$ containing control messages or barriers from the controller, a *forward queue* $fq$ consisting of at most one pair $(pkt, ports)$ through which the controller tells the switch to forward packet $pkt$ along the ports $ports$, and a boolean variable $wait$. Predicate $noBarrier(sw)$ asserts $sw.cq$ does not contain a barrier. Predicate $bestmatch(sw, r, pkt)$ asserts that $r$ is the highest priority rule whose pattern matches the packet $pkt$ in switch $sw$'s flow table.

Intuitively, a switch has a normal mode and a waiting mode determined by the $wait$ variable. When the switch is in the normal mode, as long as there is no barrier in its control queue, it can either attempt to forward a packet from its packet queue

```
type switch {
  Port : set of nat
  ft : set of rules
  pq : multiset of packets
  cq : list of barriers and
       multisets of control messages
  fq : set of forward messages
  wait : boolean
}
rule "match(sw,pkt,r)"
  !sw.wait & noBarrier(sw) &
  exist(pkt:sw.pq,
    exist(r:sw.ft, bestmatch(sw,r,pkt))) ==>
  match(sw,pkt,r)
end
rule "nomatch(sw,pkt)"
  !sw.wait & noBarrier(sw) & !RqFull(controller) &
  exist(pkt:sw.pq,
    !exist(r:sw.ft,bestmatch(sw,r,pkt))) ==>
  nomatch(sw,pkt)
end
rule "add(sw,r)"
  !sw.wait & noBarrier(sw) &
  exist(add(r):sw.cq[0],true) ==>
  add(sw,r)
end
rule "delete(sw,r)"
  !sw.wait & noBarrier(sw) &
  exist(del(r):sw.cq[0],true) ==>
  delete(sw,r)
end
rule "fwd(sw,pkt,pts)"
  sw.wait & noBarrier(sw) &
  exist((pkt,pts):fq, true) ==>
  fwd(sw,pkt,pts)
end
rule "barrier(sw)"
  !noBarrier(sw) ==>
  barrier(sw)
end
```

Listing 3: Switch

based on its flow table, or update its flow table according to a control message in its control queue. When the switch cannot find a matching rule in its flow table for a packet, it can initiate a request to the controller, change to the waiting mode, and wait for a forward message from the controller telling it how to forward the packet. Once it receives a forward message $(pkt, pts)$ and there is no barrier in the control queue, it forwards the pending packet $pkt$ to the ports in $pts$, and changes back to the normal mode. If the control queue contains one or more barriers, the switch dequeues all control messages up to the first barrier from its control queue and updates its flow table.

```
type controller {
  CS : set of control states
  cs0 : CS                    cs : CS
  rq : set of packets         κ : ℕ+
  pktIn : function
}
rule "ctrl(pkt,cs)"
  exist(pkt:controller.rq, true) ==>
  ctrl(pkt,controller.cs)
end
```

Listing 4: Controller

A controller *controller* is modeled as in List 4. It is a tuple $(CS, cs_0, cs, rq, \kappa, pktIn)$ where $CS$ is a finite set of *control states*, $cs_0 \in CS$ is the *initial control state*, $cs$ is the *current control state*, $rq$ is a finite *request queue* of size $\kappa \geq 1$ consisting of packets forwarded to the controller from switches, and $pktIn$ is a function that takes a packet $pkt$ and a control state $cs_1$, and returns a tuple $(\eta, (pkt, pts), cs_2)$ where $\eta$ is a function from *Switches* to $(\mathbb{M}[CM] \cup \{b\})^*$, $(pkt, pts)$

is a forward message, and $cs_2$ is a control state. Intuitively, in each step, the controller removes a packet $pkt$ from $rq$ and executes $pktIn(pkt, controller.cs)$. Based on the result $(\eta, (pkt, pts), cs')$, it sends back to the source of the packet the forward message $(pkt, pts)$ that specifies $pkt$ should be forwarded along $pts$, and goes to a new control state $cs'$. Further, for each switch $sw$ in the network it appends $\eta(sw)$ to $sw$'s control queue.

**Semantics of Software-defined Networks** The semantics of an SDN is given as a transition system. Let $\mathcal{N} = (Clients, Switches, \lambda, Packet, Rule, controller)$ be an SDN, where each component is as defined above.

A *state* $s$ of the SDN $\mathcal{N}$ is a quadruple $(\pi, \delta, cs, rq)$, where $\pi$ is a function mapping each client $c \in Clients$ to its packet queue $pq$ and $\delta$ is a function mapping each switch $sw \in Switches$ to a tuple $(pq, cq, fq, ft, wait)$ consisting of its packet queue, control queue, forward queue, flow table, and the wait variable.

For a non-empty list $l = [x_1, x_2, \ldots, x_n]$, define $l.hd{=}x_1$, $l.tl{=}[x_2, \ldots, x_n]$, and $l[i]$ as the $i$-th element in $l$. Given two lists $l_1$ and $l_2$, let $l_1@l_2$ be the concatenation of $l_1$ and $l_2$. For two non-empty lists $l_1 = [x_1, \ldots, x_m]$ and $l_2 = [y_1, \ldots, y_n]$ in $(\mathbb{M}[CM] \cup \{b\})^*$, define $l_1{+}l_2$ be the list $[x_1, \ldots, x_{m-1}, x_m \oplus y_1, y_2, \ldots, y_n]$ if $x_m \neq b$ and $y_1 \neq b$; $l_1@l_2$ otherwise.

Given a flow table $ft$ and a list $l \in (\mathbb{M}[CM] \cup b)^*$, let $update(ft, l)$ be a procedure that updates $ft$ based on $l$ as follows. It dequeues the head of $l$ and sets $l$ to $l.tl$. If the head is a barrier $b$, then ignore it. If the head is a multiset $m$, it nondeterministically chooses a *fetching order* $p$ and based on $p$, removes a control message $cm$ with $m(cm) > 0$ from $m$. If $cm$ is $\mathsf{add}(r)$, then add the rule $r$ to $ft$, or if $cm$ is $\mathsf{del}(r)$, then delete $r$ from $ft$. It keeps updating $ft$ based on $p$ until $m$ becomes empty. It repeats the above instructions on $l$ until $l$ becomes empty. Then it returns the resulting flow table $ft$.

For a function $f\colon X \to Y$, $x \in X$, and $y \in Y$, let $f[x \mapsto y]$ denote the function that maps $x$ to $y$ and all $x' \neq x$ to $f(x')$. Let $f[x_1 \mapsto y_1; x_2 \mapsto y_2; \ldots; x_n \mapsto y_n]$ denote the function $f[x_1 \mapsto y_1][x_2 \mapsto y_2] \ldots [x_n \mapsto y_n]$. Given a subset $X' = \{x_1, \ldots, x_n\} \subseteq X$, let $f[\text{foreach } x_i \in X' : x_i \mapsto y_i]$ be the function $f[x_1 \mapsto y_1] \ldots [x_n \mapsto y_n]$ where $1 \leq i \leq n$. Given a tuple $t = (f_1, \ldots, f_n)$, let $t.f_i$ be the field $f_i$, for $1 \leq i \leq n$. By abuse of notation, we write $t[f_i \mapsto v]$ to be the tuple such that $t[f_i \mapsto v].f_i = v$ and for any $j \neq i$, $t[f_i \mapsto v].f_j = t.f_j$.

We define the following *basic operations* over $\delta$ and $\pi$:

1) Add or delete packets in switches or in clients. Given a set $X \subseteq Switches \times Packet^{\mathbb{N}}$, define $addPkt(\delta, X) = \delta[\text{foreach } (sw, pkt^k) \in X, sw \mapsto \delta(sw)[pq \mapsto \delta(sw).pq \oplus [\![pkt^k]\!]]]$. Given a set $Y \subseteq Clients \times Packet^{\mathbb{N}}$, define $addPkt(\pi, Y) = \pi[\text{foreach } (c, pkt^k) \in Y, c \mapsto \pi(c) \oplus [\![pkt^k]\!]]$. We define $delPkt(\delta, X)$ and $delPkt(\pi, Y)$ analogously by replacing $\oplus$ with $\ominus$ above.
2) Set the $wait$ bit of a switch $sw$ to true or false. Define $setWait(\delta, sw) = \delta[sw \mapsto \delta(sw)[wait \mapsto true]]$ and $unsetWait(\delta, sw) = \delta[sw \mapsto \delta(sw)[wait \mapsto false]]$.
3) Add or delete a rule $r$ in the flow table of a switch $sw$. Define $addRule(\delta, sw, r) = \delta[cq \mapsto [\delta(sw).cq.hd \ominus [\![\mathsf{add}(r)]\!]; sw \mapsto \delta(sw)[ft \mapsto \delta(sw).ft \cup \{r\}]]$. Define $delRule(\delta, sw, r) = \delta[cq \mapsto [\delta(sw).cq.hd \ominus [\![\mathsf{del}(r)]\!]; sw \mapsto \delta(sw)[ft \mapsto \delta(sw).ft \backslash \{r\}]]$.
4) Add or delete a forward message $msg$ in a switch $sw$. Define $addFwdMsg(\delta, sw, msg) = \delta[sw \mapsto \delta(sw)[fq \mapsto \delta(sw).fq \cup \{msg\}]]$ and $delFwdMsg(\delta, sw, msg) = \delta[sw \mapsto \delta(sw)[fq \mapsto \delta(sw).fq \backslash \{msg\}]]$.
5) Flush and run all control messages up to the first barrier in a switch. Define $flush(\delta, sw) = \delta[sw \mapsto \delta(sw)[cq \mapsto l; ft \mapsto update(\delta(sw).ft, [m, b])]]$ where $l = [\emptyset]$, if $\delta(sw).cq = [m, b]$; $l = l'$, if $\delta(sw).cq = [m, b]@l'$ and $l'$ is not an empty list.
6) Flush and run all control messages up to the last barrier in a switch. Define $flushall(\delta, sw) = \delta[sw \mapsto \delta(sw)[cq \mapsto l_1; ft \mapsto update(\delta(sw).ft, l_2)]]$ where $l_1 = [\emptyset]$ and $l_2 = \delta(sw).cq$ if the last element of $\delta(sw).cq$ is a barrier. Otherwise, let $\delta(sw).cq{=}l@[m]$. Then $l_1{=}[m]$ and $l_2{=}l$.
7) Add control messages and barriers to the control queues of the switches. Given a total function $f : Switches \to (\mathbb{M}[CM] \cup \{b\})^*$, define $addCtrlCmd(\delta, f) = \delta[\text{foreach } sw \in Switches : sw \mapsto \delta(sw)[cq \mapsto \delta(sw).cq + f(sw)]]$.

For a switch $sw$, a packet $pkt$, and a multiset of ports $pts$, let $FwdToC(sw, pkt, pts)$ be a set $\{(c, pkt'^k) \mid \exists pt \in sw.Port.\ pts(pt) = k \wedge \lambda(sw, pt) = (c, pt') \wedge c \in Clients \wedge pkt' = pkt[loc \mapsto (c, pt')]\}$ and $FwdToSw(sw, pkt, pts)$ be a set $\{(sw', pkt'^k) \mid \exists pt \in sw.Port.\ pts(pt) = k \wedge \lambda(sw, pt) = (sw', pt') \wedge sw' \in Switches \wedge pkt' = pkt[loc \mapsto (sw', pt')]\}$. Intuitively, when $sw$ is about to forward $pkt$ on its ports $pts$, these two sets summarize how many packets should be forwarded to its connected clients and switches.

For an SDN $\mathcal{N}$, let $Send = \{send(c, pkt) \mid c \in Clients \wedge pkt \in Packet\}$ be the set of *send actions*. We define analogously the set of *receive actions* $Recv$, the set of *match actions* $Match$, the set of *no-match actions* $NoMatch$, the set of *add actions* $Add$, the set of *delete actions* $Del$, the set of *forward actions* $Forward$, the set of *barrier actions* $Barrier$, and the set of *control actions* $Ctrl$.

Let $\pi_0 = \lambda c \in Clients.\emptyset$ and $\delta_0 = \lambda sw \in Switches.(\emptyset, [\emptyset], \{\}, \{\}, false)$. The semantics of an SDN $\mathcal{N}$ is given by a transition system $TS(\mathcal{N}) = (S, A, \to, s_0, AP, L)$. Here, $S$ is the set of states, $s_0 = (\pi_0, \delta_0, cs_0, \{\})$ is the initial state, and $A = Send \cup Recv \cup Match \cup NoMatch \cup Add \cup Del \cup Forward \cup Barrier \cup Ctrl$. The transition relation $s \xrightarrow{\alpha} s'$ is defined as follows.

1) $\alpha = send(c, pkt).\ (\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs, rq)$ where $\delta' = addPkt(\delta, \{(sw, pkt)\})$ and $sw = pkt.loc.n$.
2) $\alpha = recv(c, pkt, pkts).\ (\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi', \delta', cs, rq)$ where $\pi' = delPkt(\pi, \{(c, pkt)\})$, $\delta' = addPkt(\delta, X)$ and $X = \{(sw, pkt'^k) \mid pkts(pkt') = k \wedge pkt'.loc.n = sw\}$.
3) $\alpha = match(sw, pkt, r).\ (\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi', \delta', cs, rq)$ where $\pi' = addPkt(\pi, FwdToC(sw, pkt, r.ports))$ and $\delta' = addPkt(\delta, FwdToSw(sw, pkt, r.ports))$.
4) $\alpha = nomatch(sw, pkt).\ (\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs, rq')$ where $rq' = rq \cup \{pkt\}$, $\delta'' = delPkt(\delta, \{(sw, pkt)\})$, and $\delta' = setWait(\delta'', sw)$.
5) $\alpha = add(sw, r).\ (\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs, rq)$ where $\delta' = addRule(\delta, sw, r)$.
6) $\alpha = del(sw, r).\ (\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs, rq)$ where $\delta' = delRule(\delta, sw, r)$.
7) $\alpha = fwd(sw, pkt, pts).\ (\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi', \delta', cs, rq)$ where $\pi' = addPkt(\pi, FwdToC(sw, pkt, pts))$, $\delta_1 = delFwdMsg(\delta, sw, (pkt, pts))$, $\delta_2 = addPkt(\delta_1, FwdToSw(sw, pkt, pts))$, and $\delta' = unsetWait(\delta_2, sw)$.
8) $\alpha = barrier(sw).\ (\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs, rq)$ where $\delta' = flush(\delta, sw)$.

9) $\alpha = ctrl(pkt, cs)$. Let $pktIn(pkt, cs) = (\eta, msg, cs')$ and $sw = pkt.loc.n$. $(\pi, \delta, cs, rq) \xrightarrow{\alpha} (\pi, \delta', cs', rq')$ where $rq' = rq \backslash \{pkt\}$, $\delta'' = addFwdMsg(\delta, sw, msg)$, and $\delta' = addCtrlCmd(\delta'', \eta)$.

An atomic proposition $p \in AP$ is an assertion over packet fields or over control states. Define an SDN specification as a safety property $\Box\phi$ where $\phi$ is a formula over $AP$ and $\Box$ is the "globally" operator of linear-temporal logic. The *model checking problem for an SDN* asks, given an SDN $\mathcal{N}$ and an SDN specification $\Box\phi$, if $TS(\mathcal{N})$ satisfies $\Box\phi$. For example, blocking SSH packets can be specified as $\Box \bigwedge_{pkt \in Packet}(pkt.loc.n \in Clients \wedge pkt.src \in Clients \wedge pkt.loc.n \neq pkt.src \Rightarrow pkt.prot \neq \text{SSH})$.

# III. Optimizations

We now describe partial-order reduction and abstraction techniques that reduce the state space. These techniques use the structure of SDNs and, as we demonstrate empirically, are crucial in making the model checking scale to non-trivial examples. We state the correctness theorems; the proofs are in the technical report [11].

**Partial Order Reduction** Let $TS = (S, A, \rightarrow, s_0, AP, L)$ be an action-deterministic transition system, i.e., $s \xrightarrow{\alpha} s'$ and $s \xrightarrow{\alpha} s''$ implies $s' = s''$. Given two actions $\alpha, \beta \in A$ with $\alpha \neq \beta$, $\alpha$ and $\beta$ are *independent* if for any $s \in S$ with $\alpha, \beta \in A(s)$, $\beta \in A(\alpha(s))$, $\alpha \in A(\beta(s)))$, and $\alpha(\beta(s)) = \beta(\alpha(s))$. The actions $\alpha$ and $\beta$ are dependent if $\alpha$ and $\beta$ are not independent. An action $\alpha \in A$ is a *stutter action* if for each transition $s \xrightarrow{\alpha} s'$ in $TS$, we have $L(s) = L(s')$.

For $i \in \{1, 2\}$, let $TS_i = (S_i, A_i, \rightarrow_i, s_0^i, AP, L_i)$ be transition systems. Infinite executions $\rho_1$ of $TS_1$ and $\rho_2$ of $TS_2$ are *stutter-equivalent*, denoted $\rho_1 \triangleq \rho_2$, if there is an infinite sequence $A_0 A_1 A_2 \ldots$ with $A_i \subseteq AP$, and natural numbers $n_0, n_1, n_2, \ldots, m_0, m_1, m_2, \ldots \geq 1$ such that

$$trace(\rho_1) = \underbrace{A_0 \ldots A_0}_{n_0 \text{ times}} \underbrace{A_1 \ldots A_1}_{n_1 \text{ times}} \underbrace{A_2 \ldots A_2}_{n_2 \text{ times}} \ldots$$

$$trace(\rho_2) = \underbrace{A_0 \ldots A_0}_{m_0 \text{ times}} \underbrace{A_1 \ldots A_1}_{m_1 \text{ times}} \underbrace{A_2 \ldots A_2}_{m_2 \text{ times}} \ldots$$

$TS_1$ and $TS_2$ are *stutter equivalent*, denoted $TS_1 \triangleq TS_2$ , if $TS_1 \trianglelefteq TS_2$ and $TS_2 \trianglelefteq TS_1$, where $\trianglelefteq$ is defined by: $TS_1 \trianglelefteq TS_2$ iff for all $\rho_1 \in Traces(TS_1)$. $\exists \rho_2 \in Traces(TS_2)$. $\rho_1 \triangleq \rho_2$.

## A. Barrier Optimization

Intuitively, barrier optimization uses the observation that for any state, we can always flush out control queues of switches until there are no barriers in them. This implies that after a control action is executed, one can immediately update flow tables of switches whose control queue has barriers added by the controller. Hence a control action and successive barrier actions can be merged. We prove its correctness by viewing it as an instance of partial order reduction.

For an SDN $\mathcal{N}$, note that $TS(\mathcal{N})$ is not action-deterministic due to barrier actions. With different fetching orders, $barrier(sw)$ may lead to multiple states. Define $b(s, sw)$ as the number of transitions of the form $s \xrightarrow{barrier(sw)} s'$. Note that a barrier action from any $s$ leads to at most $2^{|Rule|}$ states. Hence for each transition $s \xrightarrow{barrier(sw)} s_i$ where $1 \leq i \leq b(s, sw)$, we can append the action with the index $i$, i.e., $s \xrightarrow{barrier(sw)_i} s_i$. In the following, we redefine the set $Barrier = \{barrier(sw)_i \mid sw \in Switches \wedge 1 \leq i \leq 2^{|Rule|}\}$, and assume that $TS(\mathcal{N})$ is action-deterministic by renaming barrier actions.

A switch $sw$ has a barrier iff there is a barrier in $sw$'s control queue. A state $s$ has a barrier, denoted $hasb(s)$, iff some switch $sw \in Switches$ has a barrier in $s$. Define the *ample set* for every state $s$ in $TS(\mathcal{N})$ as follows: if $s$ has a barrier, then $ample(s) = \{barrier(sw)_i \mid 1 \leq i \leq b(s, sw) \wedge sw \text{ has a barrier in } s\}$, that is, all barrier actions enabled in $s$. If $s$ does not have a barrier, then $ample(s) = A(s)$.

Given $TS(\mathcal{N})$, we now define a transition system $\widehat{TS} = (\hat{S}, A, \Rightarrow, s_0, AP, L)$ where $\hat{S} = S$ is the set of states, and the transition relation $\Rightarrow$ is defined as: if $s \xrightarrow{\alpha} s'$ and $\alpha \in ample(s)$, then $s \overset{\alpha}{\Rightarrow} s'$.

*Theorem 1:* Let $TS(\mathcal{N})$ be an action-deterministic transition system. $TS(\mathcal{N}) \triangleq \widehat{TS}$.

Intuitively, Theorem 1 holds because any barrier action is independent of other actions and is a stutter action. Hence for an infinite execution $s \xrightarrow{\alpha_1} s_1 \ldots \xrightarrow{\alpha_n} s_n \xrightarrow{barrier(sw)} t$ in $TS(\mathcal{N})$ where $s$ has a barrier and $\alpha_i$ is not a barrier action for all $1 \leq i \leq n$, we can permute $barrier(sw)$ forward until $s$ and obtain a stutter-equivalent execution in $\widehat{TS}$.

Since Theorem 1 holds, we can merge a control action and successive barrier actions into a single transition $s \xrightarrow{ctrl(pkt,cs)}_2 s'$ where we define the new semantics of $ctrl(pkt, cs)$ under the transition relation $\rightarrow_2$. Formally, Let $(\eta, (pkt, pts), cs') = pktIn(pkt, cs)$ and $sw = pkt.loc.n$.

Ctrl. $(\pi, \delta, cs, rq) \xrightarrow{ctrl(pkt,cs)}_2 (\pi, \delta', cs', rq')$ where $rq' = rq \backslash \{pkt\}$. Define $\delta'' = addFwdMsg(\delta, sw, (pkt, pts))$, and $\delta''' = addCtrlCmd(\delta'', \eta)$. Let $\{sw_1, \ldots, sw_n\}$ be the set of all switches whose control queue has barriers in $\delta'''$. Let $\delta_0 = \delta'''$ and $\delta_i = flushall(\delta_{i-1}, sw_i)$ for all $1 \leq i \leq n$. Define $\delta' = \delta_n$.

Given $\widehat{TS} = (\hat{S}, A, \Rightarrow, s_0, AP, L)$, define a transition system $TS_2 = (S_2, A_2, \rightarrow_2, s_0, AP_2, L_2)$ where $S_2 \subseteq \hat{S}$ is a set of states reachable by $\rightarrow_2$, $A_2$ is $A \backslash Barrier$, $AP_2 = AP$, $L_2 = L$, and $\rightarrow_2$ is defined inductively as

$$\frac{s_0 \overset{\alpha}{\Rightarrow} s'}{s_0 \xrightarrow{\alpha}_2 s'} \qquad \frac{s_0 \rightarrow_2^+ s \overset{\alpha}{\Rightarrow} s' \wedge \alpha \notin Ctrl}{s \xrightarrow{\alpha}_2 s'}$$

$$\frac{s_0 \rightarrow_2^+ s \overset{\alpha}{\Rightarrow} t \Rightarrow^* s' \wedge \alpha \in Ctrl \wedge \neg hasb(s')}{s \xrightarrow{\alpha}_2 s'}$$

Since we only remove barrier actions which are stutter actions, we have $TS_2 \triangleq \widehat{TS} \triangleq TS(\mathcal{N})$. Hence we have the following theorem:

*Theorem 2:* Given an SDN $\mathcal{N}$ and a safety property $\Box\phi$, $TS(\mathcal{N})$ satisfies $\Box\phi$ iff $TS_2$ satisfies $\Box\phi$.

## B. Client Optimization

Given transition system $TS_2 = (S_2, A_2, \rightarrow_2, s_0, AP_2, L_2)$, we further reduce the state space by observing that any receive action of a client is a stutter action and is independent of other actions. Formally, we define $ample(s)$ for each state $s \in S_2$ as follows: if there is a client in $s$ such that its packet queue is not empty, then $ample(s) = \{recv(c, pkt, pkts) \mid pkt \text{ is in } c.pq \text{ at } s\}$, that is, all receive actions enabled in $s$. Otherwise, $ample(s) = A(s)$. We now define a transition system $TS_3 = (S_3, A_3, \rightarrow_3, s_0, AP_3, L_3)$ where $S_3 = S_2$,

$A_3 = A_2$, $AP_3 = AP_2$, $L_3 = L_2$, and where the transition relation $\rightarrow_3$ is defined as: if $s \xrightarrow{\alpha}_2 s'$ and $\alpha \in ample(s)$, then $s \xrightarrow{\alpha}_3 s'$.

*Theorem 3:* (1) $TS_2 \triangleq TS_3$. (2) Given a safety property $\Box\phi$, $TS_2$ satisfies $\Box\phi$ iff $TS_3$ satisfies $\Box\phi$.

### C. $(0, \infty)$ Abstraction

The $(0, \infty)$ abstraction bounds the size of packet queues and the multiset in each control queue. The idea is as follows. One can regard a multiset as a counter that counts the number of elements in it exactly. Instead, $(0, \infty)$ abstraction abstracts a multiset so that for each element $e$, it either does not contain $e$ (i.e. 0) or contains unboundedly many copies of $e$ (i.e. $\infty$). Then the size of an abstracted multiset is bounded. Note that for any state $s$ in $TS_3$, any switch's control queue contains exactly one multiset. Hence, the abstraction bounds the length of control queues.

Let $\mathbb{N}_\infty = \mathbb{N} \cup \{\infty\}$ be the extension of the natural numbers with infinity. We naturally extend the addition operation by assuming that $\infty + \infty = \infty$ and $\infty + c = \infty$ for all $c \in \mathbb{Z}$. Given a multiset $m \in \mathbb{M}[D]$ for some finite set $D$, define an *extended multiset* $\mathsf{over}(m)$ such that for each element $d \in D$, $\mathsf{over}(m)(d) = 0$ if $m(d) = 0$, and $\mathsf{over}(m)(d) = \infty$ otherwise. Define $\mathbb{M}[D]^\infty$ as the set of all extended multisets and multisets over $D$. Given a control queue $cq$ with length $n$, let $\mathsf{over}(cq)$ be such that for $1 \leq i \leq n$, $\mathsf{over}(cq)[i] = \mathsf{over}(cq[i])$ if $cq[i] \neq b$; $\mathsf{over}(cq)[i] = b$ otherwise. For $m_1, m_2 \in \mathbb{M}[D]^\infty$, we write $m_1 \leq_e m_2$ iff for all $d \in D$, $m_1(d) \leq m_2(d)$ or $m_2(d) = \infty$. Given two control queues $cq, cq'$ of same length $n$, define $cq \leq_e cq'$ iff for each $1 \leq i \leq n$, $(cq[i] = b \leftrightarrow cq'[i] = b) \wedge (cq[i] \neq b \rightarrow cq[i] \leq_e cq'[i])$.

Given an SDN and the transition system $TS_3 = (S_3, A_3, \rightarrow_3, s_0, AP_3, L_3)$, Define a transition system $TS_4 = (S_4, A_4, \rightarrow_4, s_0, AP_4, L_4)$ where $S_4 = \{\mathsf{over}(s) \mid s \in S_3\}$, $A_4 = A_3$, $AP_4 = AP_3$, and $L_4 = L_3$. The definition of $\rightarrow_4$ is given in detail in [11]. We provide the intuition of $\rightarrow_4$ here: $\rightarrow_4$ is defined so that (1) whenever a packet $pkt$ is added $k \geq 1$ times into a packet queue $pq$, we set $pq$ to $\mathsf{over}(pq \oplus [\![pkt^k]\!])$, and (2) whenever $\eta(sw)$ is added into switch $sw$'s control queue $cq$, we set $cq$ to $\mathsf{over}(cq + \eta(sw))$. The following lemma claims that $TS_4$ simulates $TS_3$, which leads to Theorem 4.

*Lemma 1:* For any infinite initial execution $s_0 \xrightarrow{\beta_1}_3 s_1 \xrightarrow{\beta_2}_3 s_2 \ldots$ in $TS_3$, there is an infinite initial execution $t_0 \xrightarrow{\beta_1}_4 t_1 \xrightarrow{\beta_2}_4 t_2 \ldots$ in $TS_4$ such that for all $i \geq 0$, $s_i = (\pi_i, \delta_i, cs_i, rq_i)$ and $t_i = (\pi'_i, \delta'_i, cs'_i, rq'_i)$ satisfy the following condition: for all $c \in Clients$, $\pi_i(c) \leq_e \pi'_i(c)$ and for all $sw \in Switches$, $\delta_i(sw).pq \leq_e \delta'_i(sw).pq$, $\delta_i(sw).cq \leq_e \delta'_i(sw).cq$, $\delta_i(sw).fq = \delta'_i(sw).fq$, $\delta_i(sw).ft = \delta'_i(sw).ft$, and $\delta_i(sw).wait = \delta'_i(sw).wait$, and $cs_i = cs'_i$, and $rq_i = rq'_i$.

*Theorem 4:* Given a safety property $\Box\phi$, if $TS_4$ satisfies $\Box\phi$ then $TS_3$ satisfies $\Box\phi$.

### D. All Packets in One Shot Abstraction

So far, a switch processes a single packet at a time. We can further reduce the reachable state space by forcing a switch to process all packets matched by some rule at a time. The intermediate states produced by successive match actions in a switch are removed. Let $TS_4 = (S_4, A_4, \rightarrow_4, s_0, AP_4, L_4)$. Define a transition system $TS_5 = (S_5, A_5, \rightarrow_5, s_0, AP_5, L_5)$ where $S_5 = S_4$, $AP_5 = AP_4$, $L_5 = L_4$, $A_5$ is the union of the new "multiple" match actions and $A_4$ excluding the old "single" match actions, and $\rightarrow_5$ is defined as:

$$\frac{s \xrightarrow{\alpha}_4 s' \wedge \alpha \text{ is not a match action}}{s \xrightarrow{\alpha}_5 s'}$$

and if $pkt\_lst = [pkt_1, \ldots, pkt_n]$ and $r\_lst = [r_1, \ldots, r_n]$

$$\frac{s \xrightarrow{match(sw, pkt_1, r_1)}_4 s_1 \ldots s_{n-1} \xrightarrow{match(sw, pkt_n, r_n)}_4 s'}{s \xrightarrow{match(sw, pkt\_lst, r\_lst)}_5 s'}$$

We prove $TS_5$ simulates $TS_4$. We define a relation $R \subseteq S_4 \times S_5$ such that $((\pi, \delta, cs, rq), (\pi', \delta', cs', rq')) \in R$ iff for all $pkt \in Packet$, for all $c \in Clients$, $\pi(c)(pkt) = \infty \rightarrow \pi'(c)(pkt) = \infty$ and for all $sw \in Switches$, $\delta(sw).pq(pkt) = \infty \rightarrow \delta'(sw).pq(pkt) = \infty$, $\delta(sw).cq = \delta'(sw).cq$, $\delta(sw).fq = \delta'(sw).fq$, $\delta(sw).ft = \delta'(sw).ft$, and $\delta(sw).wait = \delta'(sw).wait$, and $cs = cs'$, and $rq = rq'$.

*Theorem 5:* (1)The relation $R$ is a simulation relation. (2)For a safety property $\Box\phi$, if $TS_5$ satisfies $\Box\phi$, then $TS_4$ satisfies $\Box\phi$.

### E. Controller Optimization

We consider a restricted class of SDNs in which the size $\kappa$ of the controller's request queue is one. Under this restriction, we can define a new transition system $TS_6$ that is stutter equivalent to $TS_5$ and has fewer reachable states. The idea is to observe that a no-match action is a stutter action and is independent of any actions before a corresponding control action is executed. Formally, given $TS_5 = (S_5, A_5, \rightarrow_5, s_0, AP_5, L_5)$, we define a new transition relation $\rightarrow_6$ inductively:

$$\frac{s_0 \xrightarrow{\alpha}_5 s'}{s_0 \xrightarrow{\alpha}_6 s'} \qquad \frac{s_0 \rightarrow_6^+ s_1 \xrightarrow{nomatch(sw,pkt)}_5 s_2 \xrightarrow{ctrl(pkt,cs)}_5 s'}{s_1 \xrightarrow{nomatch\_ctrl(sw,pkt,cs)}_6 s'}$$

$$\frac{s_0 \rightarrow_6^+ s_1 \xrightarrow{\alpha}_5 s' \wedge \alpha \text{ is not a no-match action}}{s_1 \xrightarrow{\alpha}_6 s'}$$

where a new action $nomatch\_ctrl(sw, pkt, cs)$ merges $nomatch(sw, pkt)$ and $ctrl(pkt, cs)$ actions. We define a transition system $TS_6 = (S_6, A_6, \rightarrow_6, s_0, AP_6, L_6)$, where $S_6 = S_5$ is the set of states, $A_6$ is the union of all $nomatch\_ctrl(sw, pkt, cs)$ actions and $A_5 \backslash (NoMatch \cup Ctrl)$, $AP_6 = AP_5$, and $L_6 = L_5$.

*Theorem 6:* Given an SDN $\mathcal{N}$ where the size of the request queue of the controller is one, and a safety property $\Box\phi$. (1) $TS_5 \triangleq TS_6$. (2) $TS_5$ satisfies $\Box\phi$ iff $TS_6$ satisfies $\Box\phi$.

## IV. Implementation and Evaluation

Kuai[1] is implemented on top of PReach [2], a distributed enumerative model checker built on Murphi. We model switches, clients, and the controller as concurrent Murphi processes which communicate using message passing, with the queues modeled as multisets. We manually abstract IP packets using predicates used in the controller. We implement $(0, \infty)$-counter abstraction as a library on top of Murphi multisets.

Kuai takes as input topology information such as the number of switches, clients, and their connections, (manually) abstracted packets, and the controller code written as a Murphi process, and invariants written in Murphi syntax. We found it fairly straightforward to port POX [15] controllers due to the imperative features of Murphi. Murphi allows arbitrary first order logic formulas as invariants and it is easy to specify

[1]The tool is can be downloaded at https://github.com/t-saideep/kuai

| Program | Bytes/ state | w/o optimizations States | Time | w/ optimizations States | Time |
|---|---|---|---|---|---|
| SSH 2×2 | 304 | 2,283,527 | 23.52s | 13 | 6.40s |
| ML 3×3 | 320 | 9,109,456 | 89.99s | 5308 | 6.39s |
| ML 6×3 | 748 | | | 23,926,202 | 604.07s |
| ML 9×2 | 1276 | | | 18,615,767 | 793.84s |
| FW(S) 1×2 | 332 | 2,110,986 | 26.89s | 3645 | 5.45s |
| FW(M) 2×4 | 448 | | | 45,507 | 8.03s |
| FW(M) 3×4 | 560 | | | 512,439 | 55.06s |
| FW(M) 4×4 | 676 | | | 5,360,871 | 475.54s |
| RS 4×4 | 764 | | | 4998 | 6.60s |
| RS 4×5 | 764 | | | 590,570 | 82.82s |
| RS 4×6 | 764 | | | 5,112,013 | 327.39s |
| SIM 5×6 | 632 | | | 167 | 6.23s |
| SIM 5×8 | 632 | | | 167 | 6.34s |
| SIM 5×12 | 1108 | | | 167 | 6.85s |

TABLE I: Experimental results. Omitted entries indicate that model checking did not terminate. The number X×Y in the Program column means that there are X switches and Y clients in the example.

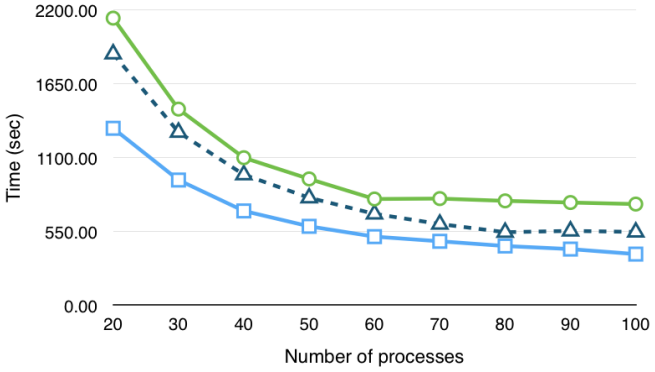


Fig. 2: Verification time vs processes ∘ ML 9×2 Δ ML 6×3 □ FW(M) 4×4

safety properties. Kuai compiles them into a single Murphi file and the model checking effort is then distributed across several machines using PReach. Finally the output of the tool is an error trace if the program invariant fails, or *success* otherwise.

We have evaluated Kuai on a number of real world OpenFlow benchmarks. The experiments were performed on a cluster of 5 Dell R910 rack servers each with 4 Intel Xeon X7550 2GHz processors, 64 x 16GB Quad Rank RDIMMs memory and 174GB storage. Our experiments had access to a total of 150 cores and had access to 4TB of RAM.

Table I shows a summary of experimental results and compares against model checking without the optimizations from Section III. Empty rows indicate model checking did not terminate in 1 hour or ran out of memory. Figure 2 shows the scalability of model checking with increasing distribution on the three largest examples. We noticed that the performance of the distributed model checker plateaued around 70 Erlang processes on these and other large examples. Thus, times (in table I) are provided for configurations that use 70 Erlang processes. As we introduced abstractions, it is possible that we get false positives. We verified the existence of all bugs reported by Kuai manually and there were no false positives.

Besides the table, we plot the MAC learning example in Figure 3, which shows how significantly our optimization techniques reduce the state space. Though we still suffer from the state-space explosion problem, our optimizations delay it and enable us to verify SDNs with much larger configurations.

We now describe the benchmarks in detail.

**SSH** We run Kuai on the SSH controller from Listing 1. It finds the control message reordering bug in 0.1 seconds. By adding a barrier after line 15, Kuai proves the correctness in 6.4 seconds by exploring 13 states. In contrast, the unoptimized version explores over 2 million states.

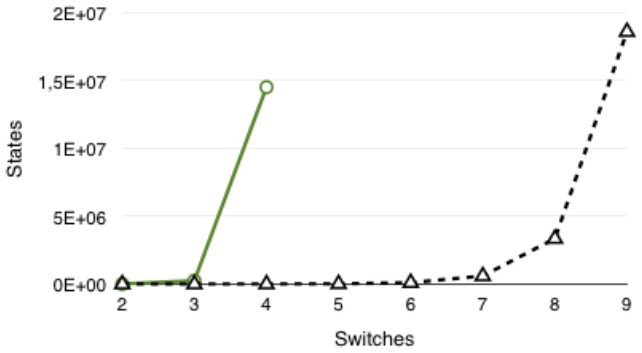


Fig. 3: State space of MAC learning controller: Δ: optimized, ∘ unoptimized

**MAC Learning Controller (ML)** This is based on the POX [15] implementation of the standard ethernet discovery protocol. We checked there are no forwarding loops (similar to [17]), i.e., a packet should not reach a switch more than once. Packets are augmented with a bit for each switch which gets set when the switch processes that packet. The invariant is specified using these visit-bits (called *reached*): $\Box\, \forall sw \in Switches.\ \forall pkt \in sw.pq.\ (\neg pkt.reached(sw))$.

A cycle in the topology will lead to forwarding loops as the controller does not compute the minimum spanning tree. We discover the bug in a cyclic topology of 3 switches 3 clients in 0.47 seconds. We re-ran the example on a topology containing the minimum spanning tree of the original cyclic topology and the tool is able to prove that there were no forwarding loops in 6.39 seconds. We scale the example by adding more switches. We notice that while the verification on topology with 9 switches and 2 clients has fewer states than the one with 6 switches and 3 clients, each state in the latter case is bigger than the former and hence the memory and communication overheads are higher.

**Single Switch Firewall (FW(S))** This is based on an advanced GENI assignment [7] on building an OpenFlow based firewall. The controller takes as input a simple configuration file which is a list of tuples of the form (client1, port1, client2, port2). This specifies that packets originating from client1 on port1 can be forwarded to client2 on port2. We abbreviate the tuples as $(client1{:}\ port1 \rightarrow client2{:}\ port2)$. Any flow not explicitly allowed is forbidden. The flows are uni-directional and the above flow will reject traffic initiated by client2 on port2 towards client1 on port1. However, once client1 initiates a flow, the firewall should allow client2 to reply back, making the flow bi-directional until client1 closes the connection.

The naive implementation of the controller is as follows: on receiving a packet $(c1{:}\ p1 \rightarrow c2{:}\ p2)$, check if there is a tuple matching the flow in the policy. If it does, add rules $(c1{:}\ p1 \rightarrow c2{:}\ p2)$ and $(c2{:}\ p2 \rightarrow c1{:}\ p1)$ and forward the packet to $c2$. Otherwise add a rule to drop packets of the form $(c1{:}\ p1 \rightarrow c2{:}\ p2)$. The invariant to verify here is to ensure the policy of the firewall, i.e., a packet from $c1{:}\ p1$ should be forwarded to $c2{:}\ p2$ if and only if $(c1{:}\ p2 \rightarrow c2{:}\ p2)$ exists in the firewall policy or if $(c2{:}\ p2 \rightarrow c1{:}\ p1)$ exists in the policy and $c2$ has already initiated the corresponding flow. The following formula specifies that allowed packets should not be dropped: $\Box \forall p \in Packet.\ on\_dropped(p) \Rightarrow \neg flows[p.src][packet.src\_port][packet.dest][p.dest\_port]$, where *on_dropped(p)* is set if a packet-drop transition is fired on packet $p$ (and reset at the beginning of every transition). *flows* is an auxiliary variable in the controller which keeps

track of allowed flows based on the firewall policy and initiating client.

We ran the experiment on a topology with 2 clients and a firewall. We found an interesting bug in our implementation which is caused by not assigning proper priorities to rules. For example, when $(c1\colon p1 \rightarrow c2\colon p2)$ is present in the policy but not $(c2\colon p2 \rightarrow c1\colon p1)$, the rule to drop flows should have a lower priority than the rules to allow flows. Otherwise, the following bug would occur. If $c2$ initiates the flow $(c2\colon p2 \rightarrow c1\colon p1)$ then the controller adds a rule to drop packets matching that flow. Later on, if $c1$ initiates $(c1\colon p1 \rightarrow c2\colon p2)$ and the controller adds the corresponding rules to allow the flow on both directions, the switch now has two conflicting rules of the same priority. One to allow and the other to drop $(c2\colon p2 \rightarrow c1\colon p1)$. The switch may non-deterministically choose to drop the packet. Once we fixed the bug, the tool could prove the invariant in 5.45 seconds.

**Multiple Switch Firewalls (FW(M))** We extend the above example to include multiple replicated firewalls for load balancing. We now allow the clients to send packets to all of these firewalls. We augment the implementation of the single switch controller to add the same rules on all firewalls. However, this implementation no longer ensures the invariant in the multi-switch setting.

Consider the case with two firewalls, $f1$ and $f2$. The tool reports the following bug: $c1$ initiates $(c1\colon p1 \rightarrow c2\colon p2)$ on firewall $f1$. The controller adds the corresponding rules to allow flows in both directions to $f1$ and $f2$ but only sends a barrier to $f1$. Now $f2$ delays the installation of $(c2\colon p2 \rightarrow c1\colon p1)$ and $c2$ replies back to $c1$ through $f2$ which forwards the packet to the controller. The controller then drops the packet.

The fix here is to add the rules along with barriers on all switches and not just the switch from which the packet originates. With this fix the tool is able to prove the property in 8 seconds. In order to test the scalability, we tested the tool on increasing number of firewalls in the topology.

**Resonance (RS)** Resonance [13] is a system for ensuring security in large networks using OpenFlow switches. When a new client enters the network, it is assigned *registration* state and is only allowed to communicate with a web portal. The portal either authenticates a client by sending a signal to the controller (and the controller assigns the client an *authenticated* state), or sets the client to *quarantined* state. In the authenticated state, the client is only allowed to communicate with a scanner. The scanner ensures that the client is not infected and sends a signal to the controller and lets the controller assign it an *operational* state. If an infection is detected, it is assigned a *quarantined* state. The clients in operational state are periodically scanned and moved to the quarantined state if they are infected. Quarantined clients cannot communicate with other clients.

In our model, the web portal non-deterministically chooses to authenticate or quarantine a client and the scanner non-deterministically marks a client operational or quarantined. We check the invariant that packets from quarantined clients should not be forwarded: $\Box\forall p \in Packet.\ on\_forward(p) \Rightarrow (state(p.src) \neq Quarantined)$. Similar to *on_drop*, *on_forward* is set when packet-forward transition is fired and reset before the beginning of every transition. The controller follows the Resonance algorithm [13].

We ran the experiment on a topology of two clients, one portal, one scanner and four switches. The topology is the same as in Figure 2 of [13] without DHCP and DNS clients. Kuai proves the invariant in 6.6 seconds. We scale up the example by increasing the number of clients.

**Simple (SIM)** Simple [16] is a policy enforcement layer built on top of OpenFlow to ensure efficient middlebox traffic steering. In many network settings, traffic is routed through several middleboxes, such as firewalls, loggers, proxies, etc., before reaching the final destination. Simple takes a middlebox policy as input and translates this to forwarding rules to ensure the policy holds. The invariant ensures that all source packets to a client will be received and forwarded by the middleboxes specified in a given policy before the packet reaches its destination.

We ran the experiment on a topology of two clients, two firewalls, one IDS, one proxy and five switches (see Figure 1 of [16]). Kuai can prove the invariant in 6.48 seconds.

We scale up the example by fixing the destination client and increasing the number of source clients that can send packets to it. Because of our "all packets in one shot" optimization (section III-D), no matter how many packets get queued initially, they are all forwarded in lock-step as the controller forwarding rule applies to all incoming packets.

## References


[1] T. Ball, N. Bjørner, A. Gember, S. Itzhaky, A. Karbyshev, M. Sagiv, M. Schapira, and A. Valadarsky. Vericon: Towards verifying controller programs in software-defined networks. PLDI '14, pages 282–293, 2014.

[2] B. Bingham, J. Bingham, F. de Paula, J. Erickson, G. Singh, and M. Reitblatt. Industrial strength distributed explicit state model checking. In *PDMC-HIBI*, pages 28–36, Sept 2010.

[3] M. Canini, D. Venzano, P. Perešíni, D. Kostić, and J. Rexford. A NICE way to test openflow applications. NSDI'12, pages 127–140, 2012.

[4] D. L. Dill. The Murphi verification system. CAV '96, pages 390–393, London, UK, UK, 1996. Springer-Verlag.

[5] N. Feamster, J. Rexford, and E. Zegura. The road to SDN. *Queue*, 11(12):20:20–20:40, Dec. 2013.

[6] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM.

[7] GENI Assignment. http://groups.geni.net/geni/wiki/GENIEducation/SampleAssignments/OpenFlowFirewallAssignment/ExerciseLayout/Execute.

[8] A. Guha, M. Reitblatt, and N. Foster. Machine-verified network controllers. PLDI '13, pages 483–494, New York,USA, 2013. ACM.

[9] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat. B4: Experience with a globally-deployed software defined wan. SIGCOMM13, pages 3–14, New York, NY, USA, 2013.

[10] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, pages 113–126, 2012.

[11] R. Majumdar, S. Tetali, and Z. Wang. Kuai: A Model Checker for Software-defined Networks. Technical report. http://www.mpi-sws.org/~zilong/papers/kuai_tr.pdf.

[12] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. Openflow: Enabling innovation in campus networks. *SIGCOMM*, 38(2):69–74, Mar. 2008.

[13] A. K. Nayak, A. Reimers, N. Feamster, and R. Clark. Resonance: Dynamic access control for enterprise networks. WREN '09, pages 11–18, New York, NY, USA, 2009. ACM.

[14] T. Nelson, A. Guha, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. A balance of power: Expressive, analyzable controller programming. HotSDN '13, pages 79–84, New York, NY, USA, 2013. ACM.

[15] POX. http://www.noxrepo.org/pox/about-pox/.

[16] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. SIMPLE-fying middlebox policy enforcement using SDN. SIGCOMM13, pages 27–38, New York, NY, USA, 2013. ACM.

[17] D. Sethi, S. Narayana, and S. Malik. Abstractions for model checking SDN controllers. In *Formal Methods in Computer-Aided Design (FMCAD), 2013*, pages 145–148, Oct 2013.