Efficient Synthesis of Network Updates

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

Hossein Hojjat
Cornell University, USA
hojjat@cornell.edu

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

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

Abstract

Software-defined networking (SDN) is revolutionizing the networking industry, but current SDN programming platforms do not provide automated mechanisms for updating global configurations on the fly. Implementing updates by hand is challenging for SDN programmers because networks are distributed systems with hundreds or thousands of interacting nodes. Even if initial and final configurations are correct, naïvely updating individual nodes can lead to incorrect transient behaviors, including loops, black holes, and access control violations. This paper presents an approach for automatically synthesizing updates that are guaranteed to preserve specified properties. We formalize network updates as a distributed programming problem and develop a synthesis algorithm based on counterexample-guided search and incremental model checking. We describe a prototype implementation, and present results from experiments on real-world topologies and properties demonstrating that our tool scales to updates involving over one-thousand nodes.

Categories and Subject Descriptors D.2.4 [*Software Engineering*]: Software/Program Verification—Formal methods; D.2.4 [*Software Engineering*]: Software/Program Verification—Model checking; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—Logics of programs; F.4.1 [*Mathematical Logic and Formal Languages*]: Mathematical Logic—Temporal logic; C.2.3 [*Computer-communication Networks*]: Network Operations—Network Management

Keywords synthesis, verification, model checking, LTL, network updates, software-defined networking, SDN

1. Introduction

Software-defined networking (SDN) is a new paradigm in which a logically-centralized controller manages a collection of programmable switches. The controller responds to events such as topology changes, shifts in traffic load, or new connections from hosts, by pushing forwarding rules to the switches, which process packets efficiently using specialized hardware. Because the controller has global visibility and full control over the entire network, SDN makes it possible to implement a wide variety of network applications ranging from basic routing to traffic engineering, datacenter virtualization, fine-grained access control, etc. [6]. SDN has been used in production enterprise, datacenter, and wide-area networks, and new deployments are rapidly emerging.

Much of SDN's power stems from the controller's ability to change the *global* state of the network. Controllers can set up end-to-end forwarding paths, provision bandwidth to optimize utilization, or distribute access control rules to defend against attacks. However, implementing these global changes in a running network is not easy. Networks are complex systems with many distributed switches, but the controller can only modify the configuration of one switch at a time. Hence, to implement a global change, an SDN programmer must explicitly transition the network through a sequence of intermediate configurations to reach the intended final configuration. The code needed to implement this transition is tedious to write and prone to error—in general, the intermediate configurations may exhibit new behaviors that would not arise in the initial and final configurations.

Problems related to network updates are not unique to SDN. Traditional distributed routing protocols also suffer from anomalies during periods of reconvergence, including transient forwarding loops, blackholes, and access control violations. For users, these anomalies manifest themselves as service outages, degraded performance, and broken connections. The research community has developed techniques for preserving certain invariants during updates [9, 32, 36], but none of them fully solves the problem, as they are limited to specific protocols and properties. For example, *consensus routing* uses distributed snapshots to ensure connectivity, but only applies to the Border Gateway Protocol (BGP) [16].

It might seem that SDN would exacerbate update-related problems by making networks even more dynamic—in particular, most current platforms lack mechanisms for implementing updates in a graceful way. However, SDN offers opportunities to develop high-level abstractions for implementing updates automatically while preserving key invariants. The authors of B4—the controller managing Google's world-wide inter-datacenter network—describe a vision where: "multiple, sequenced manual operations [are] not involved [in] virtually any management operation" [14].

Previous work proposed the notion of a *consistent update* [33], which ensures that every packet is processed either using the initial configuration or the final configuration but not a mixture of the two. Consistency is a powerful guarantee preserving *all* safety properties, but it is expensive. The only general consistent update mechanism is *two-phase update*, which tags packets with versions and maintains rules for the initial/final configurations simultaneously. This leads to problems on switches with limited memory and can also make update time slower due to the high degree of rule churn.

We propose an alternative. Instead of forcing SDN operators to implement updates by hand (as is typically done today), or using powerful but expensive mechanisms like two-phase update, we develop an approach for synthesizing correct update programs efficiently and automatically from formal specifications. Given initial and final configurations and a Linear Temporal Logic (LTL) property capturing desired invariants during the update, we either

PLDI'15, June 13–17, 2015, Portland, OR, USA.

ACM 978-1-4503-3468-6/15/06…$15.00.
http://dx.doi.org/10.1145/2737924.2737980

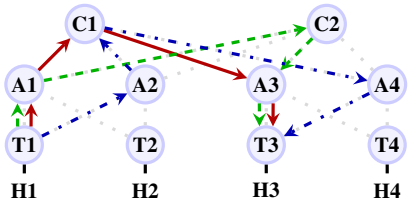

Figure 1: Example topology.

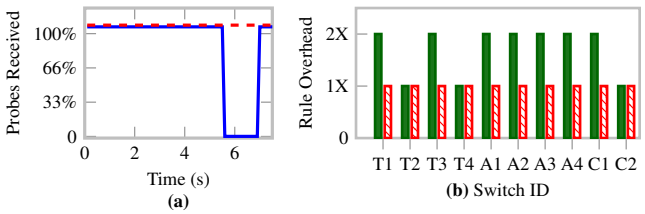

Figure 2: Example naïve (blue/solid-line), two-phase (green/solid-bar), and ordering (red/dashed) updates: (a) probes received; (b) per-switch rule overhead.

generate an SDN program that implements the initial-to-final transition while ensuring that the property is never violated, or fail if no such program exists. Importantly, because the synthesized program is only required to preserve the specified properties, it can leverage strategies that would be ruled out in other approaches. For example, if the programmer specifies a trivial property, the system can update switches in any order. However, if she specifies a more complex property (e.g. firewall traversal) then the space of possible updates is more constrained. In practice, our synthesized programs require less memory and communication than competing approaches.

Programming updates correctly is challenging due to the concurrency inherent in networks—switches may interleave packet and control message processing arbitrarily. Hence, programmers must carefully consider all possible event orderings, inserting synchronization primitives as needed. Our algorithm works by searching through the space of possible sequences of individual switch updates, learning from counterexamples and employing an incremental model checker to re-use previously computed results. Our model checker is *incremental* in the sense that it exploits the loop-freedom of correct network configurations to enable efficient re-checking of properties when the model changes. Because the synthesis algorithm poses a series of closely-related model checking questions, the incrementality yields enormous performance gains on real-world update scenarios.

We have implemented the algorithm and heuristics to further speed up synthesis and eliminate spurious synchronization. We have interfaced the tool with Frenetic [8], synthesized updates for OpenFlow switches, and used our system to process actual traffic generated by end-hosts. We ran experiments on a suite of real-world topologies, configurations, and properties—our results demonstrate the effectiveness of synthesis, which scales to over one-thousand switches, and incremental model checking, which outperforms a popular symbolic model checker used in *batch* mode, and a state-of-the-art network model checker used in incremental mode.

In summary, the main contributions of this paper are:

- We investigate using synthesis to automatically generate network updates (§2).
- We develop a simple operational model of SDN and formalize the network update problem precisely (§3).
- We design a counterexample-guided search algorithm that solves instances of the network update problem, and prove this algorithm to be correct (§4).
- We present an incremental LTL model checker for loop-free models (§5).
- We describe an OCaml implementation with backends to third-party model checkers and conduct experiments on real-world networks and properties, demonstrating strong performance improvements (§6). †

Overall, our work takes a challenging network programming problem and automates it, yielding a powerful tool for building dynamic SDN applications that ensures correct, predictable, and efficient network behavior during updates.

2. Overview

To illustrate key challenges related to network updates, consider the network in Figure 1. It represents a simplified datacenter topology [1] with core switches (C1 and C2), aggregation switches (A1 to A4), top-of-rack switches (T1 to T4), and hosts (H1 to H4). Initially, we configure switches to forward traffic from H1 to H3 along the solid/red path: T1-A1-C1-A3-T3. Later, we wish to shift traffic from the red path to the dashed/green path, T1-A1-C2-A3-T3 (perhaps to take C1 down for maintenance). To implement this update, the operator must modify forwarding rules on switches A1 and C2, but note that certain update sequences break connectivity—e.g., updating A1 followed by C2 causes packets to be forwarded to C2 before it is ready to handle them. Figure 2(a) demonstrates this with a simple experiment performed using our system. Using the Mininet network simulator and OpenFlow switches, we continuously sent ICMP (`ping`) probes during a "naïve" update (blue/solid line) and the ordering update synthesized by our tool (red/dashed line). With the naïve update, 100% of the probes are lost during an interval, while the ordering update maintains connectivity.

Consistency. Previous work [33] introduced the notion of a *consistent update* and also developed general mechanisms for ensuring consistency. An update is said to be consistent if every packet is processed entirely using the initial configuration or entirely using the final configuration, but never a mixture of the two. For example, updating A1 followed by C2 is not consistent because packets from H1 to H3 might be dropped instead of following the red path or the green path. One might wonder whether preserving consistency during updates is important, as long as the network eventually reaches the intended configuration, since most networks only provide best-effort packet delivery. While it is true that errors can be masked by protocols such as TCP when packets are lost, there is growing interest in strong guarantees about network behavior. For example, consider a business using a firewall to protect internal servers, and suppose that they decide to migrate their infrastructure to a virtualized environment like Amazon EC2. To ensure that this new deployment is secure, the business would want to maintain the same isolation properties enforced in their home office. However, a best-effort migration strategy that only eventually reaches the target configuration could step through arbitrary intermediate states, some of which may violate this property.

Two-Phase Updates. Previous work introduced a general consistency-preserving technique called *two-phase update* [33]. The idea is to explicitly tag packets upon ingress and use these version tags to determine which forwarding rules to use at each hop. Unfortunately, this has a significant cost. During the transition, switches must maintain forwarding rules for *both* configurations, effectively doubling the memory requirements needed to complete the update. This is not always practical in networks where the switches store forwarding rules using ternary content-addressable

† The PLDI 2015 Artifact Evaluation Committee (AEC) found that our tool "met or exceeded expectations."

memories (TCAM), which are expensive and power-hungry. Figure 2(b) shows the results of another simple experiment where we measured the total number of rules on each switch: with two-phase updates, several switches have twice the number of rules compared to the synthesized ordering update. Even worse, it takes a non-trivial amount of time to modify forwarding rules—sometimes on the order of 10ms per rule [15]! Hence, because two-phase updates modify a large number of rules, they can increase update latency. These overheads can make two-phase updates a non-starter.

Ordering Updates. Our approach is based on the observation that consistent (two-phase) updates are overkill in many settings. Sometimes consistency can be achieved by simply choosing a correct order of switch updates. We call this type of update an *ordering update*. For example, to update from the red path to the green path, we can update C2 followed by A1. Moreover, even when we cannot achieve full consistency, we can often still obtain sufficiently strong guarantees for a specific application by carefully updating the switches in a particular order. To illustrate, suppose that instead of shifting traffic to the green path, we wish to use the blue (dashed-and-dotted) path: T1-A2-C1-A4-T3. It is impossible to transition from the red path to the blue path by ordering switch updates without breaking consistency: we can update A2 and A4 first, as they are unreachable in the initial configuration, but if we update T1 followed by C1, then packets can traverse the path T1-A2-C1-A3-T3, while if we update C1 followed by T1, then packets can traverse the path T1-A1-C1-A4-T3. Neither of these alternatives is allowed in a consistent update. This failure to find a consistent update hints at a solution: if we only care about preserving connectivity between H1 and H3, then either path is actually acceptable. Thus, either updating C1 before T1, or T1 before C1 would work. Hence, if we relax strict consistency and instead provide programmers with a way to specify properties that must be preserved across an update, then ordering updates will exist in many situations. Recent work [15, 25] has explored ordering updates, but only for specific properties like loop-freedom, blackhole-freedom, drop-freedom, etc. Rather than handling a fixed set of "canned" properties, we use a specification language that is expressive enough to encode these properties and others, as well as conjunctions/disjunctions of properties—e.g. enforcing loop-freedom *and* service-chaining during an update.

In-flight Packets and Waits. Sometimes an additional synchronization primitive is needed to generate correct ordering updates (or correct two-phase updates, for that matter). Suppose we want to again transition from the red path to blue one, but in addition to preserving connectivity, we want every packet to traverse either A2 or A3 (this scenario might arise if those switches are actually middleboxes which scrub malicious packets before forwarding). Now consider an update that modifies the configurations on A2, A4, T1, C1, in that order. Between the time that we update T1 and C1, there might be some packets that are forwarded by T1 before it is updated, and are forwarded by C1 after it is updated. These packets would not traverse A2 or A3, and so indicate a violation of the specification. To fix this, we can simply pause after updating T1 until any packets it previously forwarded have left the network. We thus need a command "wait" that pauses the controller for a sufficient period of time to ensure that in-flight packets have exited the network. Hence, the correct update sequence for this example would be as above, with a "wait" between T1 and C1. Note that two-phase updates also need to wait, once per update, since we must ensure that all in-flight packets have left the network before deleting the old version of the rules on switches. Other approaches have traded off control-plane waiting for stronger consistency, e.g. [24] performs updates in "rounds" that are analogous to "wait" commands, and Consensus Routing [16] relies on timers to obtain wait-like functionality. Note that the single-switch update time can be on the order of seconds [15, 22], whereas typical datacenter transit time (the time for a packet to traverse the network) is much lower, even on the order of microseconds [3]. Hence, waiting for in-flight packets has a negligible overall effect. In addition, our reachability-based heuristic eliminates most waits in practice.

Summary. This paper presents a sound and complete algorithm and implementation for synthesizing a large class of ordering updates efficiently and automatically. The updates we generate initially modify each switch at most once and "wait" between updates to switches, but a heuristic removes an overwhelming majority of unnecessary waits in practice. For example, in switching from the red path to the blue path (while preserving connectivity from H1 to H3, and making sure that each packet visits either A3 or A4), our tool produces the following sequence: update A2, then A4, then T1, then wait, then update C1. The resulting update can be executed using the Frenetic SDN platform and used with OpenFlow switches—e.g., we generated Figure 2 (a-b) using our tool.

3. Preliminaries and Network Model

To facilitate precise reasoning about networks during updates, we develop a formal model in the style of Chemical Abstract Machine [4]. This model captures key network features using a simple operational semantics. It is similar to the one used by [11], but is streamlined to model features most relevant to updates.

3.1 Network Model

Basic structures. Each *switch* sw, *port* pt, or *host* h is identified by a natural number. A *packet* pkt is a record of fields containing header values such as source and destination address, protocol type, and so on. We write $\{f_1; \ldots; f_k\}$ for the type of packets having fields f_i and use "dot" notation to project fields from records. The notation $\{r \text{ with } f = v\}$ denotes functional update of $r.f$.

Forwarding Tables. A switch configuration is defined in terms of forwarding rules, where each rule has a *pattern* pat specified as a record of optional packet header fields and a port, a list of *actions* act that either forward a packet out a given port ($fwd\ pt$) or modify a header field ($f{:=}n$), and a priority that disambiguates rules with overlapping patterns. We write $\{pt?; f_1?; \ldots; f_k?\}$ for the type of patterns, where the question mark denotes an option type. A set of such rules $ruls$ forms a forwarding table tbl. The semantic function $[\![tbl]\!]$ maps packet-port pairs to multisets of such pairs, finding the highest-priority rule whose pattern matches the packet and applying the corresponding actions. If there are multiple matching rules with the same priority, the function is free to pick any of them, and if there are no matching rules, it drops the packet. The forwarding tables collectively define the network's *data plane*.

Commands. The *control plane* modifies the data plane by issuing commands that update forwarding tables. The command (sw, tbl) replaces the forwarding table on switch sw with tbl (we call this a *switch-granularity* update). We model this command as an atomic operation (it can be implemented with OpenFlow *bundles* [31]). Sometimes switch granularity is too coarse to find an update sequence, in which case one can update individual rules (*rule-granularity*). Our tool supports this finer-grained mode of operation, but since it is not conceptually different from switch granularity, we frame most of our discussion in terms of *switch-granularity*.

To synchronize updates involving multiple switches, we include a $wait$ command. In the model, the controller maintains a natural-number counter known as the current epoch ep. Each packet is annotated with the epoch on ingress. The control command $incr$ increments the epoch so that subsequent incoming packets are annotated with the next epoch, and $flush$ blocks the controller until all packets annotated with the previous epoch have exited the network.

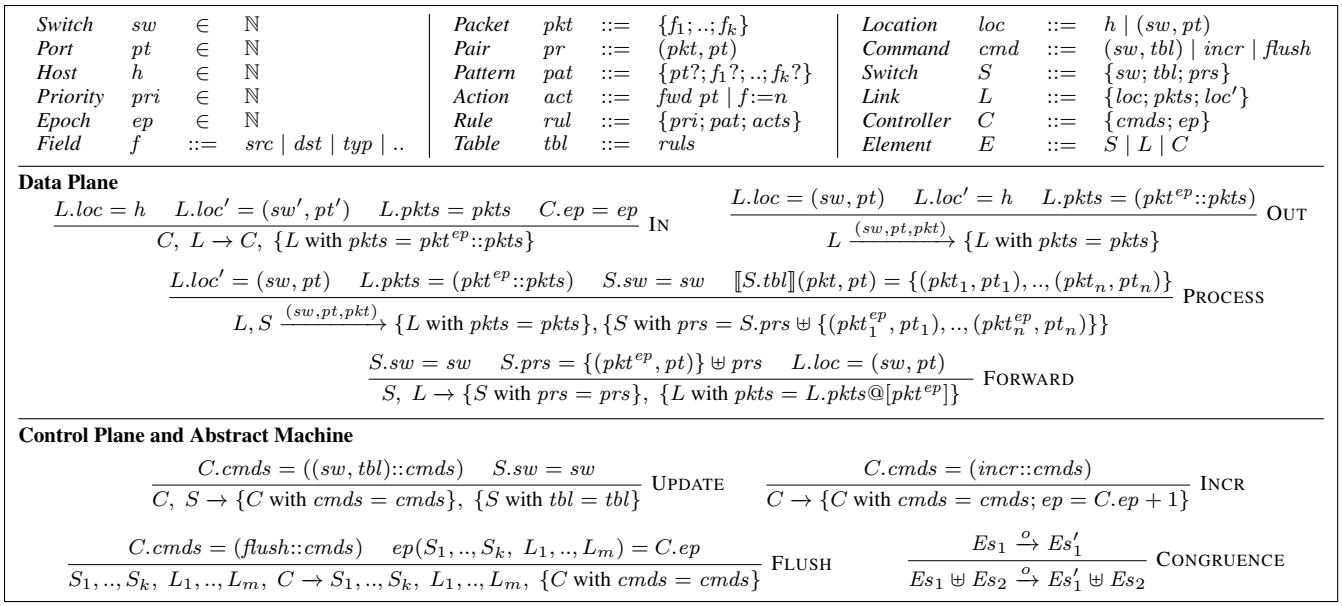

Figure 3: Network model.

We introduce a command $wait$ defined as $incr; flush$. The epochs are included in our model solely to enable reasoning. They do not need to be implemented in a real network—all that is needed is a mechanism for blocking the controller to allow a flush of all packets currently in the network. For example, given a topology, one could compute a conservative delay based on the maximum hop count, and then implement $wait$ by sleeping, rather than synchronizing with each switch. Note that we implicitly assume failure-freedom and packet-forwarding fairness of switches and links, i.e. there is an upper bound on each element's packet-processing time.

Elements. The elements E of the network model include switches S_i, links L_j, and a single controller element C, and a *network* N is a tuple containing these. Each switch S_i is encoded as a record comprising a unique identifier sw, a table tbl of prioritized forwarding rules, and a multiset prs of pairs (pkt, pt) of buffered packets and the ports they should be forwarded to respectively. Each link L_j is represented by a record consisting of two *locations* loc and loc' and a list of queued packets $pkts$, where a location is either a host or a switch-port pair. Finally, controller C is represented by a record containing a list of commands $cmds$ and an epoch ep. We assume that commands are totally-ordered. The controller can ensure this by using OpenFlow *barrier* messages.

Operational semantics. Network behavior is defined by small-step operational rules in Figure 3. These define interactions between subsets of elements, based on OpenFlow semantics [28]. States of the model are given by multisets of elements. We write $\{x\}$ to denote a singleton multiset, and $m_1 \uplus m_2$ for the union of multisets m_1 and m_2. We write $[x]$ for a singleton list, and $l_1 @ l_2$ for concatenation of l_1 and l_2. Each transition $N \xrightarrow{o} N'$ is annotated, with o being either an empty annotation, or an *observation* (sw, pt, pkt) indicating the location and packet being processed.

The first rules describe date-plane behavior. The IN rule admits arbitrary packets into the network from a host, stamping them with the current controller epoch. The OUT rule removes a packet buffered on a link adjacent to a host. PROCESS processes a single packet on a switch, finding the highest priority rule with matching pattern, applying the actions of that rule to generate a multiset of packets, and adding those packets to the output buffer. FORWARD moves a packet from a switch to the adjacent link. The final rules describe control-plane behavior. UPDATE replaces the table on a single switch. INCR increments the epoch on the controller, and FLUSH blocks the controller until all packets in the network are annotated with *at least* the current epoch ($ep(Es)$ denotes the smallest annotation on any packet in Es). Finally, CONGRUENCE, allows any sub-collection of network elements to interact.

3.2 Network Update Problem

In order to define the network update problem, we need to first define *traces* of packets flowing through the network.

Packet traces. Given a network N, our operational rules can generate sequences of observations. However, the network can process many packets concurrently, and we want observations generated by a single packet. We define a successor relation $\sqsubseteq$ for observations (Definition 7, Appendix A). Intuitively $o \overset{ep}{\sqsubseteq} o'$ if the network can directly produce the packet in o' by processing o in the epoch ep.

Definition 1 (Single-Packet Trace)**.** *Let N be a network. A sequence $(o_1 \cdots o_l)$ is a* single-packet trace of N *if $N \xrightarrow{o'_1} \ldots \xrightarrow{o'_k} N_k$ such that $(o_1 \cdots o_l)$ is a subsequence of $(o'_1 \cdots o'_k)$ for which*

- *every observation is a successor of the preceding observation in monotonically increasing epochs, and*
- *if $o_1 = o'_j = (sw, pt, pkt)$, then $\exists o'_i \in \{o'_1, \cdots, o'_{j-1}\}$ such that the o'_i transition is an* IN *moving pkt from host to (sw, pt) and none of $o'_i, \cdots, o'_{j-1}$ is a predecessor of o_1, and*
- *the o_l transition is an* OUT *terminating at a host.*

Intuitively, single-packet traces are end-to-end paths through the network. We write $\mathcal{T}(N)$ for the set of single-packet traces generated by N. A trace $(o_1 \cdots o_k)$ is *loop-free* if $o_i \neq o_j$ for all distinct i and j between 1 and k. We consider only loop-free traces, since a network that forwards packets around a loop is generally considered to be misconfigured. In the worst case, forwarding loops can cause a packet storm, wasting bandwidth and degrading performance. Our tool automatically detects/rejects such configurations.

LTL formulas. Many important network properties can be understood by reasoning about the traces that packets can take through

the network. For example, reachability requires that all packets starting at src eventually reach dst. Temporal logics are an expressive and well-studied language for specifying such trace-based properties. Hence, we use Linear Temporal Logic (LTL) to describe traces in our network model. Let AP be atomic propositions that test the value of a switch, port, or packet field: $f_i = n$. We call elements of the set 2^{AP} *traffic classes*. Intuitively, each traffic class T identifies a set of packets that agree on the values of particular header fields. An LTL formula φ in negation normal form (NNF) is either $true$, $false$, atomic proposition p in AP, negated proposition $\neg p$, disjunction $\varphi_1 \vee \varphi_2$, conjunction $\varphi_2 \wedge \varphi_2$, next $X\varphi$, until $\varphi_1 U \varphi_2$, or release $\varphi_1 R \varphi_2$, where φ_1 and φ_2 are LTL formulas in NNF. The operators F and G can be defined using other connectives. Since (finite) single-packet traces can be viewed as infinite sequences of packet observations where the final observation repeats indefinitely, the semantics of the LTL formulas can be defined in a standard way over traces. We write $t \models \varphi$ to indicate that the single-packet trace t satisfies the formula φ and $\mathcal{T} \models \varphi$ to indicate that $t \models \varphi$ for each t in $\mathcal{T}$. Given a network N and a formula φ, we write $N \models \varphi$ if $\mathcal{T}(N) \models \varphi$.

Problem Statement. Recall that our network model includes commands for updating a single switch, incrementing the epoch, and waiting until all packets in the preceding epoch have been flushed from the network. At a high-level, our goal is to identify a sequence of commands to transition the network between configurations without violating specified invariants. First, we need a bit of notation. Given a network N, we write $N[sw \leftarrow tbl]$ for the *switch update* obtained by updating the forwarding table for switch sw to tbl. We call N *static* if $C.cmds$ is empty. If static networks N_1, N_n have the same traces $\mathcal{T}(N_1) = \mathcal{T}(N_n)$, then we say they are trace-equivalent, $N_1 \simeq N_n$.

Definition 2 (Network Update)**.** *Let N_1 be a static network. A command sequence $cmds$ induces a sequence $N_1, \ldots, N_n$ of static networks if $c_1 \cdots c_{n-1}$ are the update commands in $cmds$, and for each $c_i = (sw, tbl)$, we have $N_i[sw \leftarrow tbl] \simeq N_{i+1}$.*

We write $N_1 \xrightarrow{cmds} N_n$ if there exists such a sequence of static networks induced by $cmds$ which ends with N_n.

We call N *stable* if all packets in N are annotated with the same epoch. Intuitively, a stable network is one with no in-progress update, i.e. any preceding update command was finalized with a *wait*. Consider the set of *unconstrained* single-packet traces generated by removing the requirement that traces start at an ingress (see Definition 8, Appendix A). This includes $\mathcal{T}(N)$ as well as traces of packets initially present in N. We call this $\overline{\mathcal{T}}(N)$, and note that for a *stable* network N, $\overline{\mathcal{T}}(N)$ is equal to $\mathcal{T}(N)$.

Definition 3 (Update Correctness)**.** *Let N be a stable static network and let φ be an LTL formula. The command sequence $cmds$ is* correct *with respect to N and φ if $\hat{N} \models \phi$ where $\hat{N}$ is obtained from N by setting $C.cmds = cmds$.*

A *network configuration* is a static network which contains no packets. We can now present the problem statement.

Definition 4 (Update Synthesis Problem)**.** *Given stable static network N, network configuration N', and LTL specification φ, construct a sequence of commands $cmds$ such that (i) $N \xrightarrow{cmds} N''$ where $N'' \simeq N'$, and (ii) $cmds$ is correct with respect to φ.*

3.3 Efficiently Checking Network Properties

To facilitate efficient checking of network properties via LTL model checkers, we show how to model a network as a Kripke structure.

Kripke structures. A *Kripke structure* is a tuple $(Q, Q_0, \delta, \lambda)$, where Q is a finite set of states, $Q_0 \subseteq Q$ is a set of initial states, $\delta \subseteq Q \times Q$ is a transition relation, and $\lambda : Q \to 2^{AP}$ labels each state with a set of atomic propositions drawn from a fixed set AP. A Kripke structure is *complete* if every state has at least one successor. A state $q \in Q$ is a *sink state* if for all states q', $\delta(q, q')$ implies that $q = q'$, and we call a Kripke structure *DAG-like* if the only cycles are self-loops on sink states. In this paper, we will consider complete and DAG-like Kripke structures. A *trace* t is an infinite sequence of states, $t_0 t_1 \ldots$ such that $\forall i \geq 0 : \delta(t_i, t_{i+1})$. Given a trace t, we write t^i for the suffix of t starting at the i-th position—i.e., $t^i = t_i t_{i+1} \ldots$. Given a set of traces $\mathcal{T}$, we let $\mathcal{T}^i$ denote the set $\{t^i \mid t \in \mathcal{T}\}$. Given a state q of a Kripke structure K, let $traces_K(q)$ be the set of traces of K starting from q and $succ_K(q)$ be the set of states defined by $q' \in succ_K(q)$ if and only if $\delta(q, q')$. We will omit the subscript K when it is clear form the context. A Kripke structure $K = (Q, Q_0, \delta, \lambda)$ satisfies an LTL formula φ if for all states $q_0 \in Q_0$ we have that $traces(q_0) \models \varphi$.

Network Kripke structures. For every static N, we can generate a Kripke structure $\mathcal{K}(N)$ containing traces which correspond according to an intuitive trace relation $\lesssim$ (Definition 9, 10, Appendix A). We currently do not reason about packet modification, so the Kripke structure has disjoint parts corresponding to the traffic classes. It is straightforward to enable packet modification, by adding transitions between the parts of the Kripke structure, but we leave this for future work. We now show that the generated Kripke structure faithfully encodes the network semantics.

Lemma 1 (Network Kripke Structure Soundness)**.** *Let N be a static network and $K = \mathcal{K}(N)$ a network Kripke structure. For every single-packet trace t in $\mathcal{T}(N)$ there exists a trace t' of K from a start state such that $t \lesssim t'$, and vice versa.*

This means that checking LTL over single-packet traces can be performed via LTL model-checking of Kripke structures.

Checking network configurations. One key challenge arises because the network is a distributed system. Packets can "see" an inconsistent configuration (some switches updated, some not), and reasoning about possible interleavings of commands becomes intractable in this context. We can simplify the problem if we ensure that each packet traverses at most one switch that was updated after the packet entered the network.

Definition 5 (Careful Command Sequences)**.** *A sequence of commands $(cmd_1 \cdots cmd_n)$ is* careful *if every pair of switch updates is separated by a $wait$ command.*

In the rest of this paper, we consider careful command sequences, and develop a sound and complete algorithm that finds them efficiently. Section 4 describes a technique for removing wait commands that works well in practice, but we leave *optimal* wait removal for future work. Recall that $\mathcal{T}(N)$ denotes the sequence of all traces that a packet could take through the network, regardless of when the commands in $N.cmds$ are executed. This is a superset of the traces induced by each static N_i in a solution to the network update problem. However, if $cmds$ is careful, then each packet only encounters a single configuration, allowing the correctness of the sequence to be reduced to the correctness of each N_i.

Lemma 2 (Careful Correctness)**.** *Let N be a stable network with $C.cmds$ careful and let φ be an LTL formula. If $cmds$ is careful and $N_i \models \phi$ for each static network in any sequence induced by $cmds$, then $cmds$ is correct with respect to φ.*

In Lemmas 5 and 6 (Appendix A), we show that checking the *unique sequence of network configurations* induced by $cmds$ is equivalent to the above. Next we will develop a sound and complete algorithm that solves the update synthesis problem for careful sequences by checking configurations.

Procedure ORDERUPDATE(N_i, N_f, φ)
Input: Initial static network N_i, final static configuration N_f, formula φ.
Output: update sequence L, or error ϵ if no update sequence exists
1: $W \leftarrow false$ ▷ Formula encoding wrong configurations.
2: $V \leftarrow false$ ▷ Formula encoding visited configurations.
3: $(ok, L) \leftarrow$ DFSFORORDER($N_i, \mathcal{K}(N_i), \perp, \varphi, \lambda_0$)
4: **if** ok **then return** L
5: **else return** ϵ ▷ Failure—no update exists.

Procedure DFSFORORDER(N,K,s,φ,λ)
Input: Static network N and Kripke structure K, next switch to update s, formula φ, and labeling λ.
Output: Boolean ok if a correct update exists; correct update sequence L
6: **if** $N \models V \vee W$ **then return** $(false, [])$
7: **if** $s = \perp$ **then** $(ok, cex, \lambda) \leftarrow modelCheck(K, \varphi)$
8: **else**
9: $(N, K, S) \leftarrow swUpdate(N, s)$
10: $(ok, cex, \lambda) \leftarrow incrModelCheck(K, \varphi, S, \lambda)$
11: $V \leftarrow V \vee makeFormula(N)$
12: **if** $\neg ok$ **then**
13: $W \leftarrow W \vee makeFormula(cex)$
14: **return** $(false, [])$
15: **if** $N = N_f$ **then return** $(true, [s])$
16: **for** $s' \in possibleUpdates(N)$ **do**
17: $(ok, L) \leftarrow$ DFSFORORDER($N, K, s', \varphi, \lambda$)
18: **if** ok **then return** $(true, (upd\ s') :: wait :: L)$
19: **return** $(false, [])$

Figure 4: ORDERUPDATE Algorithm.

4. Update Synthesis Algorithm

This section presents a synthesis algorithm that searches through the space of possible solutions, using counterexamples to detect wrong configurations and exploiting several optimizations.

4.1 Algorithm Description

ORDERUPDATE (Figure 4) returns a *simple* sequence of updates (one in which each switch appears at most once), or fails if no such sequence exists. Note that we could broaden our *simple* definition, e.g. *k-simple*, where each switch appears at most k times, but we have found the above restriction to work well in practice. The core procedure is DFSFORORDER, which manages the search and invokes the model checker (we use DFS because we expect common properties/configurations to admit many update sequences). It attempts to add a switch s to the current update sequence, yielding a new network configuration. We maintain two formulas, V and W, tracking the set of configurations that have been visited so far, and the set of configurations excluded by counterexamples.

To check whether all packet traces in this configuration satisfy the LTL property φ, we use our (incremental) model checking algorithm (discussed in Section 5). First, we call a full check of the model (line 7). The model checker labels the Kripke structure nodes with information about what formulas hold for paths starting at that state. The labeling (stored in λ) is then re-used in the subsequent model checking calls for related Kripke structures (line 10). The parameters passed in the incremental model checking call are: updated Kripke structure K, specification φ, set of nodes S in K whose transition function has changed by the update of the switch s, and correct labeling λ of the Kripke structure before the update. Note that before the initial model checking, we convert the network configuration N to a Kripke structure K. The update of K is performed by a function $swUpdate$ that returns a triple (N', S, K'), where N' is the new static network, K' is the updated Kripke structure obtained as $\mathcal{K}(N')$, and S is the set of nodes that have different outgoing transitions in K'.

If the model checker returns *true*, then N is safe and the search proceeds recursively, after adding $(upd\ s')$ to the current sequence of commands. If the model checker returns *false*, the search backtracks, using the counterexample-learning approach below.

4.2 Optimizations

We now present optimizations improving synthesis (*pruning with counterexamples*, *early search termination*), and improving efficiency of synthesized updates (*wait removal*).

Counterexamples. Counterexample-based pruning learns network configurations that do not satisfy the specification to avoid making future model checking calls that are certain to fail. The function $makeFormula(cex)$ (Line 13) returns a formula representing the set of switches that occurred in the counterexample trace cex, with flags indicating whether each switch was updated. This allows equivalent future configurations to be eliminated without invoking the model checker. Recall the red-green example in Section 2 and suppose that we update A1 and then C2. At the intermediate configuration obtained by updating just A1, packets will be dropped at C2, and the specification (H1-H3 connectivity) will not be satisfied. The formula for the unsafe set of configurations that have A1 updated and C2 not updated will be added to W. In practice, many counterexamples are small compared to network size, and this greatly prunes the search space.

Early search termination. The early search termination optimization speeds up termination of the search when no (switch-granularity) update sequence is possible. Recall how we use counterexamples to prune *configurations*. With similar reasoning, we can use counterexamples for pruning possible *sequences of updates*. Consider a counterexample trace which involves three nodes A, B, C, with A updated, B updated, and C not updated. This can be seen as requiring that C must be updated before A, or C must be updated before B. Early search termination involves collecting such constraints on possible updates, and terminating if these constraints taken together form a contradiction. In our tool, this is done efficiently using an (incremental) SAT solver. If the solver determines that no update sequence is possible, the search terminates. For simplicity, early search termination is not shown in Figure 4.

Wait removal. This heuristic eliminates waits that are unnecessary for correctness. Consider an update sequence $L = cmd_0\, cmd_1 \cdots cmd_n$, and consider some switch update $cmd_k = (upd\ s)$. In the configuration resulting from executing the sequence $cmd_0\, cmd_1 \cdots cmd_{k-1}$, if the switch s cannot possibly receive a packet which passed through some switch s_0 before an update $cmd_j{=}(upd\ s_0)$ where $j < k$, then we can update s without waiting. Thus, we can remove some unnecessary waits if we can maintain reachability-between-switches information during the update. Wait removal is not shown in Figure 4, but in our tool, it operates as a post-processing pass once an update sequence is found. In practice, this removes a majority of unnecessary waits (see § 6).

4.3 Formal Properties

The following two theorems show that our algorithm is sound for careful updates, and complete if we limit our search to *simple* update sequences (see Appendix B for proofs).

Theorem 1 (Soundness)**.** *Given initial network N_i, final configuration N_f, and LTL formula φ, if* ORDERUPDATE *returns a command sequence cmds, then $N_i \xrightarrow{cmds} N'$ s.t. $N' \simeq N_f$, and cmds is correct with respect to φ and N_i.*

Theorem 2 (Completeness)**.** *Given initial network N_i, final configuration N_f, and specification φ, if there exists a simple, careful sequence cmds with $N_i \xrightarrow{cmds} N'$ s.t. $N' \simeq N_f$, then* ORDERUPDATE *returns one such sequence.*

5. Incremental Model Checking

We now present an incremental algorithm for model checking Kripke structures. This algorithm is central to our synthesis tool, which invokes the model checker on many closely related structures. The algorithm makes use of the fact that the only cycles in the Kripke structure are self-loops on sink nodes—something that is true of structures encoding loop-free network configurations—and re-labels the states of a previously-labeled Kripke structure with the (possibly different) formulas that hold after an update.

5.1 State Labeling

We begin with an algorithm for labeling states of a Kripke structure with sets of formulas, following the approach of [39] (WVS) and [37]. The WVS algorithm translates an LTL formula φ into a local automaton and an eventuality automaton. The local automaton checks consistency between a state and its predecessor, and handles labeling of all formulas except $\varphi_1 \, U \, \varphi_2$, which is checked by the eventuality automaton. The two automata are composed into a single Büchi automaton whose states correspond to subsets of the set of subformulas of φ and their negations. Hence, we label each Kripke state by a set L of sets of formulas such that if a state q is labeled by L, then for each set of formulas S in L, there exists a trace t starting from q satisfying all the formulas in S.

We now describe state labeling precisely. Let φ be an LTL formula in NNF. The *extended closure* of φ, written $ecl(\varphi)$, is the set of all subformulas of φ and their negations:

- $true \in ecl(\varphi)$
- $\varphi \in ecl(\varphi)$
- If $\psi \in ecl(\varphi)$, then $\neg\psi \in ecl(\varphi)$ (we identify ψ with $\neg\neg\psi$, for all ψ).
- If $\varphi_1 \vee \varphi_2 \in ecl(\varphi)$, then $\varphi_1 \in ecl(\varphi)$ and $\varphi_2 \in ecl(\varphi)$.
- If $\varphi_1 \wedge \varphi_2 \in ecl(\varphi)$, then $\varphi_1 \in ecl(\varphi)$ and $\varphi_2 \in ecl(\varphi)$.
- If $X \, \varphi_1 \in ecl(\varphi)$, then $\varphi_1 \in ecl(\varphi)$.
- If $\varphi_1 \, U \, \varphi_2 \in ecl(\varphi)$, then $\varphi_1 \in ecl(\varphi)$ and $\varphi_2 \in ecl(\varphi)$
- If $\varphi_1 \, R \, \varphi_2 \in ecl(\varphi)$, then $\varphi_1 \in ecl(\varphi)$ and $\varphi_2 \in ecl(\varphi)$.

A subset $M \subset ecl(\varphi)$ of the extended closure is said to be *maximally consistent* if it contains $true$ and is simultaneously closed and consistent under boolean operations:

- $true \in M$
- $\psi \in M$ iff $\neg\psi \notin M$ (we identify ψ with $\neg\neg\psi$, for all ψ)
- $\varphi_1 \vee \varphi_2 \in M$ iff ($\varphi_1 \in M$ or $\varphi_2 \in M$)
- $\varphi_1 \wedge \varphi_2 \in M$ iff ($\varphi_1 \in M$ and $\varphi_2 \in M$)

Likewise, the relation $follows(M_1, M_2)$ captures the notion of successor induced by LTL's temporal operators, lifted to maximally-consistent sets. We say $follows(M_1, M_2)$ holds if and only if all of the following hold:

- $X \, \varphi_1 \in M_1$ iff $\varphi_1 \in M_2$
- $\varphi_1 \, U \, \varphi_2 \in M_1$ iff $\big(\varphi_2 \in M_1 \vee (\varphi_1 \in M_1 \wedge \varphi_1 \, U \, \varphi_2 \in M_2)\big)$
- $\varphi_1 \, R \, \varphi_2 \in M_1$ iff $\big(\varphi_1 \in M_1 \vee (\varphi_2 \in M_1 \wedge \varphi_1 \, R \, \varphi_2 \in M_2)\big)$

Given a trace t and a maximally-consistent set M, we write $t \models M$ if and only if for all $\psi \in M$, we have $t \models \psi$.

For the rest of this section, we fix a Kripke structure $K = (Q, Q_0, \delta, \lambda)$, a state q in Q, an LTL formula φ in NNF, and a maximally-consistent set $M \subset ecl(\varphi)$.

To compute the label of a state q, there are two cases depending on whether it is a sink state or a non-sink state. If q is a sink state, the function $HoldsSink(q, M)$ computes a predicate that is true if and only if, for all $\psi \in M$ and the unique trace t starting from q, we have $t \models \psi$. More formally, $HoldsSink(q, M)$ is defined to be $(\forall \psi \in M : Holds_0(q, \psi))$, where $Holds_0$ is defined as in Figure 5. The function $Holds_0$ computes a predicate that is true if and only if ψ holds at q. For example, $Holds_0(q, \phi_1 \, \mathrm{U} \, \phi_2)$ is defined as $Holds_0(q, \phi_2)$ because the only transition from q is a self-loop.

$$
\begin{array}{rcl}
Holds_0(q, p) & = & q \models p \\
Holds_0(q, \neg p) & = & q \not\models p \\
Holds_0(q, \phi_1 \wedge \phi_2) & = & Holds_0(q, \phi_1) \wedge Holds_0(q, \phi_2) \\
Holds_0(q, \phi_1 \vee \phi_2) & = & Holds_0(q, \phi_1) \vee Holds_0(q, \phi_2) \\
Holds_0(q, \mathrm{X}\phi) & = & Holds_0(q, \phi) \\
Holds_0(q, \phi_1 \, \mathrm{U} \, \phi_2) & = & Holds_0(q, \phi_2) \\
Holds_0(q, \phi_1 \, \mathrm{R} \, \phi_2) & = & Holds_0(q, \phi_1) \vee Holds_0(q, \phi_2)
\end{array}
$$

Figure 5: The $Holds_0$ function

For the second case, suppose q is a non-sink state. If we are given a labeling for $succ_K(q)$ (the successors of the node q), we can extend it to a labeling for q. Let $V \subseteq Q$ be a set of vertices. A function $labGr_K$ is a *correct labeling of K with respect to φ and V* if for every $v \in V$, it returns a set L of maximally consistent sets such that (a) $M \in L$ if and only if $M \subset ecl(\varphi)$, and (b) there exists a trace t in $traces(v)$ such that $t \models M$. Suppose that $labGr_K$ is a correct labeling of K with respect to φ and $succ_K(q)$. The function $Holds_K(q, M, labGr_K)$ computes a predicate that is true if and only if there exists a trace t in $traces_K(q)$ with $t \models M$. Formally, $Holds_K(q, M, labGr_K)$ is defined as $(\lambda(q) = (AP \cap M)) \wedge \exists q' \in succ_K(q), M' \in labGr_K(q') : follows(M, M')$.

The following captures the correctness of labeling:

Lemma 3. *First, $HoldsSink(q, M) \Leftrightarrow \exists t \in traces(q) : t \models M$ for sink states q. Second, if $labGr_K$ is a correct labeling with respect to φ and $succ_K(q)$, then $Holds_K(q, M, labGr_K) \iff \exists t \in traces_K(q) : t \models M$.*

Finally, we define $labelNode_K(\varphi, q, labGr_K)$, which computes a label L for q such that $M \in L$ if and only if there exists a trace $t \in traces_K(q)$ such that $t \models M$ for all $M \subset ecl(\varphi)$. We assume that $labGr_K$ is a correct labeling of K with respect to φ and $succ(q)$. For sink states, $labelNode_K(\varphi, q, labGr_K)$ returns $\{M \mid M \in ecl(\varphi) \wedge HoldsSink(q, M)\}$, while for non-sink states it returns $\{M \mid M \in ecl(\varphi) \wedge Holds_K(q, M, labGr_K)\}$.

5.2 Incremental Algorithm

To incrementally model check a modified Kripke structure, we must re-label its states with the formulas that hold after the update.

Consider two Kripke structures $K = (Q, Q_0, \delta, \lambda)$ and $K' = (Q', Q_0', \delta', \lambda')$, such that $Q_0 = Q_0'$. Furthermore, assume that $Q = Q'$, and there is a set $U \subseteq Q$ such that δ and δ' differ only on nodes in U. We call such a triple (K, K', U) an *update* of K.

An update (K, K', U) might add or remove edges connected to a (small) set of nodes, corresponding to a change in the rules on a switch. Suppose that $labGr_K$ is a correct labeling of K with respect to φ and Q. The incremental model checking problem is defined as follows: we are given an update (K, K', U), and $labGr_K$, and we want to know whether K' satisfies φ. The naïve approach is to model check K' without using the labeling $labGr_K$. We call this the *monolithic* approach. In contrast, the *incremental* approach uses $labGr_K$ (and thus intuitively re-uses the results of model checking K to efficiently verify K').

Example. Consider the left side of Figure 6, with H the only initial state. Suppose that the update modifies J, and the δ' relation applied to J only contains the pair (J, N), and consider labeling the structure with formulas $F \, a$, $F \, b$, and $F \, a \vee F \, b$. To simplify the example, we label a node by all those formulas which hold for at least one path starting from the node (note that in the algorithm, a node is labeled by a set of sets of formulas, rather than a set of formulas). We will have that all the nodes are labeled by $F \, a \vee F \, b$, and in addition the nodes K, I, H, M, J contain label $F \, a$, and the nodes L, I, H, N contain $F \, b$. Now we want to relabel the structure after the update (right-hand side). Given that the update changes only node J, the labeling can only change for J and its ancestors.

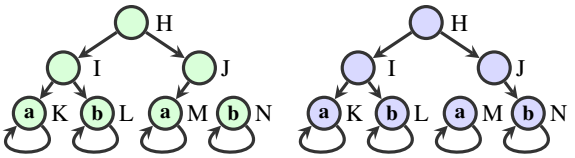

Figure 6: Incremental labeling—Initial (left), Final (right)

We therefore start labeling node J, and find that it will now be labeled with $F\ b$ instead of $F\ a$. Labeling proceeds to H, whose label does not change (still labeled by all of $F\ a$, $F\ b$, $F\ a \vee F\ b$). The labeling process could then stop, even if H has ancestors.

Re-labeling states. Let $ancestors_K(V)$ be the ancestors of V in K—i.e., a set of vertices s.t. $ancestors_K(V) \subseteq Q$ and $q \in ancestors_K(V)$, if some node $v \in V$ is reachable from q. To define incremental model checking for φ, we need a function accepting a property φ, set of vertices V, labeling $labGr_K$ that is correct for K with respect to φ and $Q \setminus ancestors_K(V)$, and returns a correct labeling of K with respect to φ and Q. This function is:

$$relbl_K(\varphi, labGr_K, V) = \begin{cases} labGr_K & \text{if } V = \emptyset \\ relbl_K(\varphi, labGr'_K, V') & \text{otherwise} \end{cases}$$

where $labGr'_K(v)$ is $labelNode_K(\varphi, v, labGr_K)$ if $v \in V$, and it is $labGr_K(v)$ if $v \notin V$. The set V' is $\{q \mid \exists v \in V : v \in succ_K(q)\}$.

Theorem 3. *Let $V \subseteq Q$ be a set of vertices and $labGr_K$ a correct labeling with respect to φ and $Q \setminus ancestors_K(V)$. Then $relbl_K(\varphi, labGr_K, V)$ is a correct labeling w.r.t. φ and Q.*

Given a labeling that is correct with respect to φ and Q, it is easy to check whether φ is true for all the traces starting in the initial states: the predicate $checkInitStates_K(labGr_K, \varphi)$ is defined as $\forall q_0 \in Q_0, M \in labGr_K(q_0) : \varphi \in M$. Next, let Q_f be the set of all sink states of K. Then $ancestors_K(Q_f)$ is the set Q of all states K. Therefore, for any initial labeling $labGr^0_K$, $relbl(\varphi, labGr^0_K, Q_f)$ is a correct labeling with respect to φ and Q. The function $modelCheck_K(\varphi)$ is defined to be equal to $checkInitStates_K(relbl_K(\varphi, labGr^0_K, Q_f), \varphi)$, where we can set $labGr^0_K$ to be the empty labeling $\lambda v.\emptyset$.

We now define our incremental model checking function. Let (K, K', U) be an update, and $labGr_K$ a previously-computed correct labeling of K with respect to φ and Q, where Q is the set of states of K. The function $incrModelCheck(K, \varphi, U, labGr_K)$ is defined as $checkInitStates_{K'}(relbl_{K'}(\varphi, labGr_K, U), \varphi)$. The following shows the correctness of our model checking functions (proof of this and the previous theorem are in Appendix C).

Corollary 1. *First, $modelCheck_K(\varphi) = true \iff K \models \varphi$. Second, for (K, K', U) and $labGr_K$ as above, we have $incrModelCheck(K, \varphi, U, labGr_K) = true \iff K \models \varphi$.*

The runtime complexity of the $modelCheck_K$ function is $O(|K| \times 2^{|\varphi|})$. The runtime complexity of the $incrModelCheck$ function is $O(|ancestors_K(U)| \times 2^{|\varphi|})$, where U is the set of nodes being updated.

Counterexamples. This incremental algorithm can generate counterexamples in cases where the formula does not hold. A formula $\neg\varphi$ does not hold if an initial state is labeled by L, such that there exists a set $M \in L$, such that $\neg\varphi \in M$. Examining the definition of $labelNode_K$, we find that in order to add a set M to the label L of a node q, there is a set M' in the label of one a child q' of q that explains why M is in L. The first node of the counterexample trace starting from q is one such child q'.

6. Implementation and Experiments

We have built a prototype tool that implements the algorithms described in this paper. It consists of 7K lines of OCaml code. The system works by building a Kripke structure (§3) and then repeatedly interacting with a model checker to synthesize an update. We currently provide four checker backends: *Incremental* uses incremental relabeling to check and recheck formulas, *Batch* re-labels the entire graph on each call, *NuSMV* queries a state-of-the-art symbolic model checker in batch mode, and *NetPlumber* queries an incremental network model checker [19]. All tools except NetPlumber provide counterexample traces, so our system learns from counterexamples whenever possible (§4).

Experiments. To evaluate performance, we generated configurations for a variety of real-world topologies and ran experiments in which we measured the amount of time needed to synthesize an update (or discover that no order update exists). These experiments were designed to answer two key questions: (1) how the performance of our Incremental checker compares to state-of-the-art tools (NuSMV and NetPlumber), and (2) whether our synthesizer scales to large topologies. We used the *Topology Zoo* [21] dataset, which consists of 261 actual wide-area topologies, as well as synthetically constructed *Small-World* [29] and *FatTree* [1] topologies. We ran the experiments on a 64-bit Ubuntu machine with 20GB RAM and a quad-core Intel i5-4570 CPU (3.2 GHz) and imposed a 10-minute timeout for each run. We ignored runs in which the solver died due to an out-of-memory error or timeout—these are infrequent (less than 8% of the 996 runs for Figure 7), and our Incremental solver only died in instances where other solvers did too.

Configurations and properties. A recent paper [23] surveyed data-center operators to discover common update scenarios, which mostly involve taking switches on/off-line and migrating traffic between switches/hosts. We designed experiments around a similar scenario. To create configurations, we connected random pairs of nodes (s, d) via disjoint initial/final paths W_i, W_f, forming a "diamond", and asserted one of the following properties for each pair:

- *Reachability:* traffic from a given source must reach a certain destination: $(\text{port} = s) \Rightarrow \text{F}\,(\text{port} = d)$
- *Waypointing:* traffic must traverse a waypoint w: $(\text{port}{=}s) \Rightarrow \big((\text{port}{\neq}d)\,\text{U}\,((\text{port}{=}w) \wedge \text{F}\,(port{=}d))\big)$
- *Service chaining:* traffic must waypoint through several intermediate nodes: $(\text{port} = s) \Rightarrow way(W, d)$, where

$$\begin{aligned} way([\,], d) &\equiv \text{F}\,(\text{port} = d) \\ way(w_i :: W, d) &\equiv \big((\textstyle\bigwedge_{w_k \in W} \text{port}{\neq}w_k \wedge \text{port}{\neq}d) \\ &\quad \text{U}\,((\text{port} = w_i) \wedge way(W, d))\big). \end{aligned}$$

Incremental vs. NuSMV/Batch. Figure 7 (a-c) compares the performance of Incremental and NuSMV backends for the reachability property. Of the 247 Topology Zoo inputs that completed successfully, our tool solved all of them faster. The measured speedups were large, with a geometric mean of 447.23x. For the 24 FatTree examples, the mean speedup was 465.03x, and for the 25 Small-World examples, the mean speedup was 4484.73x. We also compared the Incremental and Batch solvers on the same inputs. Incremental performs better on almost all examples, with mean speedup of 4.26x, 5.27x, 11.74x on the datasets shown in Figure 7(a-c) and maximum runtimes of 0.36s, 2.80s, and 0.92s respectively. The maximum runtimes for Batch were 6.71s, 39.75s, and 12.50s.

Incremental vs. NetPlumber. We also measured the performance of Incremental versus the network property checker NetPlumber (Figure 7(d-f)). Note that NetPlumber uses rule-granularity for updates, so we enabled this mode in our tool for these experiments. For the three datasets, our checker is faster on all experiments, with mean speedups of (6.41x, 4.90x, 17.19x). NetPlumber does not

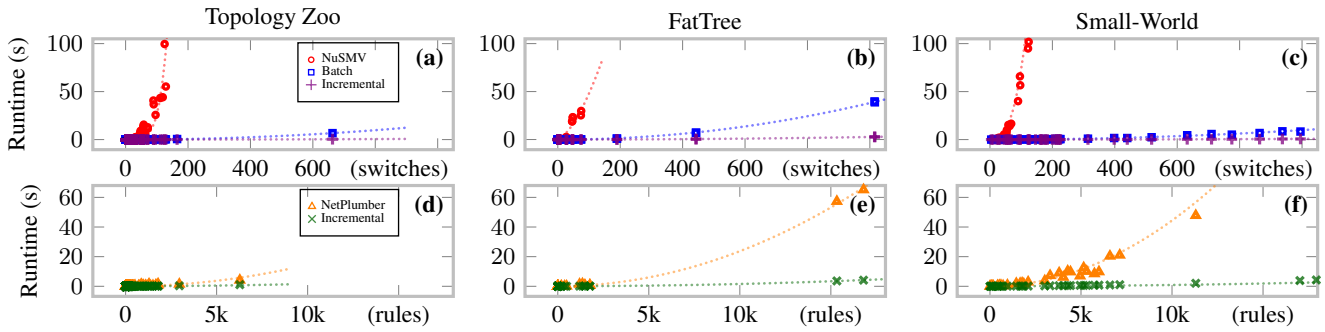

Figure 7: Relative performance results: **(a-c)** Performance of Incremental vs. NuSMV, Batch, NetPlumber solvers on Topology Zoo, FatTree, Small-World topologies (columns); **(d-f)** Performance of Incremental vs. NetPlumber (rule-granularity).

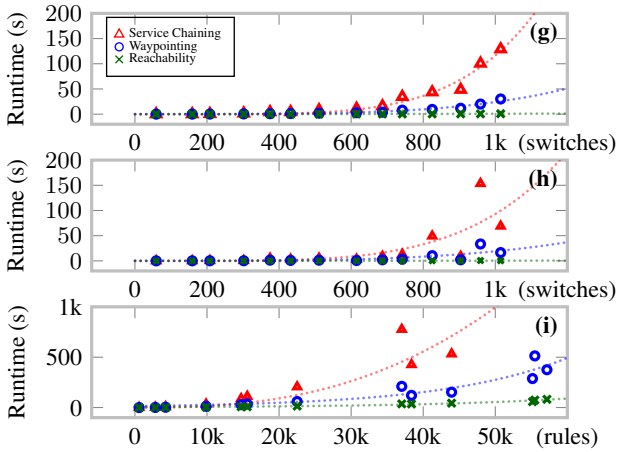

Figure 8: **(g)** Scalability of Incremental on Small-World topologies of increasing size; **(h)** Scalability when no correct switch-granularity update exists (i.e. algorithm reports "impossible"), and **(i)** Scalability of fine-grained (rule-granularity) approach for solving switch-impossible examples in (h).

report counterexamples, putting it at a disadvantage in this end-to-end comparison, so we also measured total Incremental versus NetPlumber runtime on the same set of model-checking questions posed by Incremental for the Small-World example. Our tool is still faster on all instances, with a mean speedup of 2.74x.

Scalability. To quantify our tool's scalability, we constructed Small World topologies with up to 1500 switches, and ran experiments with large diamond updates—the largest has 1015 switches updating. The results appear in Figure 8(g). The maximum synthesis times for the three properties were 129.04s, 30.11s, and 0.85s, which shows that our tool scales to problems of realistic size.

Infeasible Updates. We also considered examples for which there is no switch-granular update. Figure 8(h) shows the results of experiments where we generated a second diamond atop the first one, requiring it to route traffic in the opposite direction. Using switch-granularity, the inputs are reported as unsolvable in maximum time 153.48s, 33.48s, and 0.69s. Using rule-granularity, these inputs are solved successfully for up to 1000 switches with maximum times of 776.13s, 512.84s, and 82.00s (see Figure 8(i)).

Waits. We also separately measured the time needed to run the wait-removal heuristic for the Figure 8 experiments. For (g), the maximum wait-removal runtime was 0.89s, resulting in 2 needed waits for each instance. For (i), the maximum wait-removal runtime was 103.87s, resulting in about 2.6 waits on average (with a maximum of 4). For the largest problems in (g) and (i), this corresponds to removal of 1397/1399 and 55823/55826 waits (about 99.9%).

7. Related Work

This paper extends preliminary work reported in a workshop paper [30]. We present a more precise and realistic network model, and replace expensive calls to an external model checker with calls to a new built-in *incremental* network model checker. We extend the DFS search procedure with optimizations and heuristics that improve performance dramatically. Finally, we evaluate our tool on a comprehensive set of benchmarks with real-world topologies.

Synthesis of concurrent programs. There is much previous work on synthesis for concurrent programs [12, 35, 38]. In particular, work by Solar-Lezama et al. [35] and Vechev et al. [38] synthesizes sequences of instructions. However, traditional synthesis and synthesis for networking are quite different. First, traditional synthesis is a game against the environment which (in the concurrent programming case) provides inputs and schedules threads; in contrast, our synthesis problem involves reachability on the space of configurations. Second, our space of configurations is very rich, meaning that checking configurations is itself a model checking problem.

Network updates. There are many protocol- and property-specific algorithms for implementing network updates, e.g. avoiding packet/bandwidth loss during planned maintenance to BGP [10, 32]. Other work avoids routing loops and blackholes during IGP migration [36]. Work on network updates in SDN proposed the notion of *consistent updates* and several implementation mechanisms, including two-phase updates [33]. Other work explores propagating updates incrementally, reducing the space overhead on switches [17]. As mentioned in Section 2, recent work proposes ordering updates for specific properties [15], whereas we can handle combinations and variants of these properties. Furthermore, SWAN and zUpdate add support for bandwidth guarantees [13, 23]. Zhou et al. [40] consider customizable trace properties, and propose a dynamic algorithm to find order updates. This solution can take into account unpredictable delays caused by switch updates. However, it may not always find a solution, even if one exists. In contrast, we obtain a completeness guarantee for our static algorithm. Ludwig et al. [24] consider ordering updates for waypointing properties.

Model checking. Model checking has been used for network verification [2, 18, 20, 26, 27]. The closest to our work is the incremental checker NetPlumber [19]. Surface-level differences include the specification languages (LTL vs. regular expressions), and NetPlumber's lack of counterexample output. The main difference is incrementality: Netplumber restricts checking to "probe nodes," keeping track of "header-space" reachability information for those nodes, and then performing property queries based on this. In contrast, we look at the *property*, keeping track of *portions of the property* holding at each node, which keeps incremental recheck-

ing times low. The empirical comparison (Section 6) showed better performance of our tool as a back-end for synthesis.

Incremental model checking has been studied previously, with [34] presenting the first incremental model checking algorithm, for alternation-free μ-calculus. We consider LTL properties and specialize our algorithm to exploit the no-forwarding-loops assumption. The paper [7] introduced an incremental algorithm, but it is specific to the type of partial results produced by IC3 [5].

8. Conclusion

We present a practical tool for automatically synthesizing correct network update sequences from formal specifications. We discuss an efficient incremental model checker that performs orders of magnitude better than state-of-the-art monolithic tools. Experiments on real-world topologies demonstrate the effectiveness of our approach for synthesis. In future work, we plan to explore both extensions to deal with network failures and bandwidth constraints, and deeper foundations of techniques for network updates.

Acknowledgments

The authors would like to thank the PLDI reviewers and AEC members for their insightful feedback on the paper and artifact, as well as Xin Jin, Dexter Kozen, Mark Reitblatt, and Jennifer Rexford for helpful comments. Andrew Noyes and Todd Warszawski contributed a number of early ideas through an undergraduate research project. This work was supported by the NSF under awards CCF-1421752, CCF-1422046, CCF-1253165, CNS-1413972, CCF-1444781, and CNS-1111698; the ONR under Award N00014-12-1-0757; and gifts from Fujitsu Labs and Intel.

References

[1] M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, 2008.

[2] E. Al-Shaer and S. Al-Haj. FlowChecker: Configuration Analysis and Verification of Federated OpenFlow Infrastructures. In *SafeConfig*, 2010.

[3] M. Alizadeh, A. Greenberg, D. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan. Data Center TCP (DCTCP). In *SIGCOMM*, pages 63–74, 2010.

[4] G. Berry and G. Boudol. The Chemical Abstract Machine. In *POPL*, pages 81–94, 1990.

[5] A. Bradley. SAT-Based Model Checking without Unrolling. In *VMCAI*, 2011.

[6] M. Casado, N. Foster, and A. Guha. Abstractions for Software-Defined Networks . *CACM*, 57(10):86–95, Oct. 2014.

[7] H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. Incremental Formal Verification of Hardware. In *FMCAD*, pages 135–143, 2011.

[8] N. Foster, R. Harrison, M. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A Network Programming Language. In *ICFP*, pages 279–291, 2011.

[9] P. Francois and O. Bonaventure. Avoiding Transient Loops during the Convergence of Link-state Routing Protocols. *IEEE/ACM Transactions on Networking*, 15(6):1280–1292, 2007.

[10] P. Francois, O. Bonaventure, B. Decraene, and P.-A. Coste. Avoiding Disruptions during Maintenance Operations on BGP Sessions. *IEEE Transactions on Network and Service Management*, 4(3):1–11, 2007.

[11] A. Guha, M. Reitblatt, and N. Foster. Machine-Verified Network Controllers . In *PLDI*, June 2013.

[12] P. Hawkins, A. Aiken, K. Fisher, M. Rinard, and M. Sagiv. Concurrent Data Representation Synthesis. In *PLDI*, pages 417–428, June 2012.

[13] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer. Achieving High Utilization with Software-Driven WAN. In *SIGCOMM*, pages 15–26, Aug. 2012.

[14] 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. In *SIGCOMM*, 2013.

[15] X. Jin, H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic Scheduling of Network Updates. In *SIGCOMM*, pages 539–550, 2014.

[16] J. P. John, E. Katz-Bassett, A. Krishnamurthy, T. Anderson, and A. Venkataramani. Consensus Routing: The Internet as a Distributed System. In *NSDI*, pages 351–364, 2008.

[17] N. P. Katta, J. Rexford, and D. Walker. Incremental Consistent Updates. In *HotSDN*, pages 49–54. ACM, 2013.

[18] P. Kazemian, G. Varghese, and N. McKeown. Header Space Analysis: Static Checking for Networks. In *NSDI*, 2012.

[19] P. Kazemian, M. Chang, H. Zeng, G. Varghese, N. McKeown, and S. Whyte. Real Time Network Policy Checking Using Header Space Analysis. NSDI, pages 99–112, 2013.

[20] A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying Network-wide Invariants in Real Time. *ACM SIGCOMM CCR*, 2012.

[21] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications*, 29(9):1765–1775, Oct. 2011.

[22] A. Lazaris, D. Tahara, X. Huang, L. Li, A. Voellmy, Y. Yang, and M. Yu. Tango: Simplifying SDN Programming with Automatic Switch Behavior Inference, Abstraction, and Optimization. 2014.

[23] H. H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz. zUpdate: Updating Data Center Networks with Zero Loss. In *SIGCOMM*, pages 411–422. ACM, 2013.

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

[25] R. Mahajan and R. Wattenhofer. On Consistent Updates in Software Defined Networks. In *SIGCOMM*, Nov. 2013.

[26] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the Data Plane with Anteater. In *SIGCOMM*, 2011.

[27] R. Majumdar, S. Tetali, and Z. Wang. Kuai: A Model Checker for Software-defined Networks. In *FMCAD*, 2014.

[28] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. OpenFlow: Enabling Innovation in Campus Networks. *ACM SIGCOMM CCR*, 2008.

[29] M. E. Newman, S. H. Strogatz, and D. J. Watts. Random Graphs with Arbitrary Degree Distributions and their Applications. 2001.

[30] A. Noyes, T. Warszawski, and N. Foster. Toward Synthesis of Network Updates. In *SYNT*, July 2013.

[31] Open Networking Foundation. OpenFlow 1.4 Specification, 2013.

[32] S. Raza, Y. Zhu, and C.-N. Chuah. Graceful Network State Migrations. *IEEE/ACM Transactions on Networking*, 19(4):1097–1110, 2011.

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

[34] O. Sokolsky and S. Smolka. Incremental Model Checking in the Modal Mu-Calculus. In *CAV*, pages 351–363, 1994.

[35] A. Solar-Lezama, C. G. Jones, and R. Bodik. Sketching Concurrent Data Structures. In *PLDI*, pages 136–148, 2008.

[36] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure. Seamless Network-wide IGP Migrations. In *SIGCOMM*, 2011.

[37] M. Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification (Preliminary Report). In *LICS*, 1986.

[38] M. Vechev, E. Yahav, and G. Yorsh. Abstraction-guided Synthesis of Synchronization. In *POPL*, pages 327–338, 2010.

[39] P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about Infinite Computation Paths (Extended Abstract). In *FOCS*, 1983.

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

A. Network Model Auxiliary Definitions

We first define what it means for a table to be active, i.e. the controller contains an update that will eventually produce that table.

Definition 6 (Active Forwarding Table)**.** *Let N be a network. The forwarding table tbl is* active in the epoch ep *for the switch sw if*

1. *$ep = 0$ and tbl is the initial table of sw in N, or*
2. *$ep > 0$ and either (a) if there exists a command $(sw', tbl') \in C.cmds$ such that $sw = sw'$ and the number of wait commands preceding (sw, tbl) in $C.cmds$ is ep, then $tbl = tbl'$, or (b) if there does not exist such a command, then tbl is the table active for the switch sw in epoch $ep - 1$.*

Next we define what it means for an observation o' to succeed o.

Definition 7 (Successor Observation)**.** *Let N be a network and let $o = (sw, pt, pkt)$ and $o' = (sw', pt', pkt')$ be observations. The observation o' is a* successor of o in ep*, written $o \overset{ep}{\sqsubseteq} o'$, if either:*

- *there exists a switch S_i and link L_j such that $S_i.sw = sw$ and $S_i.tbl$ is active in ep and $L_j.loc = (sw, pt_j)$ and $L_j.loc' = (sw', pt')$ and $(pt_j, pkt') \in [\![S_i.tbl]\!](pt, pkt)$, or*
- *there exists a switch S_i, a link L_j, and a host h such that $S_i.sw = sw$ and $S_i.tbl$ is active in ep and $L_j.loc = (sw, pt')$ and $L_j.loc' = h$ and $(pt', pkt') \in [\![S_i.tbl]\!](pt, pkt)$.*

Intuitively $o \overset{ep}{\sqsubseteq} o'$ if the packet in o could have directly produced the packet in o' in ep by being processed on some switch. The two cases correspond to an internal and egress processing steps.

Definition 8 (Unconstrained Single-Packet Trace)**.** *Let N be a network. The sequence $(o_1 \cdots o_l)$ is a* unconstrained single-packet trace of N *if $N \xrightarrow{o'_1} \ldots \xrightarrow{o'_k} N_k$ such that $(o_1 \cdots o_l)$ is a subsequence of $(o'_1 \cdots o'_k)$ for which*

- *every observation is a successor of the preceding observation in monotonically increasing epochs, and*
- *if $o_1 = o'_j = (sw, pt, pkt)$, i.e. $N \xrightarrow{o'_1} \ldots \xrightarrow{o'_j = o_1} N_j \xrightarrow{o'_{j+1}} \ldots \xrightarrow{o'_k} N_k$, then no $o'_i \in \{o'_1, \cdots, o'_{j-1}\}$ precedes o_1, and*
- *the o_l transition is an* OUT *terminating at a host.*

Unconstrained single-packet traces are not required to begin at a host. We write $\bar{\mathcal{T}}(N)$ for the set of unconstrained single-packet traces generated by N, and note that $\mathcal{T}(N) \subseteq \bar{\mathcal{T}}(N)$.

Definition 9 (Network Kripke Structure)**.** *Let N be a static network. We define a Kripke structure $\mathcal{K}(N) = (Q, Q_0, \delta, \lambda)$ as follows. The set of states Q comprises tuples of the form (sw, pt, T_k). The set Q_0 contains states (sw, pt, T_k) where sw and pt are adjacent to an ingress link—i.e., there exists a link L_j and host h such that $L_j.loc = h$ and $L_j.loc' = (sw, pt)$. Transition relation δ contains all pairs of states (sw, pt, T_k) and (sw', pt', T'_k) where there exists a switch S and a link L such that $S.sw = sw$ and either:*

- *there exists a link L_j and packets $pkt \in T_k$ and $pkt' \in T'_k$ such that $L.loc' = (sw, pt)$ and $L_j.loc = (sw, pt_j)$ and $L_j.loc' = (sw', pt')$ and $(pkt', pt_j) \in [\![S.tbl]\!](pkt, pt)$.*
- *there exists a link L_j, a host h, and packets $pkt \in T_k$ and $pkt' \in T'_k$ such that $L.loc' = (sw, pt)$ and $L_j.loc = (sw, pt')$ and $L_j.loc' = h$ and $(pkt', pt') \in [\![S.tbl]\!](pkt, pt)$.*
- *$(sw, pt, T_k) = (sw', pt', T'_k)$ and there exists a packet $pkt \in T_k$ such that $L.loc' = (sw, pt)$ and $[\![S.tbl]\!](pkt, pt) = \{\}$.*
- *$(sw, pt, T_k) = (sw', pt', T'_k)$ and there exists a link L_j and host h such that $L_j.loc = (sw, pt)$ and $L_j.loc' = h$.*

Finally, the labeling function λ maps each state (sw, pt, T_k) to T_k, which captures the set of all possible header values of packets located at switch sw and port pt.

The four cases of the δ relation correspond to forwarding packets to an internal link, forwarding packets out an egress, dropping packets on a switch, or reaching an egress (inducing a self-loop).

We can relate the observations generated by a network N and the traces of the Kripke structure generated from it.

Definition 10 (Trace Relation)**.** *Let N be a static network and K a Kripke structure. Let $\lesssim$ be a relation on observations of N and states of K defined by $(sw, pt, pkt) \lesssim (sw, pt, T_k)$ if and only if $pkt \in T_k$. Lift $\lesssim$ to a relation on (finite) sequences of observations and (infinite) traces by repeating the final observation and requiring $\lesssim$ to hold pointwise: $o_1 \cdots o_k \lesssim t$ if and only if $o_i \lesssim t_i$ for i from 1 to k and $o_k \lesssim t_j$ for all $j > k$.*

Lemma 4 (Traces of a Stable Network)**.** *Let N be a stable network. Then for each trace $t \in \bar{\mathcal{T}}(N)$, there exists a trace $t' \in \mathcal{T}(N)$ such that t is a* suffix *of t'.*

Lemma 5 (Trace-Equivalence)**.** *Let N_1, N_n be static networks where $N_1 \to \cdots \to N_n$ and no transition is an update command. For a single-packet trace t, we have $t \in \mathcal{T}(N_1) \iff t \in \mathcal{T}(N_n)$.*

Lemma 6 (Induced Sequence of Networks)**.** *Let N_1 be a static network, and let N'_1 be the network obtained by emptying all packets from N_1. Let $cmds$ be a sequence of commands, and let $c_1 \cdots c_{n-1}$ be the subsequence of update commands. Construct the sequence $N'_1 \to \cdots \to N'_n$ of empty networks by executing the update commands in order. Now, given any sequence $N_1 \to \cdots \to N_n$ induced by $cmds$, we have $N_i \simeq N'_i$ for all i.*

In other words, any induced sequence of static networks is pointwise trace-equivalent to the unique sequence of *network configurations* generated by running the update commands in order.

B. Synthesis Algorithm Correctness Proofs

Lemma 1 (Network Kripke Structure Soundness)**.** *Let N be a static network and $K = \mathcal{K}(N)$ a network Kripke structure. For every single-packet trace t in $\mathcal{T}(N)$ there exists a trace t' of K from a start state such that $t \lesssim t'$, and vice versa.*

Proof. We proceed by induction over k, the length of the (finite prefix of the) trace. The base case $k = 1$ is easy to see, since the lone observation in t must be on an ingress link, meaning the corresponding state in K will be an initial state with a self-loop (case 3 of Definition 9), and these are equivalent via Definition 10.

For the inductive step ($k > 1$), we wish to show both directions of subtrace relation $\lesssim$ to conclude equivalence. First, let $t = o_1, \cdots, o_{k+1}$ be a single-packet trace of length $k + 1$ in $\mathcal{T}(N)$, and we must show that $\exists t' \in \mathcal{K}(N)$ such that $t \lesssim t'$. Let t^k be the prefix of t having length k. By our induction hypothesis, there exists $t'^k = s_1, \cdots, s_{k-1}, s_k, s_k, \cdots \in \mathcal{K}(N)$ such that $t^k \lesssim t'^k$. We have the successor relation $o_k \sqsubseteq o_{k+1}$, so Definition 7 and 9 tells us that we have a transition $s_k \to s'$ for some $s' \in K$. We see that this s' is exactly what we need to construct $t' = s_1, \cdots, s_k, s', s', \cdots$ which satisfies the relation $t \lesssim t'$.

Now, let $t' = s_1, \cdots, s_k, s_{k+1}, s_{k+1}, \cdots$ be a trace in $\mathcal{K}(N)$ for which the finite prefix has length $k + 1$. We must show that $\exists t \in \mathcal{T}(N)$ such that $t \lesssim t'$. Let $t'^k = s_1, \cdots, s_{k-1}, s_k, s_k, \cdots$, and by our induction hypothesis, and there exists $t^k = o_1, \cdots, o_k$ such that $t^k \lesssim t'^k$. Consider transition $s_k \to s_{k+1}$. If $s_k = s_{k+1}$, then $t' = t'^k$, so we can let $t = t^k$, and conclude that $t \lesssim t'$. Otherwise, if $s_k \neq s_{k+1}$, then we have one of the first two cases in Definition 9, which correspond to the cases in Definition 7, allowing us to construct an o_{k+1} such that $o_k \sqsubseteq o_{k+1}$. We let $t = o_1, \cdots, o_k, o_{k+1}$, and conclude that $t \lesssim t'$. □

We want to develop a lemma showing that the correctness of careful command sequences can be reduced to the correctness of each induced N_i, so we start with the following auxiliary lemma:

Lemma 7 (Traces of a Careful Network)**.** *Let N be a stable network with $C.cmds$ careful, and consider a sequence of static networks induced by $C.cmds$. For every trace $t \in \mathcal{T}(N)$ there exists a stable static network N_i in the sequence s.t. $t \in \mathcal{T}(N_i)$.*

Proof. I. First, we show that at most one update transition can be involved in the trace. In other words, if $N \xrightarrow{o'_1} \dots \xrightarrow{o'_k} N_k$ where $t = o_1 \cdots o_n$ is a subsequence of $o'_1 \cdots o'_k$, and if $f : \mathbb{N} \to \mathbb{N}$ is a bijection between o_i indices and o'_i indices, then at most one of the transitions $o'_{f(1)}, \cdots, o'_{f(n)}$ is an UPDATE transition.

Assume to the contrary that there are more than one such transitions, and consider two of them, o'_i, o'_j where $i, j \in \{f(1), \cdots, f(n)\}$, assuming without loss of generality that $i < j$. Now, since the sequence $C.cmds$ is careful, we must have both an INCR and FLUSH transition between o'_i and o'_j. This means that the second update o'_j cannot happen while the trace's packet is still in the network, i.e. $j > f(n)$, and we have reached a contradiction.

II. Now, if there are zero update transitions, we are done, since the trace is contained in the first static N. If there is one update transition $N_{k+1} = N_k[sw \leftarrow tbl]$, and this update occurs before the packet reaches sw in the trace, then the trace is fully contained in N_{k+1}. Otherwise, the trace is fully contained in N_k. □

Lemma 2 (Careful Correctness)**.** *Let N be a stable network with $C.cmds$ careful and let φ be an LTL formula. If $cmds$ is careful and $N_i \models \phi$ for each static network in any sequence induced by $cmds$, then $cmds$ is correct with respect to φ.*

Proof. Consider a trace $t \in \mathcal{T}(N)$. From Lemma 7, we have $t \in \mathcal{T}(N_i)$ for some N_i in the induced sequence. Thus $t \models \varphi$, since our hypothesis tells us that $N_i \models \varphi$. Since this is true for an arbitrary trace, we have shown that $\mathcal{T}(N) \models \varphi$, i.e. $N \models \varphi$, meaning that $cmds$ is correct with respect to φ. □

Theorem 1 (Soundness)**.** *Given initial network N_i, final configuration N_f, and LTL formula φ, if* ORDERUPDATE *returns a command sequence $cmds$, then $N_i \xrightarrow{cmds} N'$ s.t. $N' \simeq N_f$, and $cmds$ is correct with respect to φ and N_i.*

Proof. It is easy to show that if ORDERUPDATE returns $cmds$, then $N_i \xrightarrow{cmds} N'$ where $N' \simeq N_f$. Each update in the returned sequence changes a switch configuration of one switch s to the configuration $N_f(s)$, and the algorithm terminates when all (and only) switches s such that $N_i(s) \neq N_f(s)$ have been updated.

Observe that if ORDERUPDATE returns $cmds$, the sequence can be made careful by choosing an adequate time delay between each update command, and for all $j \in \{0, \cdots, n\}$, $N_j \models \varphi$. This is ensured by the call to a model checker (Line 7). We use Lemma 2 to conclude that $cmds$ is correct with respect to φ and N_i. □

To show that ORDERUPDATE is complete with respect to simple and careful command sequences, we observe that ORDERUPDATE searches through all simple and careful sequences.

Theorem 2 (Completeness)**.** *Given initial network N_i, final configuration N_f, and specification φ, if there exists a simple, careful sequence $cmds$ with $N_i \xrightarrow{cmds} N'$ s.t. $N' \simeq N_f$, then* ORDERUPDATE *returns one such sequence.*

C. Incremental Checking Correctness Proofs

Lemma 3. *First, $HoldsSink(q, M) \Leftrightarrow \exists t \in traces(q) : t \models M$ for sink states q. Second, if $labGr_K$ is a correct labeling with respect to φ and $succ_K(q)$, then $Holds_K(q, M, labGr_K) \iff \exists t \in traces_K(q) : t \models M$.*

Proof. First, for sink states, observe that there is a unique trace t in $traces(q)$, as q is a sink state. We first prove that $t \models \varphi$ iff $Holds_0(q, \varphi)$. We prove this by induction on the structure of the LTL formula. Then we observe that there is a unique maximally-consistent set M such that $t \models M$. This is the set $\{\psi \mid t \models \psi \wedge \psi \in ecl(\varphi)\}$. We then use the definition of $HoldsSink(q, M)$ for sink states to conclude the proof.

Now consider non-sink states: we first prove soundness, i.e., if $Holds_K(q, M, labGr_K)$, then there exists $t \in traces(q)$ such that $t \models M$. We have $Holds_K(q, M, labGr_K)$ iff $(\lambda(q) = (AP \cap M))$ and there exists $q' \in succ_K(M)$, and $M' \in labGr_K(q')$ such that $follows(M, M')$. By assumption of the theorem, we have that if $M' \in labGr_K(q')$, then there exists a trace t' in $traces(q')$ such that $t' \models M'$. Consider a trace t such that $t_0 = q$ and $t^1 = t'$. For each $\psi \in M$, we can prove that $t \models \psi$ as follows. The base case of the proof by induction is implied by the fact that $q \models (AP \cap M)$. The inductive cases are proven using the definitions of maximally-consistent set and the function $follows$. We now prove completeness, i.e., that if there exists a trace t in $traces_K(q)$ such that $t \models M$, then $Holds_K(q, M, labGr_K)$ is true. Let t be the trace $qq_1q_2 \ldots$. It is easy to see that if M is a maximally-consistent set, and $t \models M$, then $M = \{\psi \mid \psi \in ecl(\varphi) \wedge t \models \psi\}$. Let us consider the set of formulas $S = \{\psi \mid \psi \in ecl(\varphi) \wedge t^1 \models \psi\}$. Observe that S is a maximally-consistent set. By assumption of the theorem, we have that S is in $labGr_K(q_1)$. It is easy to verify that $follows(M, S)$. □

Theorem 3. *Let $V \subseteq Q$ be a set of vertices and $labGr_K$ a correct labeling with respect to φ and $Q \setminus ancestors_K(V)$. Then $relbl_K(\varphi, labGr_K, V)$ is a correct labeling w.r.t. φ and Q.*

Proof. We first note that only ancestors of nodes in V are relabeled—all the other nodes are correctly labeled by assumption on $labGr$. We say that a node q is at level k w.r.t. a set of vertices T iff the longest simple path from q to a node in T is k. Let H_k be the set of nodes at level k from V. We prove by induction on k that at k-th iteration, we have a correct labeling of K w.r.t. φ and $(S \setminus ancestors_K(V)) \cup H_k$, where S is the set of states of K. We can prove the inductive claim using Lemma 3. □

Corollary 1. *First, $modelCheck_K(\varphi) = true \iff K \models \varphi$. Second, for (K, K', U) and $labGr_K$ as above, we have $incrModelCheck(K, \varphi, U, labGr_K) = true \iff K \models \varphi$.*

Proof. Using Theorem 3, and the fact that the set $ancestors_K(S_f)$ is the set S of all states K, we obtain that $labGr_K = relbl_K(\varphi, labGr^0_K, S_f)$ is a correct labeling of K with respect to φ and S. In particular, for all initial states q_0, we have that for all $M \subset ecl(\varphi)$, $m \in labGr_K(q_0)$ iff there exists a trace $t \in traces_K(q_0)$ such that $t \models M$. We now use the definition of $checkInitStates$ to show that if $checkInitStates$ returns true, then there is no initial state q_0 such that there exists $M \in labGr_K(q_0)$ such that $\neg\varphi \in M$. Thus for all initial states q_0, for all traces t in $traces(t_0)$, we have that $t \models \varphi$.

The proof for incremental model checking is similar. □