Efficient Synthesis of Network Updates

PLDI'15 Submission #142

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, naively updating individual nodes can lead to incorrect transient behaviors, including loops, black holes, access control violations, and others. 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 that uses counterexample-guided search and incremental model checking to dramatically improve performance. We describe our prototype implementation, and present results from experiments on real-world topologies and properties demonstrating that our tool scales to updates involving thousands of nodes in a few seconds.

1. Introduction

Software-defined networking (SDN) is a new paradigm in which a logically-centralized *controller machine* 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 advanced traffic engineering, data center virtualization, fine-grained access control, and others [see Casado et al. 2014]. SDN has been used in production enterprise, datacenter, and wide-area networks, and new deployments are rapidly emerging.

Much of the power of SDN stems from the ability of controllers to effect changes to the *global* state of the underlying switches. For instance, 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 large distributed systems, with hundreds or even thousands of 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 that reaches the intended configuration. The code needed to implement this transition is tedious to write and prone to error—in general, packets may be processed by multiple intermediate configurations, creating new behaviors that would not be possible in either the initial or final configurations.

Problems related to network updates are not unique to SDN. Traditional distributed routing protocols also suffer from anomalies during reconvergence, including transient forwarding loops, blackholes, and access control violations. For users, these anomalies manifest in outages, degraded performance, and broken connections. The research community has developed techniques for preserving certain invariants during updates [Francois and Bonaventure 2007; Raza et al. 2011; Vanbever et al. 2011], but none of them fully solve 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) [John et al. 2008].

On the surface, it might seem that shifting to SDN could exacerbate update-related problems by making the network even more programmable and dynamic. But SDN also offers a tremendous opportunity to develop high-level update abstractions that implement updates automatically while preserving key invariants. Indeed, the authors of B4 [Jain et al. 2013]—the SDN controller that manages Google's worldwide inter-data center network—describe a vision where: "multiple, sequenced manual operations [are] not involved [in] virtually any management operation."

In previous work, Reitblatt et al. [2012] proposed the notion of a *consistent update*,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—it preserves *all* safety properties—but it comes at a high cost. The only general consistent update mechanism is a two-phase update, which tags packets with explicit versions and maintains rules for the initial and final configurations simultaneously. This leads to problems on switches with limited memory and also makes updates slower to complete due to the high degree of rule churn.

This paper proposes a different approach. Instead of forcing SDN programmers to implement updates by hand, as is typically done today, or using powerful but expensive mechanisms like consistent updates, we develop algorithms for synthesizing updates automatically from formal specifications. Given initial and final configurations and an Linear Temporal Logic (LTL) property that captures desired invariants during the update, we either generate an SDN program that implements the transition from initial to final configurations 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 properties specified in the formula, it is able to leverage strategies that

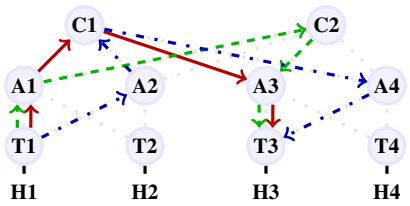

Figure 1: Example topology.

would be ruled out in other approaches. For example, if the programmer specifies a trivial property, the system is free to update the switches in any order. However, if she specifies a more complex property, such as "all packets must traverse the firewall," then the update is more constrained. In practice, our synthesized programs use less memory on switches and require less communication than competing approaches.

Synthesizing programs is challenging due to the high degree of concurrency inherent in networks—switches may interleave packet and control message processing arbitrarily, and a given packet may be processed by single switch multiple times. Hence, we must carefully analyze all possible event orderings and insert synchronization primitives to impose structure as needed. Our algorithm works by searching through the space of all possible sequences of individual switch updates, learning from counterexamples and employing an incremental model checker to re-use previously computed results whenever possible. This incremental model checking algorithm turns out to be somewhat novel in its own right—it exploits the fact that correct network configurations are loop-free to obtain an efficient procedure for re-checking properties after the underlying model has been changed. Because the synthesis algorithm poses a series of closely-related model checking questions, using an incremental model checker yields enormous performance improvements on updates to real-world networks.

We have implemented our algorithm, and identified heuristics to further speed up synthesis and eliminate spurious synchronization. We have integrated our tool into Frenetic [Foster et al. 2011], synthesized updates for OpenFlow switches, and used our system to process actual traffic generated by Linux hosts. To evaluate performance, we ran experiments on a suite of real-world topologies, network configurations, and properties. Our results demonstrate the effectiveness of synthesis, which scales to thousands of switches, and incremental model checking, which outperforms a popular symbolic model checker in *batch* mode and a state-of-the-art network model checker in incremental mode.

In summary, the main contributions of this paper are:

- We investigate the use of program 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).

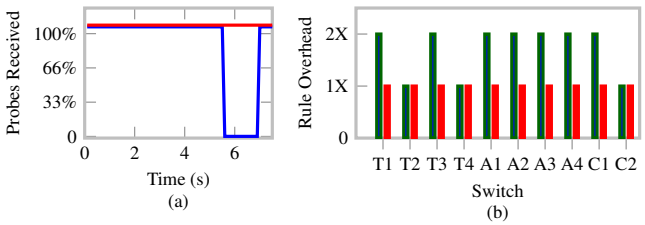

Figure 2: Experiments with naive (blue), two-phase (green), and ordering (red) updates: (a) probes received; (b) per-switch rule overhead.

- We present a incremental LTL model checker for loop-free models (§5).
- We build 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 behavior during updates.

2. Overview

To illustrate challenges that arise when updating a network, consider the network in Figure 1. It is organized into a data center FatTree topology [Al-Fares et al. 2008] with two core switches (C1 and C2), four aggregation switches (A1 to A4), four top-of-rack switches (T1 to T4), and four hosts (H1 to H4). Initially, we configure the switches to forward traffic from H1 to H3 along the solid red path: T1-A1-C1-A3-T3. Later on, we decide to shift traffic from the red path to the dashed green path, T1-A1-C2-A3-T3 (perhaps we need to take C1 down for maintenance). To implement this update, the SDN programmer would have to modify the forwarding rules installed on switch A1 and C2. Note, however, 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) shows the results of a simple experiment performed using our system that illustrates this behavior. Using the Mininet network simulator and OpenFlow switches, we continuously sent ICMP (`ping`) probes during a "naive" update (shown in blue) and the ordering update synthesized by our tool (shown in red). With the naive update, 100% of the probes are lost for several seconds while the ordering update maintains connectivity.

Consistency. Previous work by Reitblatt et al. [2012] 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 actually important, as long as the network eventually reaches the intended config-

uration, since most networks only provide best-effort packet delivery. While it is true that errors can be masked by protocols such as TCP with automatic-retry 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 they decide to migrate their infrastructure to a public cloud like Amazon EC2. To ensure that the cloud deployment is secure, the business would want to know that the same isolation properties hold in the cloud as did in their home office—in particular, all malicious traffic is blocked by the firewall. A best-effort strategy that only eventually reaches the target configuration could step through arbitrary intermediate configurations would not necessarily maintain this guarantee.

Two-Phase Updates. Reitblatt et al. [2012] also introduced a general technique for ensuring consistency called a *two-phase update*. The idea is to explicitly tag packets with versions upon ingress and use these version tags to determine which forwarding rules to use at each hop. Unfortunately, two-phase updates have a significant cost. During the transition, switches must maintain the forwarding rules for *both* configurations, effectively doubling the memory requirements needed to execute the program. This is not practical in many networks as switches store forwarding rules using ternary content-addressable 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 installed 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—on the order of tens of seconds for a few hundred rules [Jin et al. 2014]! Hence, because two-phase updates require modifying a large number of rules, they can take a correspondingly long time to complete. Together, these overheads often make two-phase updates a non-starter.

Our Approach. Our approach is based on the observation that in many settings consistent and two-phase updates are overkill. In certain situations, consistency can be achieved by simply choosing a correct order of updates of individual switches. 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 get often still get sufficiently strong guarantees for the specific application at hand 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 dashed and dotted blue 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. Fortunately, this failure to find a consistent update hints at the germ of 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 more situations. In fact, recent work by Mahajan and Wattenhofer [2013] has explored using ordering updates, but only for specific properties like loop-freedom, reachability, waypointing, service-chaining, 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 combinations of properties—e.g., a programmer could state that loop-freedom and service-chaining must both hold during an update.

In-flight Packets and Waits. To handle certain updates, an additional synchronization primitive is needed to generate correct ordering updates (or correct two-phase updates, for that matter). To illustrate, suppose we want to transition from the red path to the blue path, as above, but in addition to preserving connectivity, we want every packet to traverse either A2 or A3. Why might we want this? Perhaps those switches are actually middleboxes that scrub malicious packets before forwarding them to their destination. Now consider the update that modifies the configurations on A2, A4, T1, and C1, in that order. Between the time that we update T1 and C1, there might be a (small) number of 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 problem, we can simply pause after updating T1 until any packets it previously forwarded have left the network. We thus need a controller command "wait" that pauses 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 in all cases, once per update. Before deleting the old version of the rules on switches, we need to be sure that all in-flight packets have left the network. Further, the time needed to update a single switch can be upwards of 10 seconds [Jin et al. 2014; Lazaris et al. 2014], whereas typical transit time for data centers (the time it takes the packet to traverse a data center) is orders-of-magnitude lower, and is measure in microseconds [Alizadeh et al. 2010]. Hence, waiting for in-flight packets has a negligible overall effect.

Summary. This paper presents a sound and complete algorithm and accompanying implementation that synthesizes a large class of ordering updates 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 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 [Berry and Boudol 1990]. This model captures the key features of networks using a simple operational semantics. It is similar to the one used by Guha et al. [2013] 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 (TCP, UDP, etc.), 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 *data plane* of the network.

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, since it can be implemented with OpenFlow *bundles* [Open Networking Foundation 2013]. Sometimes switch granularity is too coarse to find an update sequence, in which case one can consider updating individual rules (*rule-granularity*). Our tool supports this finer-grained mode of operation, but since it is not conceptually different from switch granularity (a switch can be modeled as a network of smaller switches, one per rule), 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 until all packets annotated with the previous epoch have exited the network. 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 flush 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.

Elements. The elements E of the network model include switches S_i, links L_j, and a single controller element C. A *network* N is a tuple containing switches, links, and a controller: $\langle C, S_1, \cdots, S_k, L_1, \cdots, L_m \rangle$. 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, the controller C is represented by a record containing list of commands $cmds$ and an epoch ep. In this work, we assume that commands are totally-ordered. This can be ensured using OpenFlow *barrier* messages.

Operational semantics. The behavior of a network is defined by the small-step operational rules given in Figure 3. The rules define interactions between subsets of the elements, based on the semantics of OpenFlow switches [McKeown et al. 2008]. The states of the model are given by multisets Es of elements. We write $\{x\}$ to denote a singleton multiset, $m_1 \uplus m_1$ for the union of multisets m_1 and m_2. Similarly, we write $[x]$ for a singleton list, and $l_1 @ l_2$ for the concatenation of l_1 and l_2. We consider each transition $N \xrightarrow{o} N'$ to be annotated, with o being either an empty annotation, or an *observation* (sw, pt, pkt) indicating the location and packet being processed at that step.

The first rules describe network behavior in the data plane. The IN rule admits arbitrary packets into the network from a host, stamping them with the current controller epoch. The dual OUT rule removes a packet buffered on a link adjacent to a host. The PROCESS rule 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. The FORWARD rule moves a packet from a switch to the adjacent link. The final few rules describe control-plane behavior of the network. UPDATE replaces the table on a single switch. INCR increments the epoch on the con-

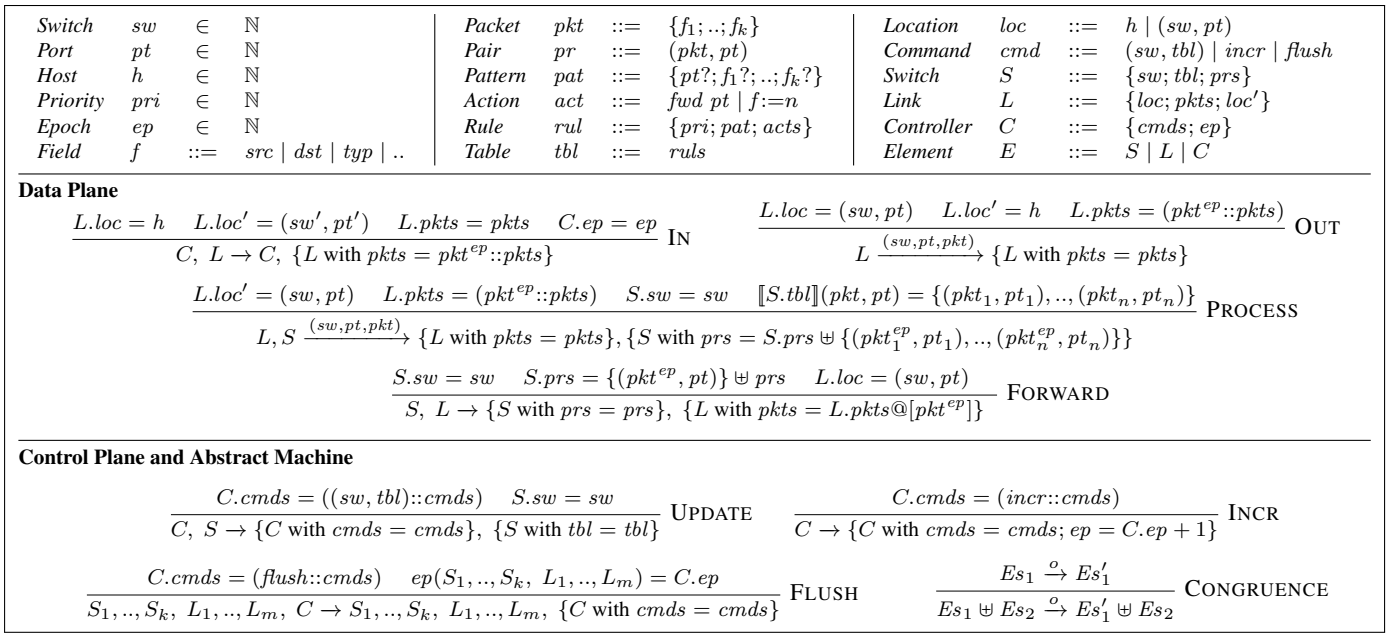

Figure 3: Network model.

troller, and FLUSH blocks the controller until all packets in the network are annotated with *at least* the current epoch. We write $ep(Es)$ to denote the smallest annotation on any packet in Es. The final rule, CONGRUENCE, allows any subcollection of elements in the network 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, we can use our operational rules to generate a sequence of observations. However, the network can process multiple packets concurrently, and we want the observations generated by a single packet. We define a successor relation $\overset{ep}{\sqsubseteq}$ for observations (Definition 8, 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. The 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 o'_i is* 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. In this paper we consider only loop-free traces. A network that forwards packets around a loop is generally considered to have an error. In the worst case, it can cause a packet storm, wasting bandwidth and degrading performance. Our tool detects and rejects configurations with loops automatically.

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 expressing trace-based properties, providing constructs for specifying the location and properties of a packet and its path through the network. In this paper, we will use Linear Temporal Logic (LTL) to describe traces in our network model. Let AP_H be atomic propositions that test the value of a switch, port, or packet field: $f_i = n$. Elements of the set 2^{AP_H} are called *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_H, 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 an formula φ, we write $N \models \varphi$ if $\mathcal{T}(N) \models \varphi$.

Problem Statement. We now formalize the network update problem. 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. We begin by developing notation for updating switches.

For 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 the network 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. 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 a sequence of static networks induced by $cmds$ which ends with N_n.

We call a network 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 9, Appendix A). This includes $\mathcal{T}(N)$ as well as traces of packets initially present in N. We call this $\bar{\mathcal{T}}(N)$, and note that for a *stable* network N, $\bar{\mathcal{T}}(N)$ 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 a stable static network N, a network configuration N', and an 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 using LTL model checkers, we now 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 network N, we can generate a Kripke structure $\mathcal{K}(N)$ containing corresponding traces (Definition 10, Appendix A). For simplicity, although networks support packet modification, we currently do not consider reasoning about them. This means that the Kripke structure has disjoint parts corresponding to the traffic classes. It is straightforward to enable packet modification, by adding the appropriate transitions between the parts of the Kripke structure, but we leave this for future work. The following lemma shows that the generated Kripke structure faithfully encodes the semantics of the network.

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 in finding correct update sequences is that the network is a distributed system. Hence, certain packets might "see" an inconsistent configuration (some switches updated, some not). Reasoning about all possible interleavings of commands quickly becomes intractable, but 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 remainder of this paper, we consider careful command sequences, and will develop a sound and complete algorithm that finds them efficiently. In Section 4, we describe a technique for removing wait commands which works very well in practice, but we leave a complete *optimal* wait removal for future work. Recall that $\mathcal{T}(N)$ denotes the sequence of all possible 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 careful command sequences 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*

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 message if no update sequence exists
1: $W \leftarrow false$; $V \leftarrow false$ ▷ Formula encoding wrong configurations.
2: (ok, L) ← DFSforOrder($N_i, \mathcal{K}(N_i), \bot, \varphi, \lambda_0$)
3: **if** ok **then return** L
4: **else return** "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** $s = \bot$ **then** (ok,cex,λ) ← modelCheck(K,φ)
8: **else**
9: (N,K,S) ← $swUpdate(N$,s)
10: (ok,cex,λ) ← incrModelCheck(K,φ,S,λ)
11: V ← V ∨ N
12: **if** ¬ ok **then**
13: W ← W ∨ analyzeCex(cex)
14: **return** (false,[])
15: **if** $N = N_f$ **then return** (true,[s])
16: **for** $s' \in$ possibleUpdates(N) **do**
17: (ok,L) ← DFSforOrder(N,K,s',φ,λ)
18: **if** ok **then return** (true,($upd\ s'$) :: L)
19: **return** (false,[])

Figure 4: ORDERUPDATE Algorithm.

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

4. Update Synthesis Algorithm

This section presents an algorithm for synthesizing network updates. It uses depth-first search (DFS) to explore the space of possible solutions, using counterexamples to detect wrong configurations, and exploiting several optimizations.

Algorithm. Figure 4 presents the ORDERUPDATE algorithm. It either returns a sequence of updates or fails if no such sequence exists. Most of the work is done by DFSFORORDER, which manages the search and invokes the model checker. We use DFS because we expect common properties/configurations to admit many update sequences.

Each call to DFSFORORDER attempts to add one switch to the current update sequence, yielding a new network configuration (the switch is given in parameter s to the procedure). 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, respectively.

To check whether all packet traces in this configuration satisfy the LTL property φ, we invoke our incremental model checking algorithm. More precisely, the first time DFSFORORDER is called, we call a full check of the model (line 7). The model checker labels the nodes of the Kripke structure 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 of related Kripke structures (line 10). The parameters passed in the incremental model checking call are: the updated Kripke structure K, the specification φ, the set of nodes S in K whose transition function has changed by the update of the switch s, and the correct labeling λ of the Kripke structure before the update. The algorithm for incremental model checking, and the functions $modelCheck$ and $incrModelCheck$ are defined in Section 5. Note that before the initial model checking, we convert the network configuration N to a Kripke structure. Furthermore, the update of the Kripke structure is performed by a function $swUpdate(N, s)$ that returns a triple (N', S, K'), where N is a static network, s is a switch, 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 has different outgoing transitions in K' than in K.

If the model checker returns true, then the N is safe and the search proceeds recursively. In this case, we add $(upd\ s')$ to the current sequence of commands. The statement $(upd\ s')$ is two commands for the Kripke structure: (s, tbl) where tbl is the forwarding table in the new configuration, and a $wait$ command. If the model checker returns false, the resulting counterexample is analyzed and added to W (see below), and the search backtracks.

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

Counterexamples. The counterexample-based pruning learns from counterexamples which network configurations do not satisfy the specification, thus avoiding model checking calls. The function $analyzeCex(cex)$ (Line 13) analyzes the counterexample cex, and returns a formula representing the set of switches that occurred in the counterexample trace, with flags indicating whether each switch was updated or not. This allows equivalent future configurations to be immediately eliminated without invoking the model checker. To illustrate, 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, meaning the intended specification (H1-H3 connectivity) will not be satisfied. The unsafe configuration consisting of A1 updated and C2 not updated will be added to W. In practice, many counterexamples are small compared to the total size of the network, and this greatly prunes the search space.

Early search termination. The early search termination optimization can help speed up the termination of the search when no update sequence is possible. Recall how the algorithm uses the 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 are a contradiction. In our implementation, this is done efficiently using an (incremental) SAT solver. If the solver determines that no update sequence is possible, the search can terminate.

Wait removal. The wait removal heuristic removes waits that are not necessary for correctness, based on the following observation. Consider two switches A, B, and consider the sequence of updates: A followed by B. If in the initial configuration, packets processed by A cannot reach B, then we do not need to wait after updating A and before updating B. Given a sequence of commands, we can remove unnecessary waits if we can maintain reachability-between-switches information. In our tool, this operates as a post-processing pass once an update sequence is found. In practice, this heuristic removes a majority of unnecessary waits (see Section 6).

Formal Properties. We prove that our algorithm is sound for the class of careful updates, and complete if we limit our search to *simple* update sequences:

Definition 6 (Simple Command Sequences)**.** *A sequence of commands* $(cmd_1 \cdots cmd_n)$ *is* simple *if each switch appears at most once in the sequence.*

Note that we could broaden our *simple* definition in many ways (e.g. *k-simple*, where each switch appears at most k times) while still remaining complete, but we have found the above restriction to work well in practice.

We prove *soundness* by showing that if the algorithm returns an update sequence, then this sequence is correct with respect to φ and N_i, and *completeness* by observing that it searches through all simple, careful sequences.

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 as it computes updates. 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. It works by re-labeling the states of a previously labeled Kripke structure with the (possibly different) set of formulas that hold after an update.

State Labeling. We begin by presenting a simple algorithm for labeling states of a Kripke structure with sets of formulas, following the approach introduced by Wolper, Vardi, and Sistla [1983] (WVS), and following the presentation by Vardi and Wolper [1986]. The WVS algorithm translates an LTL formula φ into a local automaton and an eventuality automaton. The local automaton checks the consistency between a state and its parent, and handles labeling of all formulas except $\varphi_1 \; U \; \varphi_2$, which is checked by the eventuality automaton. The two automata are then composed into a single Büchi automaton A whose states correspond to subsets of the set of subformulas of φ and their negations.

Hence, we label each state of the Kripke structure by a set L of sets of formulas s.t. 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. When we define $ecl(\varphi)$, we will see that these are exactly the formulas in $ecl(\varphi)$ that t satisfies.

We now describe state labeling precisely, deferring definitions of several auxiliary functions to Appendix C (Figure 7). 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. 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 (e.g. $\varphi_1 \vee \varphi_2 \in M$ if and only if $\varphi_1 \in M$ or $\varphi_2 \in M$). Likewise, the function $follows(M_1, M_2)$ captures the notion of successor induced by LTL's temporal operators, lifted to maximally-consistent sets (e.g., $X \; \varphi_1 \in M_1$ iff $\varphi_1 \in M_2$). 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 $(\forall \psi \in M : Holds_0(q, \psi))$. The function $Holds_0$ computes a predicate that is true if and only if ψ holds at q. For example, $Holds_0(q, \phi_1 \text{ U } \phi_2)$ is defined as $Holds_0(q, \phi_2)$ because the only transition from q is a self-loop.

For the second case, suppose q is a non-sink state. If we are given a labeling for $succ_K(q)$, 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 $M \in L$ if and only if $M \subseteq ecl(\varphi)$ and 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

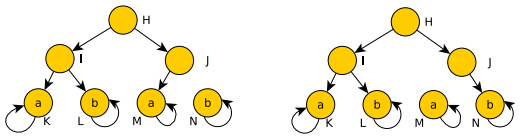

Figure 5: Incremental labeling—Initial (left), Final (right)

$traces_K(q)$ 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 w.r.t. φ and $succ_K(q)$, then $Holds_K(q, M, labGr_K) \Longleftrightarrow \exists t \in traces_K(q) : t \models M$.*

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 $\{M \mid M \in ecl(\varphi) \wedge Holds_K(q, M, labGr_K)\}$.

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

Let us assume that we have two Kripke structures $K = (Q, Q_0, \delta, \lambda)$ and $K' = (Q', Q'_0, \delta', \lambda')$, such that $Q = Q'$. 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 naive 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 structure on the left side of Figure 5, with H the only initial state. Suppose that the update modifies J, and the δ' relation only contains the pair (J, N). The resulting structure is shown on the right side of Figure 5. Consider labeling the structure with formulas $F\ a$, $F\ b$, and $F\ a \vee F\ b$. On the left hand side, we will have that the nodes K and M are labeled by $F\ a$, the nodes L and N by $F\ b$, the nodes I and H by $F\ a \vee F\ b$, and the node J by $F\ a$. Also assume that we want to label the Kripke structure after the update on the right-hand side. Given that the update changes only node J, the labeling can only change for J and its ancestors. We therefore start labeling node J, and find that its label becomes $F\ b$. Labeling proceeds to H, and finds that its label has not changed, it is still $F\ a \vee F\ b$. The labeling process could then stop, even if node H had 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 $Q \setminus ancestors(V)$, and returns a correct labeling of K with respect to φ and Q. This function $relbl_K$ is:

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

where $labGr'(v)$ is $labelNode_K(\varphi, v, labGr)$ if $v \in V$, and it is $labGr(v)$ 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 with respect to 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 $\forall q_0 \in Q_0, M \in labGr(q) : \varphi \in M$. 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 lG_0, $relbl(\varphi, lG_0, Q_f)$ is a correct labeling with respect to φ and Q. The function $modelCheck_K(\varphi)$ is defined to be equal to $checkInitStates_K(relbl_K(\varphi, lG_0, Q_f))$, where we can define lG_0 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 $checkInitStates_{K'}(relbl_{K'}(\varphi, labGr_K, U))$.

Corollary 1. *First, $modelCheck_K(\varphi) = true \Longleftrightarrow K \models \varphi$. Second, $incrModelCheck(K, \varphi, U, labGr) = true \Longleftrightarrow K \models \varphi$.*

The runtime complexity of the $modelCheck$ function is $O(|K| \times 2^{|\varphi|})$. The runtime complexity of $incrModelCheck$ is $O(|ancestors_K(U)| \times 2^{|\varphi|})$, where U is the set of nodes being updated. Note that to achieve this complexity, the function $labelNode$ must be implemented efficiently.

Counterexamples. This incremental algorithm can be extended to 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 $labelNode$, 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 ex-

plains 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 built a complete implementation of our synthesis tool. It consists of about 7K lines of OCaml code implementing the synthesis algorithm (Section 4), incremental model checker (Section 5), and several interfaces for specifying network updates. The tool works by building a Kripke structure (Section 3) and then interacting with the model checker MC to check switch updates. We provide four MC backends: *Incremental* uses the incremental algorithm to check and recheck formulas, *Batch* uses a labeling technique but re-labels the entire graph on each call, *NuSMV* encodes the current network configuration as a transition relation (symbolic model) and queries NuSMV at each step, and *NetPlumber* uses the incremental network model checker introduced in [Kazemian et al. 2013]. All except NetPlumber provide counterexamples for falsified properties, so we learn from the counterexamples when possible (Section 4).

Experimental setup. To evaluate performance of our tool, we used a 64-bit Ubuntu workstation with 20 GB RAM and a quad-core Intel i5-4570 CPU (3.2 GHz), conducting experiments on various topologies, configurations, and properties. We ask (1) whether our tool can solve update problems of realistic-size, and (2) how it scales. We obtained real/synthetic network topologies, ranging in size from small (5-10 switches) to large (1000+ switches). We used the *Topology Zoo* [Knight et al. 2011] dataset, which features 261 actual wide-area topologies, as well as synthetically constructed *Small-World* [Newman et al. 2001] and *FatTree* [Al-Fares et al. 2008] topologies, which are often used in datacenters. We then built configurations for these topologies, constructed formulas satisfied in the initial/final configurations, and ran our synthesizer to measure computation time.

Building configurations and properties. To create a configuration for a topology T, we first search for physical loops. We then randomly select (non-intersecting) loops to use for source/destination pairs. Each of the loops receives a source on one side, and a destination on the other, and we use previously-computed shortest-path information to obtain two disjoint paths from source to destination, each traversing a different set of waypoint nodes W_i and W_f on the two sides of the physical loop. This forms a "diamond" structure. We generate a property for a subset of these diamonds:

- *Reachability:* traffic from a given source must reach a certain destination:$(\text{port} = s1) \Rightarrow \text{F}\,(\text{port} = d1)$
- *Waypointing:* traffic must traverse a waypoint w: $(\text{port} = s1) \Rightarrow ((\text{port} \neq d1)\,\text{U}\,((\text{port} = w_i) \wedge \text{F}\,(port = d1)))$
- *Service chaining:* traffic must waypoint through several intermediate nodes: $(\text{port} = s1) \Rightarrow ((\text{port} \neq x \wedge \text{port} \neq d1)\,\text{U}\,((\text{port} = w_i) \wedge ((\text{port} \neq d1)\,\text{U}\,((\text{port} = x) \wedge \text{F}\,(port = d1)))))$

Once we have one of the above formulas for each chosen source-destination pair, we use the the conjunction of all the formulas as the property to be checked.

Incremental vs. monolithic. We compared the performance of Incremental against NuSMV for the reachability property (Figure 6 (a-c)). Of the 261 models in the Topology Zoo, our tool solved all of them more quickly using the Incremental backend versus the NuSMV one in batch mode. Speedups were very large, with an average of 651.48x. For the FatTree examples, average speedup was 1127.23x (36 examples), and for the Small-World examples, average speedup was 5156.35x (30 examples).

We also compared the performance of the Incremental and Batch backends using a large number of configurations (randomly perturbed from the ones in the NuSMV experiment). Batch relabels the entire graph during each model-checking call, incurring a large performance penalty. Incremental performed better on almost all examples, with average speedup of 9.95x, 70.38x, 56.46x on the three datasets shown in Figure 6(d-f). Maximum runtimes for Incremental were 0.26s, 5.23s, and 2.98s respectively.

Incremental performance We measured the performance of our Incremental backend versus the fast network property checker NetPlumber used as a backend (Figure 6(g-i)). It is important to note that NetPlumber uses rule-granularity (rather than switch-granularity) for updates, so we have enabled this in our tool for these experiments. For the three datasets, our checker was faster on (237/261=90.8%, 13/13=100%, 30/30=100%) of the examples, with overall average speedups of (8.34x, 6.85x, 15.53x). NetPlumber does not report counterexample traces, which puts it at a disadvantage in this end-to-end comparison, so we also measured total Incremental runtime vs. total NetPlumber runtime on the same set of model-checking questions posed by Incremental for the Small-World example. We were still faster on 30/30=100%, with an average speedup of 3.23x.

Scalability To quantify scalability of our system, we constructed Small World topologies up to 1500 switches, and experimented with all three types of properties, obtaining results in Figure 6(j-l). The upper figure (j) shows results for 1 large diamond update (containing about 25% of switches).

The largest example had 1015 nontrivial switches. The maximum graph generation time was 0.61s, and the maximum synthesis time was $(129.04, 30.11, 0.85)$s. Full data is tabulated in Appendix D. This experiment shows that our implementation is able to scale to networks of realistic size.

(Switch-)Impossible Updates As mentioned, sometimes switch-granularity cannot find an update. In Figure 6(k), we run the same experiment, but generate a second diamond atop the first one, requiring it to route traffic in the opposite direction. Switch-granularity reports it unsolvable in maximum time $(153.48, 33.48, 0.69)$s. We then use rule-granularity to solve these, and the tool is successful up to 1000 switches in maximum times $(793.11, 549.81, 77.91)$s.

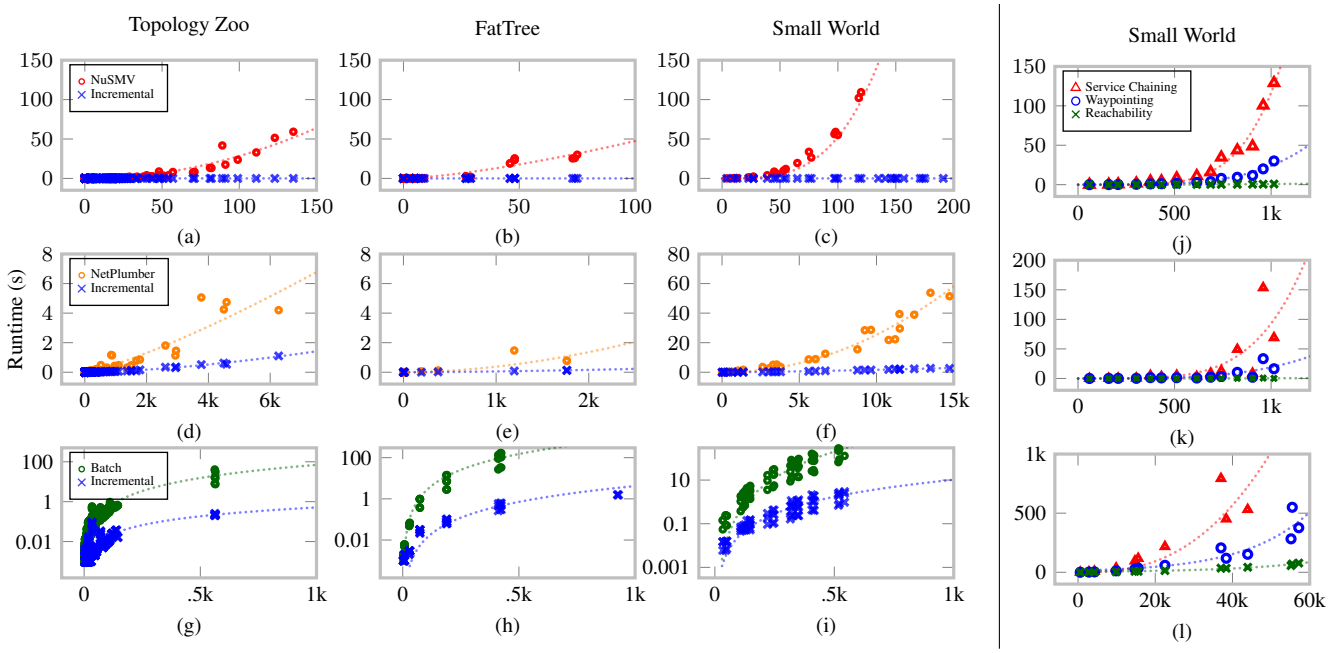

Figure 6: Experimental results: (a-i) Incr. performance vs. NuSMV, NetPlumber, Batch solvers (rows) on Topology Zoo, FatTree, SmallWorld topologies (columns); (j) scalability of Incremental on SmallWorld topologies of increasing size; (k) scalability when no correct switch-granularity update exists, and (l) rule-granularity solving switch-impossible examples of (k). The x-axis is num. of updating items (rules for *d-f* and *l*, switches otherwise) and y-axis is runtime.

7. Related Work

This paper extends preliminary work on update synthesis reported in a workshop paper [Noyes et al. 2013]. We present a more precise and realistic network model. Our system replaces expensive calls to an external model checker with calls to a new built-in *incremental* network model checker. We extend the basic 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.

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 [Francois et al. 2007; Raza et al. 2011]. Other work avoids routing loops and blackholes during IGP migration [Vanbever et al. 2011]. Work on network updates in SDN includes Reitblatt et al., who proposed the notion of *consistent updates* and several implementation mechanisms, including two-phase updates [Reitblatt et al. 2012]. Other work explores propagating updates incrementally, reducing the space overhead on switches [Katta et al. 2013]. As mentioned in Section 2, recent work proposes ordering updates for specific properties [Jin et al. 2014], whereas here we can handle combinations and variants of these properties as well. Furthermore, SWAN and zUpdate add support for bandwidth guarantees [Hong et al. 2012; Liu et al. 2013].

Model checking. Model checking has been used to verify network properties in several recent systems [Al-Shaer and Al-Haj 2010; Mai et al. 2011; Kazemian et al. 2012; Khurshid et al. 2012; Majumdar et al. 2014]. The closest to our work is NetPlumber [Kazemian et al. 2013], which is incremental. Surface-level differences include the specification languages (LTL vs. regular expressions), and NetPlumber's restriction of checking properties only on probe nodes. The main difference is incrementality of the model checking: NetPlumber keeps track of which new paths through the network have been added by an update, and checks the properties for those. We do not need to check along the whole length of the path, as we re-use previous model checking runs. The empirical comparison (Section 6) showed better performance of our tool as a back-end for synthesis.

Incremental model checking has been studied previously. Sokolsky and Smolka [1994] present 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. Chockler et al. [2011] introduced an incremental algorithm that is specific to the type of partial results produced by IC3 [Bradley 2011].

8. Conclusion

We present a practical tool for automatically synthesizing correct network update sequences from formal specifications. We build 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.

References

M. Al-Fares, A. Loukissas, and A. Vahdat. A Scalable, Commodity Data Center Network Architecture. In *SIGCOMM*, pages 63–74, Aug. 2008.

E. Al-Shaer and S. Al-Haj. FlowChecker: Configuration analysis and verification of federated OpenFlow infrastructures. In *SafeConfig*, 2010.

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.

G. Berry and G. Boudol. The chemical abstract machine. In *POPL*, pages 81–94, 1990.

A. Bradley. SAT-based model checking without unrolling. In *VMCAI*, 2011.

M. Casado, N. Foster, and A. Guha. Abstractions for software-defined networks. *CACM*, 57(10):86–95, Oct. 2014.

H. Chockler, A. Ivrii, A. Matsliah, S. Moran, and Z. Nevo. Incremental formal verification of hardware. In *FMCAD*, pages 135–143, 2011.

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.

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.

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.

A. Guha, M. Reitblatt, and N. Foster. Machine-verified network controllers. In *PLDI*, June 2013.

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.

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*, SIGCOMM '13, pages 3–14, 2013. ISBN 978-1-4503-2056-6.

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.

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.

N. P. Katta, J. Rexford, and D. Walker. Incremental consistent updates. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 49–54. ACM, 2013.

P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, 2012.

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.

A. Khurshid, W. Zhou, M. Caesar, and P. Godfrey. VeriFlow: Verifying network-wide invariants in real time. *ACM SIGCOMM Computer Communication Review*, pages 467–472, 2012.

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.

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.

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.

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

H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. Godfrey, and S. T. King. Debugging the data plane with anteater. In *SIGCOMM*, pages 290–301, Aug. 2011.

R. Majumdar, S. Tetali, and Z. Wang. Kuai: A model checker for software-defined networks. In *FMCAD*, 2014.

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 Computer Communication Review*, 38(2):69–74, 2008.

M. E. Newman, S. H. Strogatz, and D. J. Watts. Random graphs with arbitrary degree distributions and their applications. *Physical Review E*, 64(2):026118, 2001.

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

Open Networking Foundation. Openflow 1.4 specification, Oct. 2013. URL `https://www.opennetworking.org/sdn-resources/onf-specifications/`.

S. Raza, Y. Zhu, and C.-N. Chuah. Graceful network state migrations. *IEEE/ACM Transactions on Networking (TON)*, 19(4): 1097–1110, 2011.

M. Reitblatt, N. Foster, J. Rexford, C. Schlesinger, and D. Walker. Abstractions for network update. In *SIGCOMM*, pages 323–334. ACM, 2012.

O. Sokolsky and S. Smolka. Incremental model checking in the modal mu-calculus. In *CAV*, pages 351–363, 1994.

L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure. Seamless network-wide IGP migrations. In *SIGCOMM*, pages 314–325, 2011.

M. Y. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification (preliminary report). In *LICS*, pages 332–344, 1986.

P. Wolper, M. Y. Vardi, and A. P. Sistla. Reasoning about infinite computation paths (extended abstract). In *FOCS*, pages 185–194, 1983.

A. Auxiliary Definitions for Network Model

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

Definition 7 (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. *$ep > 0$, then (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 $tbl = tbl'$, (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 another observation o'.

Definition 8 (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 9 (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 exactly like regular single-packet traces, except they 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 10 (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 of initial states 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 $L_j.loc' = (sw, pt)$. The 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 (inducing a self-loop), or reaching an egress (also 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 11 (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 none of the transitions are update commands. For a single-packet trace t, we have $t \in \mathcal{T}(N_1)$ if and only if $t \in \mathcal{T}(N_n)$.*

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. Correctness Proofs for Synthesis Algorithm

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 10), and these are equivalent via Definition 11.

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 wish to show that there exists an equivalent trace $t' \in \mathcal{K}(N)$ 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 $t^k \lesssim t'^k$. Note that we have the successor relation $o_k \sqsubseteq o_{k+1}$, so Definition 8 and 10 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 wish to show that there exists an equivalent $t \in \mathcal{T}(N)$ 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 the transition $s_k \to s_{k+1}$. If $s_k = s_{k+1}$, then $t' = t'^k$, so we can simply 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 10, which correspond to the cases in Definition 8, 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 $N \xrightarrow{o_1'} \ldots \xrightarrow{o_k'} N_k$ 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 command 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 network N. If there is a single 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 $j \in [0, n]$, $N_j \models \varphi$. 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. Correctness Proofs for Incremental Model Checking.

Lemma 3. First, for sink states, observe that there is a unique trace t in $traces(s)$, as s is a sink state. We first prove that $t \models \varphi$ iff $Holds_0(s, \varphi)$. 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 $HoldsSink(s, M)$ for sink states to conclude the proof.

Now consider non-sink states: we first prove soundness, i.e., if $Holds_K(s, M, labGr_K)$, then there exists $t \in traces(s)$ such that $t \models M$. We have $Holds_K(s, M, labGr_K)$ iff $(\lambda(s) = (AP \cap M))$ and there exists $s' \in succ_K(M)$, and $M' \in labGr_K(s')$ such that $follows(M, M')$. By assumption of the theorem, we have that if $M' \in labGr_K(s')$, then there exists a trace t' in $traces(s')$ such that $t' \models M'$. Consider a trace t such that $t_0 = s$ and $t^1 = t'$. For each formula $\psi \in M$, we can prove that $t \models \psi$, which finishes the soundness proof. The base case of the proof by induction is implied by the fact that $s \models (AP \cap M)$. 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 $traces_K(s)$ such that $t \models M$, then $Holds_K(s, M, labGr_K)$ is true. Let t be the trace $ss_1s_2 \ldots$. 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 $labGr_K(s_1)$. It is easy to verify that $follows(M, S)$. This concludes the completeness proof. □

Theorem 3. We first note that only ancestors of nodes in V are re-labeled – all the other nodes are correctly labeled by assumption on $labGr$. We say that a node s is at level k with respect to a set of vertices T iff the longest simple path from s 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, $(Sancestors_K(V)) \cup H_k$ is a correct labeling of K w.r.t. φ and V. We can prove the inductive claim using Lemma 3. □

Corollary 1. The result is obtained as a corollary of Theorem 3. Using this theorem, and the fact that the set $ancestors_K(S_f)$ is the set S of all states K, we obtain that $lG = relbl_K(\varphi, lG_0, S_f)$ is a correct labeling of K with respect to φ and S. In particular, for all initial states s_0, we have that for all $M \subset ecl(\varphi)$, $m \in lG(s_0)$ iff there exists a trace $t \in traces_K(s_0)$ 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 s_0 such that there exists $M \in lG(s_0)$ such that $\neg\varphi \in M$. Thus for all initial states s_0, for all traces t in $traces(t_0)$, we have that $t \models \varphi$.

The proof for the incremental model checking function is similar. □

Extended closure of a formula φ in NNF:

- $true \in ecl(\varphi)$
- $\varphi \in ecl(\varphi)$
- 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 $\varphi_2 \in ecl(\varphi)$.

The function $Holds_0$ for sink states s:

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

Maximally-consistent sets $M \subseteq ecl(\phi)$

- $true \in M$
- $\psi \in M$ 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 $\varphi_2 \in M$)

The *follows* function

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

Figure 7: Auxiliary definitions for the state-labeling algorithm.

D. Experimental Data.

In the following tables, we report File Name, Answer, Number of Switches, Number of Nontrivial (Updating) Switches, Number of Rules, Number of Nontrivial (Updating) Rules, Graph Generation Time, Model Checking Time, Synthesis Total Time, Wait Removal Time, and Number of Waits for the scalability results in Figure 6.

"Answer" is a code where 1 means success, 0 means failure (timeout or out-of-memory), and 2 means the update is found to be impossible.

File Name	Answer	Sw. (#)	Nontriv. Sw. (#)	Rl. (#)	Nontriv. Rl. (#)	Graph Gen. (s)	Model Check. (s)	Synth. Tot. (s)	Wt. Rem. (s)	Wt. (#)
sw-100-4.gml	1	100	59	100	59	0.00	0.04	0.05	0.00	2
sw-200-4.gml	1	200	159	200	159	0.05	0.14	0.15	0.02	2
sw-300-4.gml	1	300	208	300	208	0.03	0.52	0.53	0.03	2
sw-400-4.gml	1	400	302	400	302	0.06	1.55	1.58	0.06	2
sw-500-4.gml	1	500	375	500	375	0.09	4.14	4.20	0.10	2
sw-600-4.gml	1	600	432	600	432	0.11	4.27	4.34	0.12	2
sw-700-4.gml	1	700	511	700	511	0.17	8.03	8.13	0.18	2
sw-800-4.gml	1	800	615	800	615	0.26	10.87	11.02	0.31	2
sw-900-4.gml	1	900	687	900	687	0.29	15.68	15.88	0.46	2
sw-1000-4.gml	1	1,000	742	1,000	742	0.24	34.35	34.57	0.34	2
sw-1100-4.gml	1	1,100	825	1,100	825	0.30	43.27	43.66	0.44	2
sw-1200-4.gml	1	1,200	904	1,200	904	0.33	48.25	48.64	0.60	2
sw-1300-4.gml	1	1,300	959	1,300	959	0.30	100.16	100.56	0.53	2
sw-1400-4.gml	1	1,400	1,015	1,400	1,015	0.61	128.46	129.04	0.87	2

Table 1: Data for switch-possible updates, Figure 6(j) (Service Chaining)

File Name	Answer	Sw. (#)	Nontriv. Sw. (#)	Rl. (#)	Nontriv. Rl. (#)	Graph Gen. (s)	Model Check. (s)	Synth. Tot. (s)	Wt. Rem. (s)	Wt. (#)
sw-100-4.gml	1	100	59	100	59	0.00	0.02	0.02	0.00	2
sw-200-4.gml	1	200	159	200	159	0.05	0.05	0.06	0.02	2
sw-300-4.gml	1	300	208	300	208	0.03	0.16	0.17	0.03	2
sw-400-4.gml	1	400	302	400	302	0.06	0.43	0.46	0.10	2
sw-500-4.gml	1	500	375	500	375	0.15	0.91	0.96	0.10	2
sw-600-4.gml	1	600	432	600	432	0.11	1.17	1.24	0.12	2
sw-700-4.gml	1	700	511	700	511	0.16	1.90	1.99	0.18	2
sw-800-4.gml	1	800	615	800	615	0.25	2.74	2.88	0.32	2
sw-900-4.gml	1	900	687	900	687	0.27	3.72	3.90	0.38	2
sw-1000-4.gml	1	1,000	742	1,000	742	0.23	7.76	7.96	0.35	2
sw-1100-4.gml	1	1,100	825	1,100	825	0.30	9.12	9.37	0.45	2
sw-1200-4.gml	1	1,200	904	1,200	904	0.37	11.34	11.67	0.60	2
sw-1300-4.gml	1	1,300	959	1,300	959	0.30	19.65	20.03	0.55	2
sw-1400-4.gml	1	1,400	1,015	1,400	1,015	0.59	29.58	30.11	0.89	2

Table 2: Data for switch-possible updates, Figure 6(j) (Waypointing)

File Name	Answer	Sw. (#)	Nontriv. Sw. (#)	Rl. (#)	Nontriv. Rl. (#)	Graph Gen. (s)	Model Check. (s)	Synth. Tot. (s)	Wt. Rem. (s)	Wt. (#)
sw-100-4.gml	1	100	59	100	59	0.00	0.00	0.00	0.00	2
sw-200-4.gml	1	200	159	200	159	0.05	0.01	0.02	0.02	2
sw-300-4.gml	1	300	208	300	208	0.03	0.02	0.03	0.03	2
sw-400-4.gml	1	400	302	400	302	0.06	0.03	0.07	0.06	2
sw-500-4.gml	1	500	375	500	375	0.09	0.05	0.13	0.10	2
sw-600-4.gml	1	600	432	600	432	0.11	0.06	0.12	0.12	2
sw-700-4.gml	1	700	511	700	511	0.17	0.09	0.17	0.18	2
sw-800-4.gml	1	800	615	800	615	0.25	0.14	0.28	0.31	2
sw-900-4.gml	1	900	687	900	687	0.26	0.16	0.35	0.38	2
sw-1000-4.gml	1	1,000	742	1,000	742	0.23	0.16	0.35	0.34	2
sw-1100-4.gml	1	1,100	825	1,100	825	0.37	0.20	0.44	0.45	2
sw-1200-4.gml	1	1,200	904	1,200	904	0.38	0.25	0.57	0.60	2
sw-1300-4.gml	1	1,300	959	1,300	959	0.30	0.24	0.58	0.55	2
sw-1400-4.gml	1	1,400	1,015	1,400	1,015	0.59	0.34	0.85	0.88	2

Table 3: Data for switch-possible updates, Figure 6(j) (Reachability)

File Name	Answer	Sw. (#)	Nontriv. Sw. (#)	Rl. (#)	Nontriv. Rl. (#)	Graph Gen. (s)	Model Check. (s)	Synth. Tot. (s)	Wt. Rem. (s)	Wt. (#)
sw-100-4.gml	2	100	59	100	59	0.00	0.05	0.05	0.00	0
sw-200-4.gml	2	200	159	200	159	0.05	0.17	0.18	0.00	0
sw-300-4.gml	2	300	208	300	208	0.04	0.51	0.52	0.00	0
sw-400-4.gml	2	400	302	400	302	0.09	0.61	0.61	0.00	0
sw-500-4.gml	2	500	375	500	375	0.18	5.04	5.10	0.00	0
sw-600-4.gml	2	600	432	600	432	0.15	3.25	3.27	0.00	0
sw-700-4.gml	2	700	511	700	511	0.22	5.24	5.27	0.00	0
sw-800-4.gml	2	800	615	800	615	0.36	3.23	3.25	0.00	0
sw-900-4.gml	2	900	687	900	687	0.42	8.88	8.93	0.00	0
sw-1000-4.gml	2	1,000	742	1,000	742	0.33	12.91	12.95	0.00	0
sw-1100-4.gml	2	1,100	825	1,100	825	0.39	48.86	49.23	0.00	0
sw-1200-4.gml	2	1,200	904	1,200	904	0.48	8.86	8.89	0.00	0
sw-1300-4.gml	2	1,300	959	1,300	959	0.43	152.90	153.48	0.00	0
sw-1400-4.gml	2	1,400	1,015	1,400	1,015	0.84	68.78	68.93	0.00	0

Table 4: Data for switch-impossible updates, Figure 6(k) (Service Chaining)

File Name	Answer	Sw. (#)	Nontriv. Sw. (#)	Rl. (#)	Nontriv. Rl. (#)	Graph Gen. (s)	Model Check. (s)	Synth. Tot. (s)	Wt. Rem. (s)	Wt. (#)
sw-100-4.gml	2	100	59	100	59	0.00	0.02	0.02	0.00	0
sw-200-4.gml	2	200	159	200	159	0.05	0.07	0.07	0.00	0
sw-300-4.gml	2	300	208	300	208	0.04	0.16	0.17	0.00	0
sw-400-4.gml	2	400	302	400	302	0.08	0.21	0.22	0.00	0
sw-500-4.gml	2	500	375	500	375	0.18	1.34	1.40	0.00	0
sw-600-4.gml	2	600	432	600	432	0.16	0.92	0.94	0.00	0
sw-700-4.gml	2	700	511	700	511	0.22	1.31	1.33	0.00	0
sw-800-4.gml	2	800	615	800	615	0.36	0.75	0.77	0.00	0
sw-900-4.gml	2	900	687	900	687	0.40	2.10	2.15	0.00	0
sw-1000-4.gml	2	1,000	742	1,000	742	0.32	3.29	3.32	0.00	0
sw-1100-4.gml	2	1,100	825	1,100	825	0.40	10.10	10.31	0.00	0
sw-1200-4.gml	2	1,200	904	1,200	904	0.53	2.07	2.10	0.00	0
sw-1300-4.gml	2	1,300	959	1,300	959	0.41	32.92	33.48	0.00	0
sw-1400-4.gml	2	1,400	1,015	1,400	1,015	0.82	16.28	16.43	0.00	0

Table 5: Data for switch-impossible updates, Figure 6(k) (Waypointing)

File Name	Answer	Sw. (#)	Nontriv. Sw. (#)	Rl. (#)	Nontriv. Rl. (#)	Graph Gen. (s)	Model Check. (s)	Synth. Tot. (s)	Wt. Rem. (s)	Wt. (#)
sw-100-4.gml	2	100	59	100	59	0.00	0.00	0.00	0.00	0
sw-200-4.gml	2	200	159	200	159	0.06	0.01	0.01	0.00	0
sw-300-4.gml	2	300	208	300	208	0.04	0.01	0.02	0.00	0
sw-400-4.gml	2	400	302	400	302	0.08	0.01	0.01	0.00	0
sw-500-4.gml	2	500	375	500	375	0.18	0.03	0.08	0.00	0
sw-600-4.gml	2	600	432	600	432	0.15	0.01	0.03	0.00	0
sw-700-4.gml	2	700	511	700	511	0.22	0.02	0.05	0.00	0
sw-800-4.gml	2	800	615	800	615	0.37	0.01	0.03	0.00	0
sw-900-4.gml	2	900	687	900	687	0.39	0.03	0.08	0.00	0
sw-1000-4.gml	2	1,000	742	1,000	742	0.32	0.03	0.06	0.00	0
sw-1100-4.gml	2	1,100	825	1,100	825	0.41	0.08	0.26	0.00	0
sw-1200-4.gml	2	1,200	904	1,200	904	0.52	0.02	0.05	0.00	0
sw-1300-4.gml	2	1,300	959	1,300	959	0.41	0.13	0.69	0.00	0
sw-1400-4.gml	2	1,400	1,015	1,400	1,015	0.82	0.07	0.23	0.00	0

Table 6: Data for switch-impossible updates, Figure 6(k) (Reachability)

File Name	Answer	Sw. (#)	Nontriv. Sw. (#)	Rl. (#)	Nontriv. Rl. (#)	Graph Gen. (s)	Model Check. (s)	Synth. Tot. (s)	Wt. Rem. (s)	Wt. (#)
sw-100-4.gml	1	100	59	620	579	0.00	0.36	0.38	0.05	204
sw-200-4.gml	1	200	159	2,869	2,828	0.06	2.59	2.92	0.68	852
sw-300-4.gml	1	300	208	4,349	4,256	0.04	7.65	8.29	1.23	1,355
sw-400-4.gml	1	400	302	9,972	9,874	0.10	33.94	36.22	4.58	2,702
sw-500-4.gml	1	500	375	14,862	14,737	0.22	89.12	95.55	10.09	4,283
sw-600-4.gml	1	600	432	15,754	15,585	0.18	110.81	117.26	9.93	4,400
sw-700-4.gml	1	700	511	22,698	22,509	0.27	204.59	217.36	18.75	7,713
sw-800-4.gml	1	800	615	38,561	38,376	0.41	422.71	450.86	51.58	10,219
sw-900-4.gml	1	900	687	44,153	43,940	0.48	499.78	530.97	55.85	9,544
sw-1000-4.gml	1	1,000	742	37,283	37,024	0.33	760.93	793.11	41.95	11,762
sw-1100-4.gml	0	0	0	0	0	0.00	0.00	900.00	0.00	0
sw-1200-4.gml	0	0	0	0	0	0.00	0.00	900.00	0.00	0
sw-1300-4.gml	0	0	0	0	0	0.00	0.00	900.00	0.00	0
sw-1400-4.gml	0	0	0	0	0	0.00	0.00	900.00	0.00	0

Table 7: Data for rule-granularity, Figure 6(l) (Service Chaining)

File Name	Answer	Sw. (#)	Nontriv. Sw. (#)	Rl. (#)	Nontriv. Rl. (#)	Graph Gen. (s)	Model Check. (s)	Synth. Tot. (s)	Wt. Rem. (s)	Wt. (#)
sw-100-4.gml	1	100	59	620	579	0.00	0.13	0.15	0.04	204
sw-200-4.gml	1	200	159	2,869	2,828	0.05	1.07	1.36	0.62	852
sw-300-4.gml	1	300	208	4,349	4,256	0.04	2.42	2.94	1.07	1,355
sw-400-4.gml	1	400	302	9,972	9,874	0.08	10.35	12.22	4.09	2,702
sw-500-4.gml	1	500	375	14,862	14,737	0.18	23.01	27.79	8.87	4,283
sw-600-4.gml	1	600	432	15,754	15,585	0.15	29.82	34.57	9.01	4,400
sw-700-4.gml	1	700	511	22,698	22,509	0.22	49.63	58.85	15.97	7,713
sw-800-4.gml	1	800	615	38,561	38,376	0.36	96.75	118.54	42.15	10,219
sw-900-4.gml	1	900	687	44,153	43,940	0.41	123.56	152.17	53.25	9,544
sw-1000-4.gml	1	1,000	742	37,283	37,024	0.33	181.77	206.53	36.90	11,762
sw-1100-4.gml	1	1,100	825	55,439	55,164	0.41	242.89	282.80	67.39	16,766
sw-1200-4.gml	1	1,200	904	57,464	57,167	0.55	319.71	376.71	126.60	16,947
sw-1300-4.gml	1	1,300	959	55,827	55,486	0.50	484.68	549.81	97.95	18,625
sw-1400-4.gml	0	0	0	0	0	0.00	0.00	0.00	0.00	0

Table 8: Data for rule-granularity, Figure 6(l) (Waypointing)

File Name	Answer	Sw. (#)	Nontriv. Sw. (#)	Rl. (#)	Nontriv. Rl. (#)	Graph Gen. (s)	Model Check. (s)	Synth. Tot. (s)	Wt. Rem. (s)	Wt. (#)
sw-100-4.gml	1	100	59	620	579	0.00	0.03	0.06	0.04	204
sw-200-4.gml	1	200	159	2,869	2,828	0.05	0.29	0.58	0.62	852
sw-300-4.gml	1	300	208	4,349	4,256	0.04	0.47	0.99	1.08	1,355
sw-400-4.gml	1	400	302	9,972	9,874	0.08	1.60	3.60	4.11	2,702
sw-500-4.gml	1	500	375	14,862	14,737	0.19	2.98	8.00	8.91	4,283
sw-600-4.gml	1	600	432	15,754	15,585	0.15	3.05	7.94	8.81	4,400
sw-700-4.gml	1	700	511	22,698	22,509	0.22	5.36	14.43	15.63	7,713
sw-800-4.gml	1	800	615	38,561	38,376	0.35	12.88	34.00	40.37	10,219
sw-900-4.gml	1	900	687	44,153	43,940	0.40	15.96	42.55	51.67	9,544
sw-1000-4.gml	1	1,000	742	37,283	37,024	0.33	11.51	35.26	35.08	11,762
sw-1100-4.gml	1	1,100	825	55,439	55,164	0.40	18.70	56.48	64.92	16,766
sw-1200-4.gml	1	1,200	904	57,464	57,167	0.56	26.70	77.91	89.05	16,947
sw-1300-4.gml	1	1,300	959	55,827	55,486	0.42	20.50	70.28	69.75	18,625
sw-1400-4.gml	0	0	0	0	0	0.00	0.00	0.00	0.00	0

Table 9: Data for rule-granularity, Figure 6(l) (Reachability)