Toward Synthesis of Network Updates

Andrew Noyes
Cornell University

Todd Warszawski
Cornell University

Pavol Černý
University of Colorado Boulder

Nate Foster
Cornell University

Updates to network configurations are notoriously difficult to implement correctly. Even if the old and new configurations are correct, the update process can introduce transient errors such as forwarding loops, dropped packets, and access control violations. The key factor that makes updates difficult to implement is that networks are distributed systems with hundreds or even thousands of nodes, but updates must be rolled out one node at a time. In networks today, the task of determining a correct sequence of updates is usually done manually—a tedious and error-prone process for network operators. This paper presents a new tool for synthesizing network updates automatically. The tool generates efficient updates that are guaranteed to respect invariants specified by the operator. It works by navigating through the (restricted) space of possible solutions, learning from counterexamples to improve scalability and optimize performance. We have implemented our tool in OCaml, and conducted experiments showing that it scales to networks with a thousand switches and tens of switches updating.

1 Introduction

Most networks are updated frequently, for reasons ranging from taking devices down for maintenance, to modifying forwarding paths to avoid congestion, to changing security policies. Unfortunately, implementing a network update correctly is an extremely difficult task—it requires modifying the configurations of hundreds or even thousands of routers and switches, all while traffic continues to flow through the network. Implementing updates naively can easily lead to situations where traffic is processed by switches in different configurations, leading to problems such as increased congestion, temporary outages, forwarding loops, black holes, and security vulnerabilities.

The research community has developed a number of mechanisms for implementing network updates while preserving important invariants [3, 4, 6, 8, 11, 9]. For example, *consensus routing* uses distributed snapshots to avoid anomalies in routing protocols such as BGP [6]. Similarly, *consistent updates* uses versioning to ensure that every packet traversing the network will be processed with either the old configuration or the new configuration, but not a mixture of the two [9]. But these mechanisms are either limited to specific protocols and properties, or are general but expensive to implement, requiring substantial additional space on switches to represent the forwarding rules for different configurations.

This paper explores a different idea: rather than attempting to design a new concrete update mechanism, we use synthesis to generate such mechanisms automatically. With our system, the network operator provides the current and target configurations as input, as well as a collection of invariants that are expected to hold during the transition. (The current and target configurations should also satisfy these invariants.) The system either (i) generates a sequence of modifications to the forwarding rules on individual switches that transitions the network to the new configuration and preserves the specified invariants, or (ii) halts with a failure if no such sequence exists. Overall, our system takes a challenging programming task usually done by hand today and automates it, using a back-end solver to perform all tedious and error-prone reasoning involving low-level network artifacts.

B. Finkbeiner and A. Solar-Lezama (Eds.): SYNT 2013
EPTCS 142, 2014, pp. 8–23, doi:10.4204/EPTCS.142.8

Our system provides network operators with a general and flexible tool for specifying and implementing network updates efficiently. By enabling them to specify just the properties that are needed to ensure correctness, the synthesized updates are able to make use of mechanisms that would be ruled out in other systems. For example, if the operator specifies no invariants, then the tool can simply update the switches in any order, without worrying about possible ill-effects on in-flight packets. Alternatively, if the operator specifies an invariant that encodes a firewall, then the network may forward packets along paths that are different than the ones specified by the old and new policies, as long as all packets blocked by the firewall are dropped. This flexibility gives our system substantial latitude in generating update implementations, and allows it to generate efficient updates that converge faster, or use fewer forwarding rules, compared to general techniques such as consistent updates.

Operationally, our system works by checking network properties using a model checker. We encode the configuration of each switch into the model, as well as the contents and location of a single in-flight packet. Using this model, we then pose a sequence of queries to the model checker, attempting to identify a modification to some switch configuration that will transition the network to a more updated state without violating the specified invariants. Determining whether a configuration violates the invariants is a straightforward LTL model checking problem. If this step succeeds, then we recurse and continue the process until we eventually arrive at the new configuration. Otherwise, we use the counterexample returned by the model checker to refine our model and repeat the step.

We are able to reduce our synthesis problem to a reachability problem (as opposed to a game problem) because we assume that the environment is stable during the time the updates are performed. That is, we assume that switches do not come up or go down, and that no other updates are being performed simultaneously. The key challenge in our setting stems from the fact that although the individual switch modifications only need to maintain a correct overall configuration, network configurations are rich structures, so navigating the space of possible updates effectively is critical. We plan to investigate the game version of our synthesis problem in future work.

In summary, this paper makes the following contributions

- We present a novel approach to specifying and implementing network updates using synthesis.
- We develop encodings and algorithms for automatically synthesizing network updates using a model checker, and optimizations that improve its scalability and performance.
- We describe a prototype implementation and present the results of experiments demonstrating that even our current prototype tool is able to scale to networks of realistic size.

The rest of this paper is structured as follows. Section 2 provides an overview to the network update problem and discusses examples that illustrate the challenges of synthesizing updates. Section 3 develops our abstract network model and defines the update problem formally. Section 4 presents algorithms for synthesizing network updates. Section 5 describes our implementation. Section 6 presents the results of our experiments. Section 7 discusses related work. We conclude in Section 8.

2 Overview

This section provides a basic overview of primitive network update mechanisms, and presents examples that illustrate the inherent challenges in implementing network updates.

Basics. Abstractly, a network can be thought of as a graph with switches as nodes and links as edges. The behavior of each switch is determined by a set of forwarding rules installed locally. A forwarding

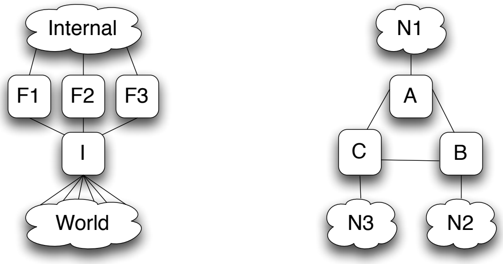


Figure 1: Example network topologies: (a) distributed firewall, (b) cycle.

rule consists of a pattern, which describes a set of packets, and a list of actions, which specify how packets matching the pattern should be processed. For the purposes of this paper, the precise capabilities of patterns and actions and the details of how they are represented on switches will not be important. However, typically patterns support matching on packet headers and actions support optionally modifying those headers and forwarding packets out one of its ports.

To process a packet, the network interleaves steps of processing using the rules installed on switches and steps of processing using the links themselves. More specifically, given a packet located at a particular switch, the switch finds a matching rule and applies its actions to the packet. This moves the packet to an output port on the switch (or drops it). Assuming there is a link connected to that port, the network will then transmit the packet to the adjacent switch, and processing continues.

A network property is a set of paths through the topology. Such properties can be used to capture basic reachability properties such as connectivity and loop freedom, as well as more intricate properties such as access control.

To implement an update to a new configuration, the operator issues commands that install or uninstall individual forwarding rules on switches. By carefully constructing sequences of commands, it is possible to implement an atomic update on a single switch. For example, the operator can install a set of new rules at a lower priority than the current rules, and then delete the current rules using a single uninstall command. But it is not possible to implement simultaneous coordinated updates to multiple switches, as the network is a distributed system.

Distributed Firewall. We now present some simple examples of networks and updates that would be difficult to implement by hand, as motivation for the synthesis tool described in the following sections. The first example is a variant of one originally proposed by Reitblatt et al. [9]. The network topology, shown in Figure 1 (a), consists of an ingress switch I and three filtering switches F_1, F_2, F_3. For simplicity, assume that traffic flows up from the "world" to the "internal" network. At all times, the network is required to implement the following security policy: (i) traffic from authenticated hosts is allowed, (ii) web traffic from guest hosts is allowed, but (iii) non-web traffic from guest hosts is blocked.

Initially the network is configured so that the ingress switch I forwards traffic from authenticated hosts to F_1 and F_2 (which passes it through), and from guest hosts to F_3 (which performs the required filtering of non-web traffic). However, some time later, the network operator decides to transition to another configuration where traffic from authenticated hosts is processed on F_1 and traffic from guest hosts is processed on F_2 and F_3. Why might they want to do this? Perhaps there is more traffic from guests hosts than from authorized hosts, and the operator wishes to allocate more filtering switches to guest traffic to better handle the load.

Implementing this update correctly turns out to be surprisingly difficult. If we start by updating switches in an arbitrary order, we can easily end up in a situation where the security policy is violated. For example, if we update the ingress switch I to the new configuration without updating the filtering switches, then traffic from guest hosts will be forwarded to F_2, which will incorrectly pass it through to the internal network! One possible correct implementation is to first update I so it forwards traffic from authenticated hosts to F_1, wait until all in-flight packets have exited the network, update F_2 to filter non-web traffic, and finally update I again so that it forwards guest traffic to F_2 or F_3. Finding this sequence is not impossible, but would pose a significant challenge for the operator, who would have to reason about all of the intermediate configurations, as well as their effect on in-flight packets. By contrast, given encodings of the configurations and the intended security policy, our system generates the correct update sequence automatically.

Ring. The second example involves the network topology shown in Figure 1 (b). The network forwards packets around the ring until they reach their destination. For example, if we forward traffic clockwise around the ring, then a packet going from a host in N_1 to a destination host in N_3 might be forwarded from A to B to C. At all times, the network is required to be free of forwarding loops—that is, no packet should arrive back at the same port on a switch where it was previously processed. Initially the network forwards packets around the ring in the clockwise direction, as just described. Some time later, the network operator decides to reverse the policy so that traffic goes around the ring in the opposite direction. Implementing this update without introducing a forwarding loop is challenging. In fact, if we implement updates at the granularity of whole switch configurations, it is impossible! No matter which switch we update first, the adjacent switch will forward some packets back to it, thereby creating a loop. To implement the update correctly, we must carefully separate out the traffic going to each of networks N_1, N_2, and N_3, and transition those traffic classes to the counter-clockwise configuration one by one. Assuming the rules have this structure, our system generates the correct update sequence automatically.

Note that the examples discussed in this section both depend on updating individual rules on switches (rule granularity). However, the formal model used in the rest of this paper, only considers updates to whole switches (switch granularity). This is not a limitation: updates at rule granularity can be easily reduced to switch granularity by introducing an additional switch into the model for each rule. Our tool assumes that this reduction has already been performed.

3 Network Model

This section develops a simple abstract model of networks, and defines the network update synthesis problem formally. Our model is based on one proposed in previous work by Reitblatt et al. [9].

Topologies and packets. A *network topology* is a tuple $(\mathscr{S}, \mathscr{P}, \mathit{inport}, \mathit{outport}, \mathit{ingress})$, where $\mathscr{S}$ is a finite set of switches; $\mathscr{P}$ is a finite set of ports with distinguished ports *Drop* and *World*; $\mathit{ingress} \in 2^{\mathscr{P}}$ is a set of ingress ports; $\mathit{inport} \in \mathscr{P} \times \mathscr{S}$ is a relation such that for every port $p \in \mathscr{P} \setminus \{\mathit{World}, \mathit{Drop}\}$, there exists a unique switch $s \in \mathscr{S}$ with $\mathit{inport}(p,s)$; and $\mathit{outport} \in \mathscr{S} \times \mathscr{P}$ is a relation such that for every port $p \in \mathscr{P} \setminus (\mathit{ingress} \cup \{\mathit{World}, \mathit{Drop}\})$, there exists a unique switch $s \in \mathscr{S}$ with $\mathit{outport}(s,p)$. A *packet pt* is a finite sequence of bits. We assume that we can "read off" the values of standard header fields such as Ethernet and IP addresses and TCP ports, and we write *Packets* for the set of all packets. A *located packet* is a pair (p, pt), where p is a port and pt is a packet.

Policies and updates. The switches in the network make decisions about how to forward packets by examining their headers and the ingress ports on which they arrive. We model this behavior using switch policies: a *switch policy SwitchPol* is a partial function $\mathscr{P} \times Packets \rightharpoonup \mathscr{P} \times Packets$. A switch policy *SwitchPol* is *compatible* with a switch s if whenever *SwitchPol* is defined on (p,pt) and returns (p',pt') then $inport(p,s)$ and $outport(s,p')$. Note that real switches can forward packets out multiple ports. For simplicity, in this paper, we restrict our attention to linear traces of packets and only consider switch policies that generate at most one packet.

A *network policy NetPol* is a function $s \rightarrow SwitchPol$ where for all switches $s \in \mathscr{S}$, the switch policy $NetPol(s)$ is compatible with s. The *path* of a packet through the network is determined by the topology and network policy.

An *update* is a pair $(s, SwitchPol)$ consisting of a switch s and a switch policy *SwitchPol*, such that *SwitchPol* is compatible with s. Given a network policy *NetPol* and an update $(s, SwitchPol)$, the expression $NetPol[s \leftarrow SwitchPol]$ denotes a network policy $NetPol'$, where $NetPol'(s) = SwitchPol$ and $NetPol'(s') = NetPol(s')$ if $s' \neq s$. Note that an update only modifies the policy for a single switch.

Commands and states. A *command com* is either an update or the special command *wait*. A *wait* command models the pause between updates needed to ensure that packets that entered the network before the previous command will leave the network before the next command. Intuitively, waiting "long enough" makes sense only for network policies which force every packet to leave in a bounded number of steps. This is formalized below as the notion of *wait correctness*.

A *network state ns* is a tuple $(lp, NetPol, b_W, comSeq)$, where lp is a located packet, *NetPol* is a network policy, b_W is a Boolean modeling whether updates are enabled, and *comSeq* is a command sequence. Note that our model only includes a single packet in the network at any given time. As we are only interested in properties involving paths of individual packets through the network, intuitively, this is sufficient. However, we also need to be able to generate new packets at ingress ports at any given time during the execution of the network. One can prove (in a straightforward way) that this model is equivalent to a full model [9] with respect to LTL properties of paths of individual packets.

Traces. A *network transition* is a relation $ns \longrightarrow ns'$ on states. There are four types of transitions:

- A *packet move* if
$$\begin{aligned} ns &= ((p,pt), NetPol, b_W, comSeq) \\ ns' &= ((p',pt'), NetPol, b_W, comSeq) \end{aligned}$$
and $p \notin \{World, Drop\}$, where there exists a switch s such that $inport(p,s)$ and
$$NetPol(s)(p,pt) = (p',pt')$$
or $pt = pt'$ and $p = p' = World$ or $pt = pt'$ and $p = p' = Drop$.
- An *update transition* if
$$\begin{aligned} ns &= (lp, NetPol, false, (s, SwitchPol).comSeq) \\ ns' &= (lp, NetPol[s \leftarrow SwitchPol], false, comSeq) \end{aligned}$$
- A *wait transition* if
$$\begin{aligned} ns &= (lp, NetPol, b_W, wait.comSeq) \\ ns' &= (lp, NetPol, true, comSeq) \end{aligned}$$
The wait transition disables update transitions (by setting b_W to true), thus modeling the semantics of wait commands as explained above.

- A *new packet transition* if

$$\begin{array}{rcl} ns & = & ((p,pt),NetPol,b_W,comSeq) \\ ns' & = & ((p',pt'),NetPol,false,comSeq) \end{array}$$

where $p' \in ingress$. (Note that there is no condition on the new packet.) This transition models that, non-deterministically, we can decide to track a new packet.

A *network trace* nt is an infinite sequence of states $ns_0ns_1\ldots$ such that for all $i \geq 0$ we have that $ns_i \longrightarrow ns_{i+1}$. A network trace *initialized with* a policy *NetPol* and a command sequence *comSeq* is a network trace such that $ns_0 = (lp,NetPol,b_w,comSeq)$, for some located packet lp and Boolean b_w.

A *one-packet trace* $t = lp_0lp_1\ldots$ is a sequence of located packets that conforms to the network topology. That is, for all $i < |t|$, we have that if $lp_i = (p,pt)$ and $lp_{i+1} = (p',pt')$, then there exists a switch $s \in Switches$, such that $inport(p,s)$ and $outport(s,p')$. A *complete one-packet trace* is a finite trace such that $lp_0lp_1\ldots lp_n$ such that $lp_0 = (p,pt)$ where p is in *ingress* and $lp_n = (World,pt')$ or $lp_n = (Drop,pt')$.

A one-packet trace $t = lp_0lp_1\ldots lp_n$ is *contained* in a network trace $nt = ns_0ns_1\ldots$ if there is a function f (witnessing the containment) from $[0,n]$ to $\mathbb{N}$ with the following properties:

- for all $i \in [0,n-1]$, $f(i) < f(i+1)$;
- for all $i \in [0,n]$, we have that if $ns_{f(i)} = (lp,NetPol,b_W,comSeq)$, then $lp = lp_i$;
- for all $i \in [0,n-1]$, the transitions occurring between $f(i)$ and $f(i+1)-1$ in nt are only update and wait transitions, and the transition between $f(i+1)-1$ and $f(i+1)$ is a packet move transition.

A given network trace may contain traces of many packets generated by new packet transitions.

Wait and command correctness. A network state ns is *wait-correct* if, intuitively, the packet cannot stay in the network for an unbounded amount of time. Formally, ns is wait-correct if for all infinite network traces $nt = ns_0ns_1\ldots$ such that $ns_0 = ns$, and the transition from ns_0 to ns_1 is a wait transition, either there exists $i \in \mathbb{N}$ such that for all $j \in \mathbb{N}$ with $j > i$ the packet at ns_j is located at *Drop* or at *World*, or there exists $i \in \mathbb{N}$ such that the transition from ns_i to ns_{i+1} is a new packet transition.

The function $infin(cpt)$ appends an infinite suffix of the form lp_n^{ω} to the complete one-packet trace $cpt = lp_0lp_1\ldots lp_n$. Recall that a complete one-packet trace ends with the packet located at *Drop* or *World*, so $infin(cpt)$ models a packet staying outside of the network.

LTL. We now define LTL formulas and their semantics over infinite one-packet traces. Atomic formulas are of the form $packet = pt$ or $port = p$. A formula φ is an LTL formula, if it is an atomic formula, or is of the form $\neg\varphi_1$, $\varphi_1 \vee \varphi_2$, $X\varphi$, $\varphi_1 U \varphi_2$, where φ_1 and φ_2 are LTL formulas. As is standard, we will also use connectives F and G that can be defined in terms of the other connectives. Let t be an infinite one-packet trace $lp_0lp_1\ldots$. We have that $t \models packet = pt$ if there exists a port p such that $lp_0 = (pt,p)$. Similarly, we have that $t \models port = p$ if there exists a packet pt such that $lp_0 = (pt,p)$. The semantics of Boolean and temporal connectives is standard. An example of an LTL specification for the distributed firewall example is given in Figure 3 in Section 5.

Let φ be an LTL formula. A network trace nt satisfies an LTL formula φ (written $nt \models \varphi$) if for all complete one-packet traces cpt contained in nt, we have that $infin(cpt) \models \varphi$. Let $comSeq = com_0com_1\ldots com_{k-1}$ be a sequence of commands, and let *NetPol* be a network policy. A sequence of network policies $NetPol_0NetPol_1\ldots NetPol_n$ is induced by *comSeq* and *NetPol*, if

- $NetPol_0 = NetPol$
- for all i in $[0,k-1]$, if $com_i = wait$ then $NetPol_{i+1} = NetPol_i$
- for all i in $[0,k-1]$, if $com_i = (s,SwitchPol)$ then $NetPol_{i+1} = NetPol_i[s \leftarrow SwitchPol]$

We write $\mathit{NetPol} \overset{\mathit{comSeq}}{\longrightarrow} \mathit{NetPol}'$ if the last element of the sequence induced by *comSeq* is NetPol'. A command sequence *comSeq* is *correct* with respect to a formula φ and policy NetPol_i if for all network traces *nt* initialized with NetPol_i and *comSeq*, we have that *nt* is wait-correct and $nt \models \varphi$.

Update synthesis problem. With this notation in hand, we are now ready to formally state the *network update synthesis problem*. Given an initial network policy NetPol_i, a final network policy NetPol_f, and a specification φ, construct a sequence of commands *comSeq* such that:

- $\mathit{NetPol}_i \overset{\mathit{comSeq}}{\longrightarrow} \mathit{NetPol}_f$, and
- *comSeq* is correct with respect to φ and NetPol_i.

The next section develops an algorithm that solves this problem.

4 Update Synthesis Algorithm

This section presents an algorithm that synthesizes correct network updates automatically. The algorithm attempts to find a sequence of individual switch updates that transition the network from the initial configuration to the final configuration, while ensuring that the path of every packet traversing the satisfies the invariants specified by the operator.[1] It works by searching through the space of possible update sequences, but incorporates three important optimizations aimed at making synthesis more efficient.

Optimizations. The first optimization restricts the search space to solutions that update every switch in the network at most once. We call solutions with this property *simple*. Because the space of simple solutions is much smaller than the full space of solutions, this leads to a much more efficient synthesis procedure in practice.

The second optimization restricts the search space to solutions for which the synthesis procedure can efficiently check correctness. Because the network continues to process packets even as it is being updated, in general a packet may traverse the network during multiple updates. Hence, to ensure the correctness of the path of such a packet, it is necessary to check properties of sequences of network configurations, which can lead to an explosion of model checking tasks. We therefore introduce the notion of *careful* updates—update sequences where the system pauses between each step to ensure that all packets that were in flight before the step will have exited the network. There is one caveat worth noting: waiting only makes sense only for configurations for which every packet leaves a network after a bounded number of steps. To ensure this is possible, we require configurations to be loop-free in the sense that the policy has the property that every packet is processed by a given switch at most once. We thus have that every packet is in the network during at most one update. For such loop-free configurations, every packet either has a path using the configuration before a given step was applied, or the configuration after the step was applied. This enables us to check correctness of configurations separately. We do not need to check all possible configurations, we only need to check those encountered during the search.

The third optimization uses counterexamples to reduce the number of calls to the model checking procedure. The purpose of a call to the model checker is to check that all possible packet paths satisfy the specified invariants. However, if the model checker identifies a path that does not satisfy an invariant, the path is returned as a counterexample and can be used to eliminate future configurations quickly. In particular, any intermediate configurations in which the switches are configured in the same way as in the counterexample can be eliminated without having to consult the model checker.

[1] We assume that the topology is fixed, so that a network configuration is just a network policy.

```
Procedure OrderUpdate($NetPol_i$, $NetPol_f$, $\varphi$)
Input: Initial network policy $NetPol_i$, final network policy $NetPol_f$, and LTL specification $\varphi$.
Output: Simple and careful sequence of switch updates $L$, if it exists
if hasLoops($NetPol_i$) $\vee$ hasLoops($NetPol_f$) then
    return "Loops in initial or final configuration."
else
    $W \leftarrow$ false                                   ▷ Wrong configurations.
    $V \leftarrow$ false                                   ▷ Visited configurations.
    (ok, $L$) $\leftarrow$ DFSforOrder($NetPol_i$, $\bot$)
    if ok then
        return $L$
    else
        return "No simple and careful update sequence exists."

Procedure DFSforOrder($NetPol$, $cs$)
Input: Current network policy $NetPol$, most recently updated switch $cs$.
Output: Boolean ok if a correct update sequence exists; $L$ correct sequence of switch updates
if $NetPol = NetPol_f$ then
    return (true,[$NetPol$])                              ▷ Reached final configuration.
if $NetPol \models V$ then
    return (false,[])                                     ▷ Already visited $NetPol$.
$V \leftarrow V \vee NetPol$                              ▷ Add to visited configurations.
if $NetPol \models W$ then
    return (false,[])                                     ▷ Previous counterexample applies.
if $cs \neq \bot$ then                                    ▷ If there was a previous update,
    (ok,cex) $\leftarrow$ hasNewLoops($NetPol$,$cs$)      ▷ Check for forwarding loops.
    if ($\neg$ ok) then
        $W \leftarrow W \vee$ analyzeCex(cex)             ▷ Learn from loop counterexample.
        return (false,[])
(ok,cex) $\leftarrow$ ModelCheck($NetPol$,$\varphi$)
if ($\neg$ ok) then
    $W \leftarrow W \vee$ analyzeCex(cex)                 ▷ Learn from property counterexample.
    return (false,[])
for all ($NetPol_{next}, cs$) $\in$ NextPolicies($NetPol$) do    ▷ Try to update one more switch.
    (ok,$L$) $\leftarrow$ DFSforOrder($NetPol_{next}, cs$)       ▷ Recursive call.
    if ok then
        return (true,$NetPol$ :: $wait$ :: $L$)
return (false,[])
```

Figure 2: OrderUpdate Algorithm.

Algorithm. Figure 2 presents pseudocode for the ORDERUPDATE algorithm. It returns a sequence of careful and simple commands that implement the update correctly, or fails if no such sequence exists. The notions of simple and careful command sequences are defined formally below. We speak of sequences of commands, rather than sequences of updates, because we also include wait-commands for the reasons described above. The rest of this section describes the algorithm in detail and proves that it is sound and complete with respect to simple and careful command sequences. That is, if a simple and careful command sequence exists, then the algorithm will find it.

As we are interested only in simple command sequences, the main task is to find an order of switch updates. To do this, it uses a depth-first search, where at each recursive call, we update one switch. We consider only switches whose policy is different in the initial and final configurations. We opted for depth-first search as we expect that, in common cases, many update sequences will lead to a solution.

Before starting the search, we check that the initial and final configurations have no loops—otherwise a simple and careful sequence of commands does not exist. This is done by two calls to a function hasLoops. During the search, we check that each update that we encounter has not introduced new loops into the configuration. This is done in the auxiliary function hasNewLoops($\mathit{NetPol}_{next}, cs$), which takes as parameters the updated network policy and the switch that was updated. This check can be easily implemented using an LTL formula, as any new loop must pass through the updated switch.

The search maintains a formula V that encodes visited configurations, and a formula W that encodes the set of configurations excluded by counterexamples so far. The auxiliary function analyzeCex analyzes a counterexample, and outputs a formula representing the set of switches that occurred in the counterexample, and whether these switches were already updated.

If the current configuration was not visited before, and is not eliminated by previous counterexamples, we check whether all packet traces traversing this configuration satisfy the LTL specification φ. This is the purpose of the call to ModelCheck(NetPol, φ). In our implementation, we use NuSMV [2] as a back-end model checker.

If the current configuration passes all these tests, we continue the depth-first search, with next configurations being those where one more switch is updated. If we reach the final configuration, we pop out of the recursive calls, and prepend the corresponding updates (separated by wait commands) to the command sequence returned.

Soundness. Now we prove that ORDERUPDATE is sound. For the remainder of this section, let us fix a specific network topology $(\mathscr{S}, \mathscr{P}, \mathit{inport}, \mathit{outport}, \mathit{ingress})$.

A network policy *NetPol* satisfies an LTL formula φ (denoted by $\mathit{NetPol} \models \varphi$), if for all network traces *nt* initialized with *NetPol* and the empty command sequence, we have that $\mathit{nt} \models \varphi$. A network policy *NetPol* *induces* a one-packet trace t, if there exists a network trace *nt* initialized by *NetPol* and the empty sequence of commands, such that *nt* contains t.

A policy *NetPol* is *loop-free* if, intuitively, there is no loop in the graph given by the network topology and *NetPol*. More formally, for all sequences $w = p_0 s_0 p_1 s_1 \dots p_k s_k$ that conform to the network topology and to *NetPol*, we have that no port (and no switch) occurs more than once in w. A sequence $p_0 s_0 p_1 s_1 \dots p_k s_k$ conforms to the network topology and to *NetPol*, if for all $i \in [0, k-1]$, we have that $\mathit{inport}(p_i, s_i)$, and there exist packets pt and pt' such that $\mathit{NetPol}(s_i)(p_i, \mathit{pt}) = (p_{i+1}, \mathit{pt}')$.

Let $\mathit{comSeq} = \mathit{com}_0 \mathit{com}_1 \dots \mathit{com}_{n-1}$ be a command sequence, and let *NetPol* be a network policy. Let $\mathit{NetPol}_0 \mathit{NetPol}_1 \dots \mathit{NetPol}_n$ be the sequence of network policies induced by *comSeq* and *NetPol*. The command sequence *comSeq* is *careful* with respect to an LTL formula φ and a network policy *NetPol* if

- for all $i \in [0, n-1]$, if i is odd, then $\mathit{com}_i = \mathit{wait}$,

- for all $i \in [0,n]$, $NetPol_i$ is loop-free, and
- for all $i \in [0,n]$, $NetPol_i \models \varphi$.

Let *comSeq* be a careful sequence of commands. Let $NetPol_i$ be a network policy. Let $nt = ns_0 ns_1 \ldots$ be a network trace initialized with $NetPol_i$ and *comSeq*. Let $t = lp_0 lp_1 \ldots lp_n$ be a one-packet trace contained in *nt*. Let $\sigma = s_0 s_1 \ldots s_{n-1}$ be a sequence of switches such that $i < n$, we have that if $lp_i = (p, pt)$ and $lp_{i+1} = (p', pt')$, then $inport(p, s_i)$ and $outport(s_i, p')$. Our first lemma states that the path of every packet is affected by at most one update.

Lemma 1. *Let f be the function witnessing the containment of t in nt. There is at most one update transition in nt between $f(0)$ and $f(n)$.*

Proof. We use the fact that *comSeq* is careful, specifically that every command at an odd position in the sequence of commands is *wait*. Let us assume that there are two update transitions between $f(0)$ and $f(n)$ in *nt*. Let these two update transitions occur at network states ns_i and ns_j such that $f(0) \leq i < j < f(n)$. As *comSeq* is careful, we have that there is a wait-transition that occurs at ns_w, where $i < w < j$. As wait-transitions disable updates (by setting b_W to true; this is because the wait command models waiting long enough so that packets that entered the network before the previous update will leave the network before the next update), there has to be a new packet transition ns_p, where $w < p < j$. This contradicts the fact that t is contained in *nt*, which concludes the proof. □

The second lemma states that a path of every packet in the network could have occurred in of the intermediate configurations. That is, no packet takes a path non-existent in any of the configurations, even though the packet might be in-flight during the updates.

Lemma 2. *There exists $i \in \mathbb{N}$ such that ns_i induces t.*

Proof. We use the fact that *comSeq* is careful, specifically that *comSeq* is such that each $NetPol_i$ (for $0 \leq i \leq n$) is loop-free.

Let f be the function witnessing the containment of t in *nt*. By Lemma 1, we have that there is at most one update transition in *nt* between $f(0)$ and $f(n)$. Let the update transition be given by the update $(s, SwitchPol)$. Let ns_j $(f(0) \leq j < f(n))$ be the network state in which the update transition occurs. We show that either ns_j or ns_{j+1} induces t.

Now let us consider σ (defined above), which intuitively is the sequence of switches that a packet sees as it traverses the network. We analyze the following cases:

- s does not occur in σ. Then t was not influenced by the update, and is thus induced by ns_j.
- s occurs in σ, but only once. Let l be the smallest position in t such that there exist a port p and a packet pt such that $lp_l = (p, pt)$ and $outport(s, p)$. If $f(l)$ is less than j (i.e. the packet was at s before the update happened) then t is induced by $NetPol_j$. If $f(l)$ is greater than j, then t is induced by $NetPol_{j+1}$.
- s occurs more than once in σ. Let s_k s_{k+1} s_l be the subsequence of switches between two closest occurrences of s in Σ. As none of the switches in the subsequence were updated, we have that $NetPol_j$ or $NetPol_{j+1}$ is not loop-free, which contradicts the assumption that *comSeq* is careful.

This completes the proof. □

The third lemma states that carefulness (which is easily checkable) implies correctness.

Lemma 3. *If a command sequence comSeq is careful with respect to an LTL formula φ and a network policy NetPol, then comSeq is correct with respect to φ and NetPol.*

Proof. To show that *comSeq* is correct with respect to φ and *NetPol*, we need to show that for all network traces *nt* initialized with $NetPol_i$ and *comSeq*, we have that *nt* is wait-correct and $nt \models \varphi$. We first prove that *nt* is wait-correct. Let $nt = ns_0 ns_1 \ldots$ be a network trace, and let i be such that the transition from ns_i to ns_i is a wait-transition. We need to prove that for all infinite network traces that start at ns_i, and which do not contain a new packet transition, we have that the packet ends at the port *Drop* or *World* after a finite number of steps. Consider a network trace nt' that starts at ns_i and does not contain a new packet transition. Let us consider the unique one-packet trace t that starts at the last new packet transition before ns_i in *nt* (or which starts at the first position of *nt* if there is no new packet transition in *nt*), and continues as in nt'. Consider a prefix t' of t longer than the number of switches and ports in the network. By the proof of Lemma 2, t' is induced by ns_p, for p such that $p < i$. As *comSeq* is careful, we can conclude that t' is induced by a network state with a loop-free network policy, which means that the packet reaches *Drop* or *World* after a finite number of steps.

We now prove that $nt \models \varphi$. Let $nt = ns_0 ns_1 \ldots$ be a network trace and $cpt = lp_0 lp_1 \ldots lp_n$ be a complete one-packet trace contained in *nt*. We show that $infin(cpt)) \models \varphi$. By Lemma 2, we have that there exists $i \in \mathbb{N}$ such that *cpt* is induced by a ns_i. As *comSeq* is careful, we have that for all complete one-packet traces t' induced by ns_i, we have that $t' \models \varphi$. Therefore, we can conclude that $cpt \models \varphi$, and as there were no conditions on how *cpt* was chosen, we have that $nt \models \varphi$. This concludes the proof. □

Theorem 4 (Soundness). *Given an initial policy $NetPol_i$ a final policy $NetPol_f$, and an LTL formula φ,* ORDERUPDATE *returns a command sequence comSeq, then $NetPol_i \xrightarrow{comSeq} NetPol_f$, and comSeq is correct with respect to φ and $NetPol_i$.*

Proof. It is easy to show that if ORDERUPDATE returns *comSeq*, then $NetPol_i \xrightarrow{comSeq} NetPol_f$. Each update in the returned sequence changes a switch policy of one switch s to the policy $NetPol_f(s)$, and the algorithm terminates when all switches s such that $NetPol_i(s) \neq NetPol_f(s)$ have been updated. Let $NetPol_0 NetPol_1 \ldots NetPol_n$ be induced by *comSeq* and $NetPol_i$. We show that if ORDERUPDATE returns *comSeq*, then *comSeq* is careful with respect to φ and $NetPol_i$. To prove that $comSeq = com_0 com_1 \ldots com_{n-1}$ is careful, we show that:

- for all $j \in [0, n-1]$, if j is odd, then $com_j = wait$. One can simply observe that this is true, given how the sequence of updates is constructed in the algorithm (Line 30).
- for all $j \in [0, n]$, $NetPol_j$ is loop-free. This holds, as we check that the initial configuration is loop-free, and that each update does not introduce a loop (Line 19).
- for all $j \in [0, n]$, $NetPol_j \models \varphi$. This is ensured by the call to a model checker (Line 23).

Finally we can use Lemma 3 to infer that *comSeq* is careful with respect to φ and $NetPol_i$. □

Completeness. The ORDERUPDATE algorithm is also complete with respect to simple and careful command sequences. Let $comSeq = com_0 com_1 \ldots com_{n-1}$ be a command sequence. Such a sequence is *simple* if for each $s \in \mathcal{S}$ there exists at most one i in $[0, n]$ such that com_i is an update of the form $s, SwitchPol$. The proof (omitted here) uses the fact that ORDERUPDATE searches through all such sequences (more precisely, all such sequences that do not use multiple *wait* commands in a row). The following proposition characterizes the cases where the algorithm returns a solution.

Proposition 5. *Given an initial network policy $NetPol_i$, a final network policy $NetPol_f$, and a specification φ, if there exists a simple and careful sequence of commands comSeq such that $NetPol_i \xrightarrow{comSeq} NetPol_f$, then* ORDERUPDATE *returns one such sequence.*

5 Implementation

We have built an implementation of ORDERUPDATE in OCaml. The functions ModelCheck$(\mathit{NetPol}, \varphi)$ and hasNewLoops(NetPol, cs) are implemented by calling out to the NuSMV [2] model checker on suitable encodings of the network configuration. More specifically, the function hasNewLoops$(\mathit{NetPol}_{next}, cs)$ takes as parameters the updated network policy and the switch that was updated, and checks that no new loops were introduced by the update. This check can be performed using the LTL formula $G(cs \rightarrow \neg X(F\ cs))$, as any newly introduced loops must pass through the updated switch.

NuSMV models. The NuSMV encodings of network configurations are similar to the formal model described in Section 3: Packets are represented as tuples consisting of `src`, `dst`, and `purpose`, where `src` is source of the packet (e.g., a "guest" host), `dst` is the destination of the packet, and `purpose` is a general field (e.g. "Web traffic"). Switch policies are encoded as NuSMV expressions over these variables (`src`, `dst`, and `purpose`) as well as ingress ports. The model has a single entry point—a port `Start` from which a packet can transition to an ingress port on any switch. Finally, as in Section 3, we reduce the size of the NuSMV input by transitioning located packets to the next ingress port after forwarding—i.e., we inline the links between the output port on one switch and the ingress port at another. Figure 3 gives the NuSMV encoding of the initial configuration for the firewall example from Section 2.

Rule granularity. Recall that we represent switch policies as partial functions, and we model updates that apply at the granularity of whole switches. Of course, in real switches, policies are represented using *rules* that "match" the domain of the function, and the switch forwards packets according to the best matching rule. Hence, it is important to be able to encode finer-grained updates that only modify particular rules on switches—indeed, such updates are used in both of the motivating examples from Section 2. Fortunately, rule granularity can be easily reduced to switch granularity: we transform the switch into a sequence of switches, where each switch forwards packets matched by one rule, and passes all unmatched packets along to the next switch. We use this technique in many of our examples.

Other algorithms. Besides the ORDERUPDATE algorithm, we have also implemented two additional algorithms for comparison purposes. The REFINE algorithm provides a direct implementation of a counterexample-guided synthesis approach to our problem. In this approach, we add a Boolean variable for each switch to model whether the switch has updated or not. We allow switches to update as the packet traverses the network, with no more than one switch updating per new packet transition. We use counterexamples learned from NuSMV to refine our model, explicitly preventing the update order appearing in the counterexample. The process continues until either the final configuration cannot be reached or any sequence of updates possible in the refined model is safe.

The CONFIGPAIRS algorithm has the same structure as the ORDERUPDATE algorithm, but includes an additional Boolean variable for the switch being updated. This variable models whether the switch has updated or not. We allow the switch to update at any time, including while the packet traverses the network. In effect, there is a model checking call for each pair of configurations in the worst case (as opposed to a call per configuration). This is because the algorithm in the preceding section relies on Lemmas 1 and 2, rather than on checking pairs of configurations.

```
MODULE main
VAR
    port : {I_0, F1_0, F2_0, F3_0, START, WORLD, DROP};
    src : {Auth, Guest};
    purpose : {Web, Other};
ASSIGN
    next(port) := case
        port = START : I_0;
        port = I_0 & src = Auth : {F1_0, F2_0};
        port = I_0 & src = Guest : F3_0;
        port = F1_0 : WORLD;
        port = F2_0 : WORLD;
        port = F3_0 & purpose = Web : WORLD;
        port = F3_0 & purpose = Other : DROP;
        port = WORLD : WORLD;
        port = DROP : DROP;
    esac;
    next(src) := src;
    next(purpose) := purpose;
INIT port = START;
LTLSPEC G (purpose = Other & src = Guest -> F port = DROP) &
          ((src = Auth | src = Guest & purpose = Web) -> F port = WORLD);
```

Figure 3: NuSMV encoding of firewall example.

6 Experiments

To evaluate the effectiveness of our implementation, we used it to generate update sequences for several examples. To provide a comparison, we compared our main OrderUpdate algorithm to our own implementations of the (simpler) Refine and ConfigPairs algorithms.

Goals. The most important parameters of the network update problems are N, the total number of switches in the network, and M, the number of switches whose switch policy differs between initial and final configuration. Note that the size of the solution space is $M!$. The goal of our experimental evaluation is to quantify how our tool scales with growing M and N, both for problems where a solution exists and for problems where the solution does not exist. We believe that an important class of network update problems that occurs in practice is when N is on the order of 1000, and M is on the order of 10—such updates arise when there is a problem on a small number of nodes and the network must route around it.

Benchmarks. We ran our tests on specific network configurations, parameterized by N and M. The topology of the network, depicted in Figure 4 (a), is as follows: the network has an inner part consisting of a sparse but connected graph, and an outer part with a larger number of nodes and ingresses reachable in two hops. In the experiments, we removed several of the switches in the inner part of the network while maintaining connectivity, so that at all times each ingress port is reachable from the other two. Intuitively, this experiment could model taking down switches for maintenance. The two policies are computed using shortest-path computations before and after the switches are removed. This experiment allows us to both scale the inner part, increasing the number of switches that differ between the policies, and also scale the total number of switches by increasing the number of switches in the outer parts.

Results. We ran our experiments using a laptop machine with a 2.2 GHz Intel processor and 4 GB RAM. We used NuSMV version 2.5.4 as the external model checker.

Scaling network size: The first experiment tests how our tool scales with N (the total number of nodes). We fixed the number of nodes updating at 13 and ran the tool on graphs of size 100, 250, 500, and 1000. We ran each experiment using the ORDERUPDATE algorithm discussed in Section 4, as well as REFINE and CONFIGPAIRS algorithms described in Section 5. The REFINE implementation failed on the two larger inputs. The results are reported in Figure 4 (b).

Scaling update size: The next experiment tests how our tool scales with M (the number of nodes updating). In this experiment, we held N (the total number of nodes) fixed at 500 and ran the tool with the total number of nodes updating between 5 and 15. We show the results for ORDERUPDATE algorithm only, as the above experiments show that the other two do not perform well with 500 nodes. The results are reported in Figure 4 (c).

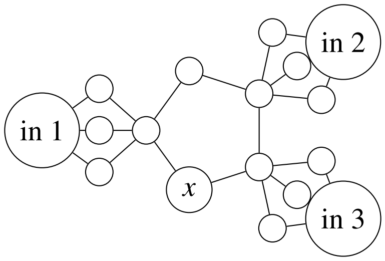
Impossible updates: The final experiment tests how our tool performs on impossible updates—i.e., updates for which no safe and careful sequence of switch updates exists. We modified the benchmark slightly so that in the final configuration, the ingress switches drop packets destined for them instead of forwarding them out to the world. In this experiment, we used updates that affected 8 of the nodes. The results of this experiment are shown in Figure 4 (d). We also report how the tool performs without counterexample analysis here (and not in the previous tables), as counterexamples are most helpful when there are many incorrect configurations. It is interesting to note that although REFINE does not scale as well to large numbers of nodes, it is able to quickly determine when an update is impossible.

Summary. Overall, our experiments show that our tool scales to the class of network updates problems outlined above. For a network with $N = 1000$ nodes, $M = 13$ of which need to be updated, the running time is 18 minutes. Our tool also scales for a larger number of nodes updating. For 500 nodes total, and 30 nodes updating, the running time is 10 minutes. These running times are far too large for online use by network operators, but we emphasize that we report on a prototype tool—our primary goal was to confirm feasibility of our approach. We leave building a well-engineered tool to future work. We note that if it is not possible to find an update, the tool takes much longer to complete. This is because the tool needs to go through a large number of possible update sequences. Here, our counterexample analysis helps significantly, reducing the running time for the case $N = 500$, $M = 8$ by 85%. However, the tool does not scale well with M in impossible updates; with $M = 10$ this example ran for over 45 minutes.

7 Related Work

Network updates are a form of concurrent programming. Synthesis for concurrent programs has attracted considerable research attention in recent years [10, 13, 1, 12]. In work by Solar-Lezama et al. [10] and Vechev et al. [12], an order for a given set of instructions is synthesized, which is a task similar to ours. However, the problem settings in the traditional synthesis work and in this paper are quite different. First, traditional synthesis is a game against the environment which (in the concurrent programming case) provides inputs and schedules threads; in contrast our synthesis problem is a reachability problem

(a)



(b)

Algorithm	100 Nodes	250 Nodes	500 Nodes	1000 Nodes
ORDERUPDATE	10	83	355	1101
CONFIGPAIRS	129	1244	3731	12077
REFINE	55	267	Out of memory	Out of memory

(c)

Nodes	5	6	7	8	9	10	11	12	13	14	15	30	60
Time	165	142	166	222	222	205	273	276	354	339	370	611	2106

(d)

Algorithm	100 Nodes	250 Nodes	500 Nodes	1000 Nodes
ORDERUPDATE	19	170	900	3963
ORDERUPDATE w/o counterexamples	101	1793	6269	Timeout
REFINE	20	101	Out of memory	Out of memory

Figure 4: Experiments: (a) topology, (b) scaling network size, (c) scaling update size, (d) impossible updates. All times are in seconds.

on the space of configurations. Second, the space of network configurations is very rich; determining whether a configuration is false is an LTL model checking problem by itself.

Update mechanisms have also been studied in the networking community. This paper builds on previous work on consistent updates by Reitblatt et al. [9]. However, unlike our tool, which allows operators to specify explicit invariants, consistent updates preserve *all* path properties. This imposes a fundamental overhead as certain efficient updates that are produced by our tool would not be valid as consistent updates. Another line of work has investigated update mechanisms that minimize disruptions in specific routing protocols [5, 4, 8, 11, 7]. However, these methods are tied to particular protocols such as BGP, and only guarantee basic properties such as connectivity. In particular, they do not allow the operator to specify explicit invariants.

8 Conclusion

Network updates is an area where techniques developed for program and controller synthesis could be very beneficial for state-of-the-art systems. There are several possible directions for future work. We plan to investigate further optimizations that could bring down the running time on realistic networks from minutes to seconds, improving usability. We also plan to investigate the network update problem with environment changing while updates are executed, leading to two-player games. It would also be interesting to abstract the structure of the network and apply parametric synthesis techniques, and to explore techniques that incorporate considerations of network traffic, using ideas from controller synthesis. Another interesting direction is to investigate algorithms that rank updates and select the "best" one when there are multiple correct updates. Finally, we would also like to extend our tool to provide guarantees about properties involving sets of packets (such as per-flow consistency from Reitblatt et al. [9]), and about properties concerning bandwidth and other quantitative resources.

Acknowledgments. We wish to thank the SYNT reviewers, Arjun Guha, and Mark Reitblatt for helpful comments and suggestions. Our work is supported in part by NSF under grants CNS-1111698, CCF-1253165, and CCF-0964409; ONR under award N00014-12-1-0757; by a Google Research Award; and by a gift from Intel Corporation.

References

[1] S. Cherem, T. Chilimbi & S. Gulwani (2008): *Inferring locks for atomic sections*. In: PLDI, pp. 304–315, doi:10.1145/1375581.1375619.

[2] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani & A. Tacchella (2002): *NuSMV 2: An OpenSource Tool for Symbolic Model Checking*. In: CAV, pp. 359–364, doi:10.1007/3-540-45657-0_29.

[3] Pierre Francois & Olivier Bonaventure (2007): *Avoiding transient loops during the convergence of link-state routing protocols*. IEEE/ACM Trans. on Networking, doi:10.1109/TNET.2007.902686.

[4] Pierre Francois, Pierre-Alain Coste, Bruno Decraene & Olivier Bonaventure (2007): *Avoiding disruptions during maintenance operations on BGP sessions*. IEEE Trans. on Network and Service Management, doi:10.1109/TNSM.2007.021102.

[5] Pierre Francois, Mike Shand & Olivier Bonaventure (2007): *Disruption-free topology reconfiguration in OSPF Networks*. In: INFOCOM, doi:10.1109/INFCOM.2007.19.

[6] John P. John, Ethan Katz-Bassett, Arvind Krishnamurthy, Thomas Anderson & Arun Venkataramani (2008): *Consensus Routing: The Internet as a Distributed System*. In: NSDI.

[7] Nate Kushman, Srikanth Kandula, Dina Katabi & Bruce M. Maggs (2007): *R-BGP: staying connected In a connected world*. In: NSDI.

[8] S. Raza, Y. Zhu & C-N. Chuah (2011): *Graceful Network State Migrations*. IEEE/ACM Transactions on Networking 19(4), doi:10.1109/TNET.2010.2097604.

[9] Mark Reitblatt, Nate Foster, Jennifer Rexford, Cole Schlesinger & David Walker (2012): *Abstractions for Network Update*. In: ACM SIGCOMM Conference on Communications Architectures, Protocols and Applications (SIGCOMM) , Helsinki, Finland, pp. 323–334, doi:10.1145/2342356.2342427.

[10] A. Solar-Lezama, C. Jones & R. Bodík (2008): *Sketching concurrent data structures*. In: PLDI, pp. 136–148, doi:10.1145/1379022.1375599.

[11] Laurent Vanbever, Stefano Vissicchio, Cristel Pelsser, Pierre Francois & Olivier Bonaventure (2011): *Seamless Network-Wide IGP Migration*. In: SIGCOMM, doi:10.1145/2018436.2018473.

[12] M. Vechev & E. Yahav (2008): *Deriving linearizable fine-grained concurrent objects*. In: PLDI, pp. 125–135, doi:10.1145/1375581.1375598.

[13] M. Vechev, E. Yahav & G. Yorsh (2010): *Abstraction-guided synthesis of synchronization*. In: POPL, pp. 327–338, doi:10.1145/1706299.1706338.