CURSOR: Configuration Update Synthesis Using Order Rules

Zibin Chen, Lixin Gao
Department of Electrical and Computer Engineering, University of Massachusetts, Amherst
{zibinchen, lgao}@engin.umass.edu

***Abstract*—Network configuration updates are frequent nowadays to adapt to the rapid evolution of networks. To ensure the safety of the configuration update, network verification can be used to verify that network properties hold for the new configuration. However, configuration updates typically involve multiple routers changing their configurations. Changes on these routers cannot be applied simultaneously. This leads to intermediate configurations, which might violate network properties such as reachability. Configuration updates synthesis aims to find an order of applying changes on routers such that network properties hold for all intermediate configurations.**

Existing approaches synthesize a safe update order by traversing the update order space, which is time-consuming and does not scale to a large number of configuration updates. This paper proposes CURSOR, a configuration update synthesis that extracts rules that update orders should follow. We implement CURSOR and evaluate its performance with real-world configuration update scenarios. The experimental results show that we can accelerate the synthesis by an order of magnitude on large-scale configuration updates.

I. Introduction

Network configurations are updated frequently due to the rapid development of the network as well as the need to enhance network security. The process of updating configurations to networks is known as the configuration update process. Configuration updates to backbone networks of large ISPs and content providers occur frequently. Google has revealed that its network configuration is constantly changing to accommodate the increasing traffic demands and to deploy new features [16]. Facebook changes its network configurations as frequently as weekly [31]. To guarantee the safety of the network, operators have to ensure that certain network properties (such as pairwise reachability and loop-free) hold at all times.

Control plane network verifications [1], [4], [14] are proposed to ensure the safety of the network for a particular configuration. Network operators can use control plane verification tools to check network properties on the new configuration and ensure the new configuration does not cause outages or security issues.

However, configuration updates often involve multiple routers updating their configurations. Changes to different routers cannot be applied simultaneously. This leads to intermediate update states where only a subset of routers changed their configurations. These intermediate states are not guaranteed to be safe from outages or security issues even though the new configuration is verified and proven safe. As a result, some orders of applying updates might lead to temporarily loss of connectivity or security issues. Even several minutes of violations of network properties can be harmful and incur substantial economic loss [21]. In addition to verifying network properties on the new configuration, network operators have to decide an order of applying changes such that all intermediate configurations satisfy the network properties.

Configuration update synthesis automatically discovers an order of applying updates such that all intermediate update states satisfy the network properties. One obvious way to derive a safe order of applying changes to routers is to enumerate all possible orders and return the one that satisfies the network properties at all times (i.e., all intermediate update states). The enumeration-based approach works for configuration updates with a small number of changes. However, real-world configuration updates might involve a large number of changes. For example, changing from a fully meshed iBGP architecture to iBGP with route reflection for a network with hundreds of routers can involve more than 10,000 changes. As a result, the enumeration-based approach can lead to enormous search space and does not scale to large number of updates.

Existing work such as Snowcap [27] proposed counter-example guided search [25], [37] to reduce the search space. While it can speed up the configuration update synthesis, even traversing partial update order space can be time-consuming.

This paper proposes CURSOR, a synthesis scheme for configuration updates. The key idea of CURSOR is to directly derive rules for update orders instead of trying out update orders. In this paper, we express the best routes in terms of update states. As a result, it is possible to describe network properties with a boolean expression of update states, from which CURSOR derives rules for update orders. The contributions of this paper are summarized as follows.

- We express network properties with a boolean expression of update states. This is done by deriving the best routes in terms of update states. Since network properties are typically expressed in terms of the best routes, we can find the equivalent boolean expression in terms of update states.
- We construct the rules for update orders by analyzing the boolean expression of update states. We can come up with rules for update orders to avoid entering update states that violate network properties.
- We synthesize a safe order of applying updates based on the rules. We propose heuristic algorithms to perform

level-based synthesis by allocating updates to different levels based on the derived rules.
- We implement and evaluate CURSOR under large-scale real-world configuration update scenarios. The experimental results show that CURSOR can reduce the synthesis time by one order of magnitude compared to the state-of-art approach.

The rest of the paper is organized as follows. Section II shows that improper order of applying updates can lead to violations of the network properties. Section III gives an overview of CURSOR with a running example. Section IV describes how we express the network properties with update states. We use the expression of network properties and construct rules for update orders in Section V. Section VI introduces level-based synthesis, where we derive an update order such that network properties hold at all times during the configuration update. We evaluate CURSOR in Section VII and discuss the related work in Section VIII. We conclude in Section IX.

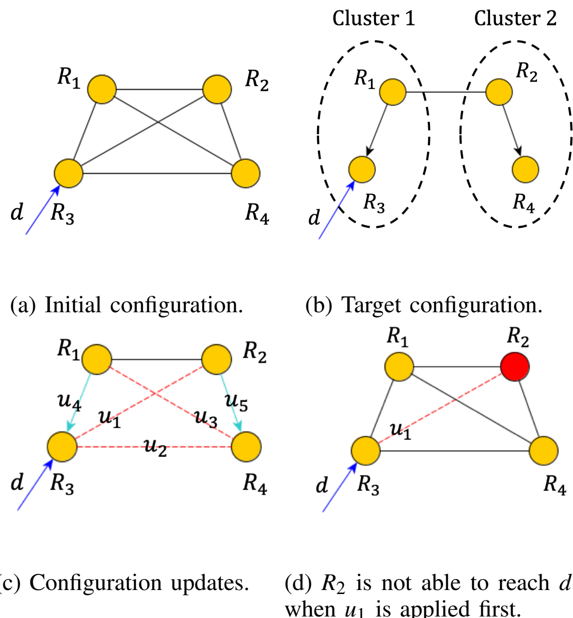
II. Motivation

In this section, we motivate our work by showing a configuration update example where applying changes in a specific order can lead to violations of network properties. Then, we show that deriving a valid sequence of updates through enumerating the order space (i.e., all possible order of applying updates) is not feasible.

We first show that network properties can be violated during the configuration update process. Consider the following real-world reconfiguration to the iBGP routing system: transiting from a fully meshed iBGP to iBGP with route reflection. Fully meshed iBGP is the simplest iBGP configuration used to redistribute routes learned from eBGP sessions. Fully meshed iBGP requires each pair of BGP routers to establish iBGP sessions. However, as the network grows, the number of iBGP sessions grows. Networks with a large number of BGP routers consider iBGP with route reflection [6] to reduce the number of iBGP sessions. BGP routers form a hierarchical structure with *route-reflector-to-client* sessions. Figure 1 illustrates the reconfiguration of the iBGP routing system where the blue arrow indicates routes to external prefix d learned from eBGP sessions. Solid black edges with arrows represent a route-reflector-to-client session where arrows point to the client.

Let us focus on the reachability of prefix d of all routers. Suppose we apply the updates with the order as they are listed (i.e., $u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4 \rightarrow u_5$). R_2 no longer receives the route when u_1 is applied first (see Figure 1 (d)) since BGP routers do not announce routes learned from a non-client iBGP neighbor to another.

One simple yet inefficient way to avoid the violation of network properties during the reconfiguration is to traverse the order space (i.e., all possible orders) and verify the network properties at all intermediate update states. An order is considered safe if the network properties hold at all intermediate update states. However, the number of possible ordering for n updates is $n!$. Given the scalability challenge on control plane



(a) Initial configuration. (b) Target configuration.

(c) Configuration updates. (d) R_2 is not able to reach d when u_1 is applied first.

Fig. 1: iBGP configuration migrating from fully meshed iBGP to route reflection.

verification, traversing the order space can be even more time-consuming and is not feasible when there is a large number of updates.

III. Overview of CURSOR

In this section, we give an overview of CURSOR: a rule-based configuration update synthesis scheme. The key idea is to generate rules for update orders based on the network properties.

CURSOR synthesizes the update order by deriving rules for update orders. In order to determine such rules, CURSOR represents the network properties with a boolean expression of update states. Then, rules for update orders can be derived from the boolean expression. After obtaining the rules, we can synthesize an order of applying updates.

Now, we show how to express network properties with update states. Consider the example in Figure 1. We aim to derive a boolean expression of update states where the reachability is *violated*. We encode update states with a set of boolean variables v_i $(1 \leq i \leq n)$, where v_i indicates that the ith update (i.e., u_i) is applied, and $\overline{v_i}$ indicates that update u_i is not applied.

Let us first focus on R_2 and consider that R_2 cannot reach d. That is, R_2 does not receive any routes from R_1 and R_3 to d. Consider R_2 does not receive a route from R_1. This means R_1's best route is received from a non-client peer. That is, R_3 is a non-client peer of R_1, which corresponds to the boolean expression $\overline{v_4}$. Consider the case that R_2 does not receive a route from R_3. The iBGP session with R_3 is disconnected, which can be expressed as a boolean expression v_1. Then the violation of reachability of R_2 can be expressed as the boolean expression $v_1 \wedge \overline{v_4}$. Similarly, we derive boolean expressions for other routers. CURSOR expresses the network properties

with the best routes and then converts them into boolean expressions of update states. Section IV shows how to express network properties with a boolean expression of update states.

We then construct the rules from the boolean expressions of update states. The rules ensure that the network properties hold for all intermediate states. We consider the opposite (i.e., which intermediate states violate the network properties) and use rules to avoid entering those update states. Intermediate update states for an update order $u_{\pi(1)} \rightarrow \cdots \rightarrow u_{\pi(n)}$ are $(v_{\pi(1)}, \overline{v_{\pi(2)}}, \dots, \overline{v_{\pi(n)}})$, $(v_{\pi(1)}, v_{\pi(2)}, \dots, \overline{v_{\pi(n)}})$, $\dots$, $(v_{\pi(1)}, v_{\pi(2)}, \dots, v_{\pi(n-1)}, \overline{v_{\pi(n)}})$. The rules have to guarantee that the boolean expression holds *false* for all intermediate update states listed above. We transform the boolean expression into Disjunctive Normal Form (DNF) and solve the encodings of the update states that violate the network properties. DNF consists of disjunctive clauses connected with *and*. Network properties are violated if any clause is true. Each clause is a conjunction of either positive literals (e.g., v_1) or negative literals (e.g., $\overline{v_4}$). The key idea is to ensure that there is at least one item in the clause is false by setting variables in negative literals to true. As a result, we apply updates in negative literals before updates in positive literals so that the clause is false for all intermediate states. For example, the rule for the clause $v_1 \wedge \overline{v_4}$ is that u_4 has to be applied before u_1. Similarly, we can derive rules from other clauses. In Section V, we show the derivation of rules from boolean expressions of update states in detail.

We synthesize an order that satisfies all of the derived rules. In Section VI, we propose a heuristic algorithm to synthesize the update order automatically.

IV. Expressing Network Properties with Update States

This section shows how we express network properties with update states. Network properties are typically expressed with best routes. As a result, we first express the best routes with update states. In this section, we derive the best routes from update states in Section IV-A and Section IV-B. After that, we combine the best routes and express network properties with update states in Section IV-C.

A. Derive IGP Best Routes from Update States

We first derive the best routes for Internal Gateway Protocol (IGP). IGP, like OSPF [23] and IS-IS [30], uses shortest-path-based routing. That is, given the update state, the best route of each router is the route with the lowest cost. We can use the Bellman-Ford algorithm to calculate the best routes under given update states. A naive approach is to run the algorithm for each update state and get the best routes. However, one needs to enumerate 2^n update states for a configuration update with n updates.

The above approach assumes that the best route depends on every update. However, this is not necessarily true. The best route of a router typically depends on only a few updates. We show an example in which the best routes depend on a subset of updates in Figure 2. In Figure 2, we have two updates: u_1

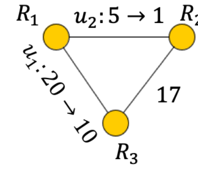


Fig. 2: R_2's smallest cost to R_1 depends only on u_2 not on u_1.

changes the IGP link weight between R_1 and R_3 from 20 to 10 and u_2 changes the IGP link weight between R_1 and R_2 from 5 to 1. Note that the cost of using the direct link to R_1 for R_2 is at most 5, while using R_3 to forward the traffic can end up with a cost of at least 27. As a result, R_2 always uses the direct link no matter whether update u_1 is applied or not. In this case, R_2's best route to R_1 does not depend on u_1.

We propose a Bellman-Ford-like algorithm to derive the best routes under configuration updates such that only updates affecting the best routes are considered. Like the Bellman-Ford algorithm, routes are propagated between neighboring routers, and each router updates its route when a better one is found. Unlike the Bellman-Ford algorithm, the route for each router consists of a set of costs[1], each of which is quantified by a set of update states. For example, in Figure 2, the best route for R_2 to R_1 can be expressed as $\{(1: *, true), (5: *, false)\}$, where 1 and 5 are the costs, and $*, true$ and $*, false$ are the quantified update states. As described above, the best route does not depend on u_1. As a result, we use $*$ to represent that no matter what value v_1 is, the costs are the same. More generally, we use $(c: b_1, \dots, b_n)$ to represent an entry in a route where c is the cost and b_1 to b_n are the values of the corresponding update variables.

We iteratively update routes on every router until convergence. Initially, routes are set to $(0: *, \dots, *)$ if the router originates the prefix and $(\infty: *, \dots, *)$ otherwise. The update process on each router consists of two iterative steps: PropagateRoute and MergeRoute.

1) *PropagateRoute:* R_i propagates its route through the link between R_i and R_j;
2) *MergeRoute:* R_j merges the propagated routes to its route.

We keep doing PropagateRoute and MergeRoute for all pairs of neighboring routers until no router updates its route in one iteration. We describe PropagateRoute and MergeRoute in detail in the following.

PropagateRoute: The route of a router is propagated to a neighbor through a link with potential update. The update on the link can change the weight, resulting in different link weights under different update states. As a result, proper link weight should be added to the proper cost when a route is propagated to neighbors. We consider the following two cases.

Case 1: The link does not have an update: The link weight keeps the same under all update states. Therefore, we

[1]For simplicity of exposition, we describe the algorithm for one specific destination. Different destinations are calculated independently and have independent best routes.

Algorithm 1 PropagateRoute.

1: **procedure** PROPAGATEROUTE(r,l)
2: Initialize empty route r_{prop} ▷ r_{prop} is the propagated route.
3: **for** entry of $(c:s)$ in r **do**
4: **if** link has update u_i **then**
5: Suppose weight is w_{new} when $u_i = 1$
6: Suppose weight is w_{old} when $u_i = 0$
7: $b_1,\ldots,b_i,\ldots,b_n \leftarrow s$
8: **if** $b_i = 1$ **then**
9: Add entry $(c+w_{new} : s|_{v_i})$ to r_{prop}
10: **else if** $b_i = 0$ **then**
11: Add entry $(c+w_{old} : s|_{\overline{v_i}})$ to r_{prop}
12: **else if** $b_i = *$ **then**
13: Add entry $(c+w_{new} : s|_{v_i})$ to r_{prop}
14: Add entry $(c+w_{old} : s|_{\overline{v_i}})$ to r_{prop}
15: **end if**
16: **else**
17: $w \leftarrow$ link weight of l
18: Add entry $(c+w:s)$ to r_{prop}
19: **end if**
20: **end for**
21: **return** r_{prop}
22: **end procedure**

Algorithm 2 MergeRoute

1: **procedure** MERGEROUTE(r_p,r_t)
2: **if** $\min\{c_p|(c_p:s_p)\in r_p\} \geq \max\{c_t|(c_t:s_t)\in r_t\}$ **then**
3: Absorb merge
4: **end if**
5: **for** each $(c_t:s_t)\in r_t$ **do**
6: **for** each $(c_p:s_p)\in r_p$ **do**
7: **if** $\exists u_e \in s_p \wedge s_t, s_p[u_e] \neq s_t[u_e]$ **then**
8: **continue**
9: **end if**
10: **if** $c_p < c_t$ **then**
11: **if** $s_p = s_t$ **then** ▷ Case 1
12: $c_t \leftarrow c_p$
13: **else** ▷ Case 2
14: $s_{int} \leftarrow s_t \cap s_c$ ▷ See Equation (1)
15: Remove entry $(c_t:s_t)$ from r_t.
16: Add entry $(c_p:s_{int})$ to r_t.
17: Add entry $(c_t:s_t\backslash s_{int})$ to r_t.
18: **end if**
19: **end if**
20: **end for**
21: **end for**
22: **end procedure**

go over all entries in the route and add the link weight to the cost. We propagate the route after adding the link weight.

Case 2: The link has an update, u_i: Suppose update u_i changes the link weight from w_{old} to w_{new}. We add w_{old} or w_{new} to costs according to the quantified update state. We go over each entry $(c:b_1,\ldots,b_i,\ldots,b_n)$ in the route and consider the following three cases of b_i.

- $b_i = true$: We add w_{new} to c.
- $b_i = false$: We add w_{old} to c.
- $b_i = *$: We generate two entries for v_i and $\overline{v_i}$ and add w_{new} and w_{old}, respectively. More specifically, we split the entry into two: $(c+w_{new}:b_1,\ldots,true,\ldots,b_n)$ and $(c+w_{old}:b_1,\ldots,false,\ldots,b_n)$.

We propose PROPAGATEROUTE(r,l), which propagates route r through link l. PROPAGATEROUTE(r,l) takes a route r and a link l as input and returns a new route r_{prop} to be propagated to the neighbor. We show PROPAGATEROUTE(r,l) in Algorithm 1.

MergeRoutes: The propagated route updates the neighbor's route (target route). Both the propagated and the target routes have multiple entries. In order to merge these two routes, we consider whether the propagated route brings a better route to the destination (i.e., a route with a smaller cost). We merge the propagated route into the target route only when a smaller cost in the propagated route is found. We do merge on each pair of entries.

Let us focus on one entry from the propagated route $(c_p:s_p)$ and merge it into an entry in the target route $(c_t:s_t)$. One obvious thing to do is to compare the cost: merge only happens when $c_p < c_t$. We skip this pair of entries if $c_p \geq c_t$.

We then consider whether s_t and s_p overlap. That is, whether two entries cover any common update states. s_p and s_t are considered not covering common update states if there exists an update u_i such that

- $s_p[u_i] \neq *$ and $s_t[u_i] \neq *$ *and*;
- $s_p[u_i] \neq s_t[u_i]$.

If s_t and s_p do not overlap, the propagated route does not convey shortest path information for any common update states with the entry in the target route. There is no need to merge these two pairs of entries.

If s_t and s_p overlap, we update the cost of the overlapped update states. We consider two cases.

Case 1: $s_p = s_t$: We update c_t to c_p.

Case 2: $s_p \neq s_t$: In this case, we split s_t into two parts by whether the state is covered by s_p. One part is the intersection of s_t and s_p. We denote the intersection update states as s_{int}. The propagated route brings a better route for s_{int}. As a result, the cost of s_{int} are updated to c_p. We express s_{int} in Equation (1). The other part is update states other than s_{int}, of which case we keep the cost c_t.

$$s_{int} = \forall u_i, v_i = \begin{cases} * & \text{if } s_p[u_i] = * \text{ and } s_t[u_i] = * \\ s_t[u_i] & \text{if } s_t[u_i] \neq * \\ s_p[u_i] & \text{otherwise} \end{cases} \quad (1)$$

We do this for all pairs of entries in the propagated route and the target route. MergeRoute merges the propagated route (r_p) to the target route (r_t). We summarize MERGEROUTE(r_p,r_t) in Algorithm 2. We summarize the iterative process to derive the IGP best route in Algorithm 3.

Algorithm 3 Derive best route from IGP.

1: Initialize the route for all router as $(\infty : *, \ldots, *)$.
2: Set the route for originated prefix as $(0 : *, \ldots, *)$.
3: **while** routes updated **do**
4:     **for** each router R_i **do**
5:         $r_i \leftarrow R_i$'s route
6:         **for** neighbor R_j of R_i **do**
7:             $r_j \leftarrow R_j$'s route
8:             $l \leftarrow$ link between R_i and R_j
9:             $r_{prop} \leftarrow \textsc{PropagateRoute}(r_i, l)$
10:             $\textsc{MergeRoute}(r_{prop}, r_j)$
11:         **end for**
12:     **end for**
13: **end while**

B. Derive BGP Best Routes from Update States

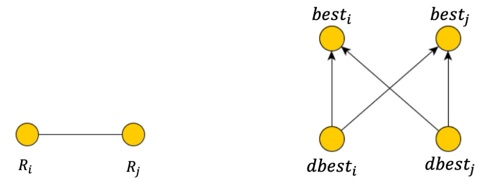
Now, we derive the best routes for Border Gateway Protocol (BGP) [26]. BGP is a policy-based routing protocol. Routers announce their best routes to neighbors according to the export policy. Neighbors pick the best route according to the import policy. In the following, we first derive the BGP best route for a given update state and then derive the BGP best routes when considering configuration updates.

1) Best Route for One Update State: A straightforward way to derive the BGP best routes is to model BGP as a Simple Path Problem and run the Simple Path Vector Protocol (SPVP) to calculate the best route [17]. However, deriving the best route with SPVP can take $O(2^{|V|})$ iterations where $|V|$ is the number of BGP routers.

Inspired by BiNode [28], we represent each BGP router with a constant number of routes. We represent each route as a node in a directed acyclic graph (referred to as the BiNode graph) and use directed edges to represent the routing decision process. Edges in the BiNode graph are determined by the BGP sessions. To build the BiNode graph, we construct edges by going over each BGP session in the network.

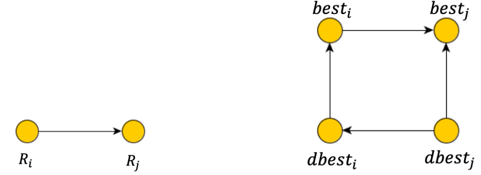
Case 1: iBGP Peer Session between R_i and R_j: We use *dbest* to represent the best route learned from iBGP client and eBGP neighbors. R_i learns route from R_j only when R_j's best route is learned from iBGP clients or eBGP peers, which is $dbest_j$. As a result, we add a direct edge from $dbest_j$ to $best_i$. Similarly, there is an edge from $dbest_i$ to $best_j$. We show the topology where R_i and R_j establish the iBGP peer session in Figure 3 (a) and the corresponding BiNode graph in Figure 3 (b).

Case 2: iBGP Client Session from R_i and R_j (R_j is the iBGP client): We use *dbest* to represent the route learned from iBGP client neighbors and eBGP neighbors. R_j is an iBGP client of R_i, as a result $dbest_i$ learns $best_j$ from R_j. We have a directed edge from $dbest_j$ to $dbest_i$. R_i announces its best route to R_j no matter where the best route is learned from since R_j is a client of R_i. We have an edge from $best_i$ to $best_j$. We show the topology where R_i and R_j establish the iBGP client session (R_j as iBGP client) in Figure 4 (a) and the corresponding BiNode graph in Figure 4 (b).



(a) R_i and R_j establish iBGP peer session. (b) BiNode graph for topology in a

Fig. 3: iBGP peer session between R_i and R_j and the corresponding BiNode.



(a) R_i and R_j establish iBGP Client session. (b) BiNode graph for topology in a

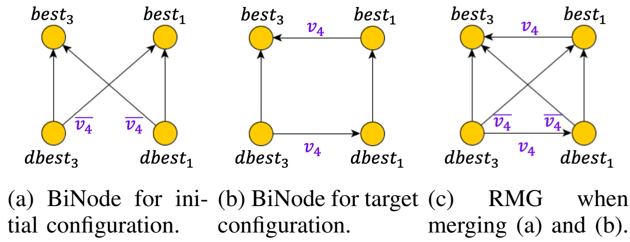
Fig. 4: iBGP client session between R_i and R_j and the corresponding BiNode.

Case 3: eBGP Sessions: eBGP sessions are used to exchange routes across multiple ASes. We add edges in the BiNode graph by splitting the best route learned from external into two parts: routes learned from customer ASes and routes learned from peer/provider ASes. The detailed construction of the BiNode graph for eBGP sessions can be found in [28].

2) Best Route under Configuration Update: As we can see from the above discussion, configuration determines the edges in the BiNode graph. That is, edges in the BiNode graph are different under different update states. We introduce Route Multiplexing Graph (RMG), which captures relationships between edges and update states. RMG is a directed acyclic graph built atop the BiNode graph. Edges in RMG are quantified with boolean expressions describing under what update states the edge exists in the BiNode graph.

To construct the RMG, one needs to know which edges are constructed when an update is not applied and applied for each update, respectively. We first consider updates as *not applied*, which is the initial configuration. We construct the BiNode graph for the initial configuration. When constructing the BiNode graph, we also label any potential updates on the edge in addition to adding edges for BGP sessions. For example, when constructing edges for an iBGP peer session between R_i and R_j, we check whether an update exists that changes this session (modify or remove). If such u_i exists, we label this edge with $\overline{v_i}$.

Similarly, we consider updates as *applied*. We build the BiNode graph for the target configuration and label the edges with any update that changes the corresponding BGP session. We build the RMG by merging the BiNode graphs for the initial and target configurations. We add an edge from node i to node j if there is an edge from node i to node j either in the BiNode graph for either the initial or target configuration. We



(a) BiNode for initial configuration. (b) BiNode for target configuration. (c) RMG when merging (a) and (b).

Fig. 5: RMG for R_1 and R_3 in Figure 1

quantify the edge from node i to node j with the corresponding label in the BiNode graph. We take the *or* of the labels in two graphs if the edge exists in both BiNode graphs.

We show the construction of RMG for the topology in Figure 1 with Figure 5. For simplicity of exposition, we only show R_1, R_3 and update u_4. We show the BiNode graph for the initial and target configuration in Figures 5 (a) and 5 (b), respectively. We merge the edge from $dbest_3$ to $dbest_1$ and $best_1$ to $best_3$ from Figure 5 (b) to 5 (a) and show the RMG in Figure 5 (c).

We derive the relationship between the best routes and update states with RMG. To do so, we represent the route decision process with a function of updates. Since the edges are quantified with a boolean expression and considered as existing conditioning on some update states, we use $\times$ operation between routes learned from neighbors and the update states to capture this. For example, when $best_i$ learns from $dbest_j$, $best_i$ picks the best routes from set $\{\dots, v_k \times dbest_j, \dots\}$. $v_k \times dbest_j$ returns $dbest_j$ when $v_k = true$ and a null route when $v_k = false$.

We use a selection function denoted by f_{sel} to capture the route decision process at each node. The selection function takes a set of routes as input and assigns an integer value called *ranking* to all the routes. The ranking is an integer that reflects the preference of a route by BGP, which BGP compares a sequence of attributes in the route to decide. Then, the routes with the highest ranking are returned. A null route always has the lowest ranking in f_{sel} and will never be selected whenever there is a non-null route.

As a result, each node takes the neighbors' exported route under a specific set of update states as input and picks the one with the highest preference. More specifically, suppose the boolean expression on edge from node j to node i is $q_{j,i}$, we express BGP routes with Equation (2).

$$p_i = f_{\text{sel}}\left(\left\{q_{j,i} \times g_{j,i}(p_j) | p_j \in Incoming(p_i)\right\}\right) \tag{2}$$

where $g_{j,i}$ denotes the transformation on routes when p_j is announced to R_i. $Incoming(p_i)$ represents the set of nodes that have a directed edge to p_i. p_j can be any route used to represent R_j (e.g., $best_j$ or $dbest_j$).

C. Expressing Network Properties

With best routes expressed with update states, we can express the network properties with update states. To do so, CURSOR expresses network properties with the best routes and then expands the best routes by exploiting the relationships between the best routes and update states derived in the previous sections.

Best routes from IGP and BGP work together to determine the forwarding behavior of packets within a network. For example, to decide the forwarding path for a destination outside the network, $best_i$ determines which egress point to use, and r_i decides how to forward the packet to that egress point. As a result, we derive the best route from IGP with Algorithm 1 and construct RMG to represent the BGP best route. After obtaining r_i and $best_i$, best route of R_i can be expressed with a function of r_i and $best_i$. We represent the best route of a router with a function of r_i and $best_i$.

$$datafwd_i = f_r(r_i, best_i) \tag{3}$$

where f_r represents the function where r_i and $best_i$ decides the forwarding path of R_i.

Now we represent network properties with $datafwd$. In general, we express the network properties (denoted with ϕ) with Equation (4).

$$\phi = f(\{datafwd_i | R_i\}) \tag{4}$$

where function f describes the constraints on routes (e.g., what values attributes should have). R_i represents the routers in the network. In the following, we show how we convert the expressions of best routes into boolean expressions of update states.

1) Network Properties for Internal Destinations: Network properties for internal destinations are determined by IGP routes only. As a result, those properties can be expressed with constraints on r_i. For example, the waypoint property for internal destinations only cares about whether a specific router is in the forwarding path.

Since we already derived r_i in Section IV-A, we know exactly which update states violate the network properties. We represent the network property with the union of all update states that violate the network properties. To get the update states, we check each entry in r_i. For entry $r_i = (c_i, s_i)$, s_i violates the network property if the forwarding behavior causes violations to network properties (i.e., $r_i \not\models \phi$[2]). We construct the boolean expression by taking the intersection of all intermediate states s_i which $r_i \not\models \phi$.

2) Network Properties for External Destinations: Network properties for external destinations require BGP to pick the egress point first. Then, a forwarding path to the destination can be determined. For such destinations, we only care about the constraints on forwarding paths within the network. We first figure out which egress points are used and then express network properties for each egress point.

To derive the egress points used, we iteratively plug in Equation (2) until we reach BGP routers with external announcements. Note that f_{sel} picks BGP routes based on ranking and update states. We can get a set of boolean expressions, each of which corresponds to one egress point.

[2]Note that forwarding path $datafwd_i$ can be easily constructed when running Algorithm 3, and therefore used to check whether $r_i \models \phi$

We express the network properties for external destinations to a set of constraints on the forwarding path to each egress point. For example, we express the waypoint property that requires passing a particular router with the set of constraints stating that the forwarding path to the egress point has to pass the same router. By doing so, we express the network properties for external destinations with the boolean expression of update states by first converting them to the corresponding properties for internal destinations. Note that the selection of the egress points is conditioned on specific update states. We encode the intersection (*and*) of the constraints of forwarding behavior (from the corresponding properties for internal destinations) and the boolean expression for the egress point (from the selection of BGP routes). We take the intersection of the constraints for all egress points as the network properties are considered as hold only when the forwarding path to all egress points satisfies the network properties.

V. Constructing Rules for Update Order

We construct the rule for update orders in this section. We construct rules from the boolean expression derived from the previous section.

We want to avoid entering update states in which network properties are violated. To do so, we first derive what update states violate the network property. That is, $s \not\models \phi$. To do so, we taking the negative of the expression constructed in the previous section and solve for the update states.

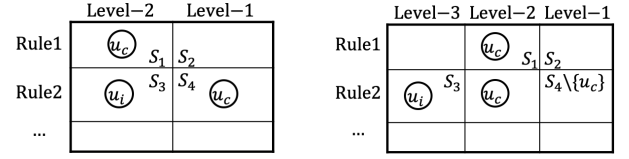
We solve the expression, $\overline{\phi} = true$, and construct a set of rules to avoid those update states. We first convert $\overline{\phi}$ into DNF, which is a disjunction of clauses where each clause is a conjunction of literals (i.e., update variables). Update states that making any clause true violates the network property. We represent one clause in the following form.

$$v_{p1} \wedge \cdots \wedge v_{pj} \wedge \overline{v_{n1}} \wedge \cdots \wedge \overline{v_{nk}} \tag{5}$$

where $v_{p1}, \ldots, v_{pj}$ are the update variables in positive literals and $v_{n1}, \ldots, v_{nk}$ are the update variables in negative literals. Note that at least one literal is positive literal. Otherwise, the initial configuration would violate the network property. Similarly, at least one literal is negative literal. Otherwise, the target configuration would violate the network property.

We consider one clause at a time. Note that update variables are connected with *and* in each clause. To avoid the clause becoming true, at least one item has to be false. Such cases are either update variables in positive literals are false, or update variables in negative literals are true.

In the configuration update process, we are finding an order of applying updates such that new configurations are finally deployed on all routers. That is, we are finding an order of updating variables from false to true. Back to Equation (5), the corresponding literal becomes true when updates in positive literals are applied and becomes false when updates in negative literals are applied. As a result, one of the updates in negative literals should be applied before all updates in positive literals are applied.



(a) u_c belongs to multiple levels in different rules.

(b) Aligning u_c to level-2 and move S_3 to level-3.

Fig. 6: Aligning u_c to the same level.

We construct the rule for each clause. Since any clause that becomes true can cause $\overline{\phi} = true$, we require all rules to be satisfied at the same time.

VI. Rule-Based Synthesis of Update Order

This section synthesis an order of applying updates that satisfies all the rules. We first generate a set of restricted rules from the original one, making synthesis easier. If no order was found from the restricted rules, we propose a heuristic algorithm to adjust the levels so that a valid rule can be found.

A. Restricting Rules

We restrict the rules by requiring *all* updates with negative literal applied before *any* updates with positive literal. That is, we apply updates in negative literals first and then apply updates in positive literals. However, we may have multiple rules causing conflicts. We propose a level-based synthesis to synthesize an order of applying updates that satisfies all rules. We introduce level-based synthesis in the following.

B. Level-based Synthesis

We propose level-based synthesis to synthesize an order of applying updates when no order is found from the restricted rules. The key idea is to align updates into levels from the rules. For example, if the rule requires u_1 to apply before u_2, we put u_1 into a level that is applied ahead of u_2. We assign an integer value for each update at each rule called "level" in such a way that

- Updates with higher levels should be applied before updates with lower levels.
- The order of applying updates at the same level does not matter.

CURSOR finds a way to assign the level to updates such that one update is assigned with one level through all rules. If that is the case, we can apply the updates from the highest to the lowest level. To start, we assign updates whose variables are negative literals level-2 and positive literals level-1 for each rule. Due to conflicts in the restricted rules, some updates can have both level-1 and level-2.

A variable cannot be in both positive and negative literals in one rule (otherwise, the corresponding clause in DNF is always false, and there is no rule for that clause). However, one variable might appear in multiple rules with both positive and negative literals. We adjust the level of updates in some rules to align the levels while keeping the levels assigned following the principles above.

We align the levels by reassigning levels of updates belonging to multiple levels. CURSOR aligns levels by reassigning the update to the highest level across all rules. However, reassigning an update with a higher level could break the principles mentioned above. For example, suppose update u_l appears in two rules. In the first rule, u_l is in level l_h and in the second rule, u_l is in l_l. Suppose $l_h > l_l$. We align u_l by reassigning u_l to level l_h in the second rule. However, there might be updates assigned to levels between l_h and l_l. For those updates, before aligning, they have a higher level than u_l, but after aligning, they have a lower or same level than u_l. Obviously, the order between these updates and u_l is flipped, which breaks the rule.

CURSOR fixes the rule by "pushing" all updates with a higher level than u_l left with $u_h - u_l$ levels. More specifically, we increase the level of any updates with a higher level than u_l by $u_h - u_l$. By doing this, CURSOR preserves the original rules while aligning u_l. This can be generalized to multiple rules by moving each rule separately. We use an example to illustrate the alignment process. We show a set of restricted rules in Figure 6 (a), where u_c is assigned level-2 in rule 2 but level-1 in rule 1. As mentioned above, we align u_c by reassigning u_c with level-2. However, there is a set of updates (S_3) in level-2 with a higher level than u_c. As a result, we increase the level of all the updates in S_3 by 1 when reassigning u_c in rule 2 with level-2.

We keep aligning updates until no update is assigned with more than one level. We can construct the order by applying updates from the highest to the lowest level. Within each level, we can apply updates in any order.

VII. Implementation and Evaluation

We implement CURSOR in C++ and evaluate the performance of CURSOR with extensive experiments in this section. We describe the experiment setting in Section VII-A. We compare the performance of CURSOR with the state-of-the-art update synthesis work on real-world configuration update scenarios in Section VII-B. We show the scalability of CURSOR in Section VII-C.

A. Experiment Setting

To evaluate the performance of CURSOR, we use a set of real-world topologies and synthesized topologies of various sizes. We use 85 real-world topologies from topology-zoo [20] and synthesize the fully meshed iBGP configuration with randomly generated IGP link weights. We also synthesize large-scale networks where the degree of routers follows the power-law distribution.

We test CURSOR with various network properties and a set of real-world configuration update scenarios. In this experiment, we focus on network properties like reachability and waypoint. We consider the following configuration update scenarios in the experiments.

- FM2SCRR: A fully meshed iBGP routing system is reconfigured into a single-clustered iBGP with route reflection architecture.
- SCRR2FM: An iBGP routing system with single-clustered route reflection is reconfigured into a fully meshed iBGP routing system.
- DoubleIGPWeight: All IGP link weights are doubled.
- HalveIGPWeight: All IGP link weights are halved.
- FM2MSRR: A fully meshed iBGP routing system is configured into a multi-clustered iBGP with route reflection architecture with IGP link cost doubled if a link crosses multiple clusters.
- Acquisition: Merge two networks into one single network.
- SplitTo2: One network is split into two networks.

We test CURSOR on a server with an 8 Core Intel(R) Xeon(R) Gold 6148 CPU 2.40GHz and 64GB memory.

B. Evaluating the Efficiency with Real-World Topologies

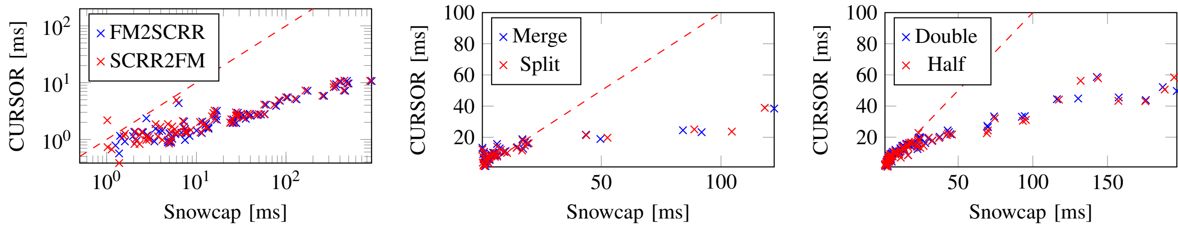
We evaluate the performance of CURSOR on real-world configuration update scenarios and compare it with the state-of-the-art synthesize approach, Snowcap. We synthesize a safe update order with both CURSOR and Snowcap on Topology Zoo and when the reachability and waypoint properties are required. We evaluate CURSOR under the following configuration update scenarios: FM2SCRR, SCRR2FM, DoubleIGPWeight (only waypoint property), and HalveIGPWeight (only waypoint property). We pick the BGP router with the highest degree as the route reflector when constructing the configuration for iBGP with route reflectors.

We evaluate CURSOR with 5 identical configuration update tasks and compare the time with Snowcap. We show the experimental results in Figure 7. The dashed link in red identifies the points where the CURSOR's synthesis time equals Snowcap. The results show that CURSOR can save 80% of time on average.

C. Evaluating the Scalability with Synthesized Topologies

We evaluate the scalability of CURSOR with synthesized large-scale networks with randomly generated IGP weight. We generate the configuration update scenario where the iBGP configuration was initially fully meshed iBGP and moved to a multi-cluster iBGP with route reflectors (FM2SCRR). To increase the complexity of the problem and evaluate CURSOR further, we double the IGP link weight if the source and target router are in different clusters in the target configuration.

We compare the synthesis time of CURSOR and Snowcap. We synthesize the update order when reachability property is required and choose one of the BGP routers with eBGP sessions and require the egress point to be chosen at all times (i.e., waypoint). Figures 8(a) and 8(b) illustrate the comparison between CURSOR and Snowcap. We set the timeout to 5 hours. The experimental results show that CURSOR can reduce the time to synthesize the update order by an order of magnitude when the number of updates grows larger. CURSOR can derive one safe order of applying updates within three hours when the number of updates grows to 100,000, while Snowcap timed out at 1,000.

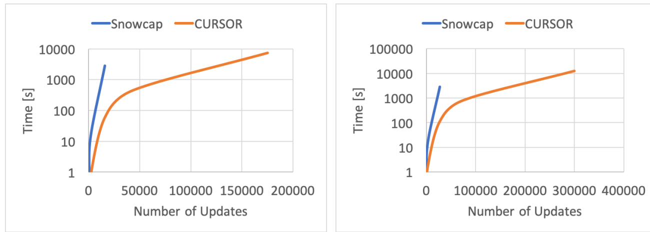


(a) Transiting between fully meshed iBGP and iBGP with route reflection.

(b) Network acquisition and split.

(c) Scaling IGP link weight (waypoint only).

Fig. 7: Synthesizing configuration update ordering on real-world topologies.



(a) Reachability

(b) Waypoint

Fig. 8: CURSOR v.s. Snowcap in terms of synthesis time.

VIII. Related Work

A. Data Plane Verification

To ensure the safety of networks, network verification has been proposed to verify network properties. Data plane verification [19], [32], [33] collects forwarding information base (FIB) from individual routers and verifies properties on a specific snapshot. However, the network is a dynamic system with configuration updates to adopt new features and accommodate increasing traffic demands. Data plane verification is fast. However, every change requires a complete verification to guarantee network safety under new environments.

Incremental data plane verification [18] verifies the data plane on the fly, blocking FIB changes in case of violation of network properties. However, not all changes can be blocked. For example, one cannot prevent a device from failing.

B. Control Plane Verification

To guarantee the safety of network configuration, control plane verifications [1], [3], [4], [10], [12], [24], [29], [38] have been proposed over the years. Control plane verification takes one or more network configurations with network environments as input and verifies certain properties hold under the configuration. In case of configuration changes, these systems can be used to verify the network property on the new configuration and guarantee that the new configuration still satisfies the network properties and does not cause violations when deployed. However, control plane verifications cannot be simply applied to derive a safe order for updates due to the large number of potential intermediate configurations.

C. Configuration Update Synthesis

Best practices and update guidelines [2], [7]–[9], [13], [15], [34], [35] have been made available by researchers to help network operators plan the configuration updates and reduce the chances of having outages and other unexpected issues during the update process. Although these guidelines have been tested and deployed in many real-world scenarios, best practice does not guarantee a safe order for any specific update scenario due to the existence of access lists, etc.

Network configuration update synthesis [22], [27], [36] takes real update scenarios and the network properties as input and synthesizes an order of applying updates such that network properties hold at all times. Network operators can follow the output sequence without worrying about how to apply guidelines. Existing works synthesize the update order by searching the order space. Although these works propose pruning mechanisms, they can lead to a large search space when the order space grows. In contrast to existing work, this paper directly exposes the rules for update order and no enumeration is needed.

D. Updates on Software Defined Networks

Forwarding rule updates on Software Defined Networks have a similar problem where the order of instructions on changing the forwarding rules needs to be carefully planned to avoid transient violations of network properties. Existing works [5], [11] have been proposed to address this problem. Our work is directly inspired by the approach for SDN, which explicitly exposes rules for update order.

IX. Conclusion

This paper proposes a rule-based synthesis, CURSOR, to derive a safe order of applying changes under network configuration updates. CURSOR accelerates the synthesis of configuration order by extracting the rules the update order should follow. CURSOR achieves this goal by representing the network properties with update states. CURSOR solves an order of applying updates with heuristic algorithms. We implemented the synthesis toolkit, which exploits CURSOR. The experimental results show that CURSOR can reduce the time it takes to derive a safe order by an order of magnitude compared to the state-of-the-art approach.

X. Acknowledgement

This work was supported in part by NSF under Grant CNS-1900866 and Grant CCF-1918187.

References

[1] A. Abhashkumar, A. Gember-Jacobson, and A. Akella. Tiramisu: Fast multilayer network verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 201–219, Santa Clara, CA, Feb. 2020. USENIX Association.

[2] P. S. Barry Raveendran Greene. *Cisco ISP Essentials*. Cisco Press, 2002.

[3] R. Beckett and A. Gupta. Katra: Realtime verification for multilayer networks. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 617–634, Renton, WA, Apr. 2022. USENIX Association.

[4] R. Beckett, A. Gupta, R. Mahajan, and D. Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, SIGCOMM '17, page 155–168, New York, NY, USA, 2017. Association for Computing Machinery.

[5] S. Brandt, K.-T. Förster, and R. Wattenhofer. On consistent migration of flows in sdns. In *IEEE INFOCOM 2016 - The 35th Annual IEEE International Conference on Computer Communications*, pages 1–9, 2016.

[6] E. Chen, T. J. Bates, and R. Chandra. BGP Route Reflection: An Alternative to Full Mesh Internal BGP (IBGP). RFC 4456, Apr. 2006.

[7] F. Clad, P. Mérindol, J.-J. Pansiot, P. Francois, and O. Bonaventure. Graceful convergence in link-state ip networks: A lightweight algorithm ensuring minimal operational impact. *IEEE/ACM transactions on networking*, 22(1):300–312, 2013.

[8] F. Clad, P. Mérindol, S. Vissicchio, J.-J. Pansiot, and P. Francois. Graceful router updates in link-state protocols. In *2013 21st IEEE International Conference on Network Protocols (ICNP)*, pages 1–10, 2013.

[9] N. El Rachkidy and A. Guitton. Changing the routing protocol without transient loops. *Computer Communications*, 82:49–58, 2016.

[10] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, OSDI'16, page 217–232, USA, 2016. USENIX Association.

[11] B. Finkbeiner, M. Gieseking, J. Hecking-Harbusch, and E.-R. Olderog. Model checking data flows in concurrent network updates (full version), 2019.

[12] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, page 469–483, USA, 2015. USENIX Association.

[13] 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.

[14] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 300–313, New York, NY, USA, 2016. Association for Computing Machinery.

[15] J. A. B. v. d. V. Gonzalo Gómez Herrero. *Network Mergers and Migrations*. 2011.

[16] R. Govindan, I. Minei, M. Kallahalla, B. Koley, and A. Vahdat. Evolve or die: High-availability design principles drawn from googles network infrastructure. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 58–72, New York, NY, USA, 2016. Association for Computing Machinery.

[17] T. G. Griffin, F. B. Shepherd, and G. Wilfong. The stable paths problem and interdomain routing. *IEEE/ACM Trans. Netw.*, 10(2):232–243, apr 2002.

[18] A. Horn, A. Kheradmand, and M. Prasad. Delta-net: Real-time network verification using atoms. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 735–749, Boston, MA, Mar. 2017. USENIX Association.

[19] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*, pages 113–126, San Jose, CA, Apr. 2012. USENIX Association.

[20] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan. The internet topology zoo. *Selected Areas in Communications, IEEE Journal on*, 29(9):1765 –1775, october 2011.

[21] A. Lerner. The cost of downtime, 1999.

[22] J. McClurg, H. Hojjat, P. Černý, and N. Foster. Efficient synthesis of network updates. *SIGPLAN Not.*, 50(6):196–207, jun 2015.

[23] J. Moy. OSPF Version 2. RFC 2328, Apr. 1998.

[24] S. Prabhu, K.-Y. Chou, A. Kheradmand, P. B. Godfrey, and M. Caesar. Plankton: Scalable network configuration verification through model checking. In *Proceedings of the 17th Usenix Conference on Networked Systems Design and Implementation*, NSDI'20, page 953–968, USA, 2020. USENIX Association.

[25] J. Regehr, Y. Chen, P. Cuoq, E. Eide, C. Ellison, and X. Yang. Test-case reduction for c compiler bugs. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, page 335–346, New York, NY, USA, 2012. Association for Computing Machinery.

[26] Y. Rekhter, S. Hares, and T. Li. A Border Gateway Protocol 4 (BGP-4). RFC 4271, Jan. 2006.

[27] T. Schneider, R. Birkner, and L. Vanbever. Snowcap: Synthesizing network-wide configuration updates. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 33–49, New York, NY, USA, 2021. Association for Computing Machinery.

[28] X. Shao, Z. Chen, D. Holcomb, and L. Gao. Accelerating bgp configuration verification through reducing cycles in smt constraints. *IEEE/ACM Transactions on Networking*, 30(6):2493–2504, 2022.

[29] X. Shao and L. Gao. Verifying policy-based routing at internet scale. In *IEEE INFOCOM 2020 - IEEE Conference on Computer Communications*, pages 2293–2302, 2020.

[30] H. Smit and T. Li. Intermediate System to Intermediate System (IS-IS) Extensions for Traffic Engineering (TE). RFC 3784, June 2004.

[31] Y.-W. E. Sung, X. Tie, S. H. Wong, and H. Zeng. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*, SIGCOMM '16, page 426–439, New York, NY, USA, 2016. Association for Computing Machinery.

[32] B. Tian, J. Gao, M. Liu, E. Zhai, Y. Chen, Y. Zhou, L. Dai, F. Yan, M. Ma, M. Tang, J. Lu, X. Wei, H. H. Liu, M. Zhang, C. Tian, and M. Yu. Aquila: A practically usable verification system for production-scale programmable data planes. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference*, SIGCOMM '21, page 17–32, New York, NY, USA, 2021. Association for Computing Machinery.

[33] B. Tian, X. Zhang, E. Zhai, H. H. Liu, Q. Ye, C. Wang, X. Wu, Z. Ji, Y. Sang, M. Zhang, D. Yu, C. Tian, H. Zheng, and B. Y. Zhao. Safely and automatically updating in-network acl configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*, SIGCOMM '19, page 214–226, New York, NY, USA, 2019. Association for Computing Machinery.

[34] L. Vanbever, S. Vissicchio, L. Cittadini, and O. Bonaventure. When the cure is worse than the disease: The impact of graceful igp operations on bgp. In *2013 Proceedings IEEE INFOCOM*, pages 2220–2228, 2013.

[35] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure. Seamless network-wide igp migrations. *SIGCOMM Comput. Commun. Rev.*, 41(4):314–325, aug 2011.

[36] L. Vanbever, S. Vissicchio, C. Pelsser, P. Francois, and O. Bonaventure. Lossless migrations of link-state igps. *IEEE/ACM Transactions on Networking*, 20(6):1842–1855, 2012.

[37] A. Zeller and R. Hildebrandt. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering*, 28(2):183–200, 2002.

[38] P. Zhang, A. Gember-Jacobson, Y. Zuo, Y. Huang, X. Liu, and H. Li. Differential network analysis. In *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, pages 601–615, Renton, WA, Apr. 2022. USENIX Association.