AED: Incrementally Synthesizing Policy-Compliant and Manageable Configurations

Anubhavnidhi Abhashkumar*, Aaron Gember-Jacobson†, Aditya Akella*
University of Wisconsin - Madison*, Colgate University†

ABSTRACT

When updating router configurations, network operators often attempt to meet a variety of management objectives (e.g., maintaining structural similarity across devices), while also ensuring all forwarding policies are correctly satisfied. Our tool, AED, automates this process. AED models configuration updates as a collection of syntax tree additions and removals, and formulates an innovative system of SMT (Satisfiability Modulo Theory) constraints that encode configurations' structure and interaction with routing algorithms. Operators express management objectives in a high-level language, and AED translates these to "soft" constraints that are maximally satisfied. Evaluations on real and synthetic network configurations show that AED can update networks with tens of routers and hundreds of policies in under a minute, and AED outperforms both hand-crafted updates and state-of-the-art tools in meeting management objectives.

CCS CONCEPTS

• **Networks** → **Network management**.

KEYWORDS

Network Synthesis

ACM Reference Format:
Anubhavnidhi Abhashkumar*, Aaron Gember-Jacobson†, Aditya Akella*, University of Wisconsin - Madison*, Colgate University†, . 2020. AED: Incrementally Synthesizing Policy-Compliant and Manageable Configurations. In *The 16th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '20), December 1–4, 2020, Barcelona, Spain.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3386367.3431304

1 INTRODUCTION

Modifying router configurations is a fact of life for network operators. For example, a study of 850 datacenter networks operated by an online service provider found over half of the networks have at least ten change events per month [23], and a study of two university campus networks found that over a million lines of configuration were changed in each university's network over a 5 year period [30]. Configuration changes are often motivated by a change in forwarding polices—e.g., to accommodate new services or end-hosts [30].

CoNEXT '20, December 1–4, 2020, Barcelona, Spain

ACM ISBN 978-1-4503-7948-9/20/12…$15.00
https://doi.org/10.1145/3386367.3431304

Modifying a network's configurations to implement new forwarding policies can be a daunting task due to the plethora of policies the network must satisfy [13], the inherent complexity of cross-protocol and cross-device dependencies [4, 12, 21], and the sheer size of router configurations [12, 35]. Fortunately, the burden of modifying network configurations can be reduced through (partial) automation of changes. Network operators have routinely leveraged configuration templates, scripts, and network management software to modify configurations [14, 24]. For example, scripts perform up to 60% of changes in the 850 networks operated by the online service provider [23]. More recently, practitioners have begun to "compile" (partial) configurations from policy databases [2], and researchers have developed systems to synthesize (partial) configurations from intents [10, 17, 18, 42, 44].

Regardless of whether changes are performed manually or automatically, network operators must ensure the changes achieve the desired forwarding policies—e.g., providing connectivity to new end-hosts—without introducing regressions—e.g., breaking connectivity for existing end-hosts. Moreover, the changes must be performed in a manner that meets an organization's network management practices—e.g., maintaining configuration similarity between devices with the same role [12, 23, 27], minimizing the number of devices affected [21], etc. Existing synthesis tools have been designed to guarantee configuration correctness [10, 11, 17, 18, 21, 42, 44], but little attention has been dedicated to guaranteeing correctness while also satisfying management objectives.

The scarcity of support for this duo of concerns is due in part to the lack of a principled understanding of network operators' configuration management objectives. Consequently, we interviewed or surveyed 58 network operators, and found that the size/scope of changes—e.g., which devices are modified—and the structure of the resulting configurations—e.g., maintaining the same packet filters across devices—are both very important to network operators (§3.1). However, existing synthesis tools either ignore these concerns [10, 18] or only support one of these two categories of objectives [11, 17, 21, 42, 44]. Furthermore, no tools allow operators to express custom management objectives.

This leads us to explore the following research question: *can we design a system that efficiently synthesizes configuration updates which conform to a network's custom management objectives and forwarding requirements?*

We answer this question in the affirmative with the design of a system we call AED. AED takes as input a network's management objectives, forwarding policies, and current configurations. Within a few minutes, AED produces a set of configuration updates that rectify forwarding policy violations and maximally satisfy management objectives.

Satisfying this duo of concerns is more complex than only guaranteeing policy-compliance, because we must find an optimal point within the typically large space of policy-compliant configurations, as opposed to choosing any policy-compliant configuration. In particular, computing suitable updates requires reasoning about: (*i*) the semantics of potential configurations, to ensure policy compliance; (*ii*) the syntax of potential configurations, to satisfy objectives for configuration structure/features; and (*iii*) the difference between current and potential configurations, to satisfy objectives for update size/scope. Some synthesizers only consider configuration correctness [10, 18] and can only reason about *i*. Other synthesizers also reason about *i* but offer partial control over configuration structure/features [17, 42] or change size/scope [21, 44] and may require additional operator input to reason about *ii* or *iii*. Automatically satisfying the wide range of management objectives that matter to operators (§3.1), requires higher-fidelity, fully automated reasoning.

Our key insight is to *model configuration updates as a collection of syntax tree additions and removals*. By modeling configurations (and updates) at a syntactic level, we can reason about configuration structure (*iii*). By modeling updates as low-level additions and removals, we can precisely reason about configuration differences (*ii*). Finally, by modeling configurations' influence on route computation and selection, we can reason about configuration semantics (*i*).

In particular, we automatically derive a symbolic sketch from the current configurations. The sketch mirrors the structure of the configuration syntax tree to facilitate reasoning about the impact of updates on configurations structure and change scope. We call the symbols in the sketch *delta variables*, because they encode potential syntax tree additions and removals. By solving for a configuration delta, rather than computing new configuration values from scratch [17, 42], we can easily reason about the size/scope of changes and compute updates efficiently.

Satisfying even simple forwarding policies—e.g., ensuring traffic takes a certain path—for certain management objectives—e.g. avoiding static routes—is NP-complete [42], so we use a MaxSMT (Maximum Satisfiability Modulo Theories) solver to determine the optimal values for delta variables. Our system of SMT constraints include the symbolic configuration sketch and a model of various routing algorithms (e.g., OSPF, BGP). We introduce "soft constraints" on the relevant delta variables to search for an update that maximally satisfies management objectives.

To facilitate flexibility in management objectives, AED provides a high-level language for operators to specify rich configuration-wide management objectives (§7.1). We observe that the objectives reported by operators (§3.1) and supported by prior tools [21, 42] focus on *restricting how specific (elements of) configurations are updated*. Consequently, AED allows objectives to be expressed as a restriction (e.g., ELIMINATE) on syntax subtrees. The subtrees are identified using a variant of XPath [3], and determine precisely which delta variables should be constrained.

We implement AED in Java using 4K LOC. Our experiments using configurations from real data center networks show that AED effectively supports a variety of management objectives, and generates updates that are better than hand-written ones. We empirically show that state-of-the-art tools either cause significantly greater configuration churn or cannot meet objectives optimally. We find that with optimizations, AED can generate updates in under a minute for real networks, and AED scales well with network/policy-set size.

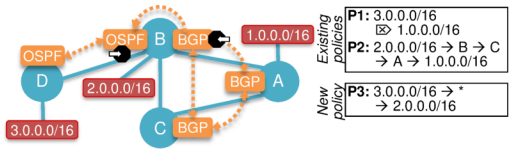


Figure 1: Example network: blue circles (*A*–*D*) are routers; red rectangles are groups of hosts; orange rectangles are routing processes; solid blue lines are physical links; dashed orange lines are routing adjacencies; black octagons are route or packet filters—*B* discards routes from *A* for 1.0.0.0/16 and assigns a local preference of 20 to other routes from *A*; *B* blocks incoming packets from 3.0.0.0/16

```
1  hostname B
2  interface Ethernet0/1
3    ip address 192.168.42.1/24
4  interface Ethernet0/2
5    ip address 70.70.70.1/24
6    ip access-group b_pfil in
7  router ospf 10
8    network 2.0.0.0/16
9    redistribute bgp
10 router bgp 50000
11   neighbor 192.168.42.2 route-map rmap in
12 route-map rmap permit 10
13   match ip address prefix-list b_rfil
14   set local-preference 20
15 ip prefix_list b_rfil deny 1.0.0.0/16
16 ip prefix_list b_rfil allow
17 access-list b_pfil deny ip 3.0.0.0/16 any
18 access-list b_pfil permit ip any any
```

Figure 2: Example configuration of router B

2 BACKGROUND

In this section, we present a brief overview of router configurations, route computation and selection algorithms, and forwarding policies. To illustrate these concepts, we use the example network and configuration in Figures 1 and 2.

In a network, each router runs one or more routing protocols (e.g., BGP, OSPF) to compute forwarding paths. Each instance is called a routing process: e.g., router *B* has two routing processes—one BGP and one OSPF—which are configured on lines 7–11. Each process receives route advertisements for specific IP prefixes from other processes running on the same or neighboring routers. Processes on neighboring routers that exchange routes—e.g., BGP_B and BGP_A—have a routing adjacency, defined on line 11. Processes on the same router that exchange routes—e.g., BGP_B and $OSPF_B$—engage in route redistribution, defined on line 9.

Routers may discard or modify route advertisements based on *route filters* defined in configurations: e.g., *B* discards routes from *A* for 1.0.0.0/16 and assigns a BGP local preference of 20 to other routes from *A*. Route filters (lines 12–16) consist of a set of match-action rules which include permit and deny statements (lines 15–16) to explicitly allow or block certain advertisements, and "set" statements (line 14) to modify certain fields of route advertisements. These fields are used in route selection algorithms to select the most preferred routes (as described below). Additionally, certain data packets can be filtered using *packet filters*, which consist of permit and deny statements: e.g., *B*'s packet filter blocks incoming packets from 3.0.0.0/16 (lines 17–18).

Route computation works as follows. Each routing process applies route filters (if any) to its incoming advertisements. Next, each

process selects and stores the best route (per IP prefix). The route selection algorithm for each process compares fields in a predefined order: e.g., BGP prefers routes with the highest local preference; if they are equal, then the shortest path length, and so on. Since each router can have only one route per prefix,[1] a router selects the best (i.e., lowest administrative distance) route among all its routing processes. The chosen process then advertises this route to all of its neighbors. Note that routers can also originate routes for specific IP prefixes: e.g., B originates a route for the directly connected prefix $2.0.0.0/16$ (line 8). Finally, during packet forwarding, each router forwards packets using the best route, provided the packet is permitted by a packet filter (if any).

Networks must often satisfy a diversity of forwarding policies [8, 17], such as: blocking traffic between specific subnets ($P1$); forwarding traffic through specific intermediate devices, or waypoints ($P2$); enabling packets from one subnet to reach another subnet ($P3$); or ensuring specific traffic is forwarded along a different path than other traffic (not shown).

When a new forwarding policy (e.g., $P3$) is introduced, a network's configurations must be updated to satisfy the new policy, without violating any existing policies. For example, $P3$ can be satisfied by updating the packet filter on router B (line 17–18) to allow $P3$'s traffic class.

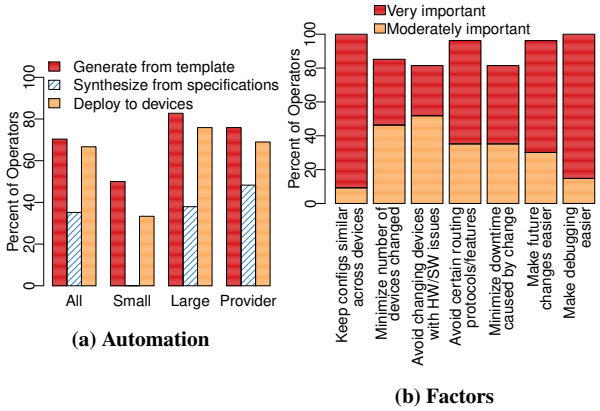
3 MOTIVATION

Satisfying all forwarding policies is the central goal of modifying network configurations. To understand what other factors operators take into account during updates, we examine the configuration change practices in over 50 organizations using one-on-one interviews and operator surveys (§3.1). We find that many operators (1) use limited automation to generate configuration changes, and (2) take into account many key factors beyond forwarding policies. Finally, we argue why existing tools [10, 17, 21, 24, 42] fail to meet operators' needs (§3.2).

3.1 Study of configuration change practices

We first conducted one-on-one interviews with operators from four different networks. Using the results from these interviews, we developed and conducted a survey of operators from an additional 54 networks[2] to study the configuration change practices used in production networks. The operators manage a variety of networks, including enterprise (41%), data center (50%), service provider (54%), and research & education (17%) networks. Half of the networks have more than 100 routers, and one-tenth have fewer than 10 routers. About two-thirds of operators employ automation to generate changes from templates and deploy changes to routers, but only one-third synthesize changes from high-level specifications (Figure 3a). The latter is not employed in small networks and is most heavily employed in service provider networks.

In our one-on-one interviews, we asked operators an open-ended question: besides satisfying policies, what additional factors do you consider when making a configuration change? A total of seven factors were suggested to us. Then, in our survey, we asked operators



(a) Automation **(b) Factors**

Figure 3: Operator survey results

to rate the importance of these seven factors when they change their network. Figure 3b shows for each factor the percentage of operators that reported the factor was moderately or very important for at least one type of change. We observe that all factors are at least moderately important for more than 80% of operators. Operators also wrote-in a few other factors that influence configuration changes, such as ensuring changes are easily verified and reversible.

Configuration similarity. The most important factor (very important to 90% of operators) is keeping configurations similar across devices with similar roles. For example, in a data center, each router's role is rack, aggregation, or spine. Devices in the same role have the same configuration template, and configuration similarity is violated if a device's configuration deviates from its template. For example, filters are often copied verbatim across devices with the same role [9, 14]. If the filters of one device are modified—i.e. new rules are added and/or existing rules are removed—then it violates the configuration similarity (template) of those devices.

Further, we observe a substantial (49%) correlation between keeping configurations similar and making debugging easier. This concurs with prior studies, which show that networks with high configuration similarity are less complex for operators to update [12].

Interestingly, our survey results show that configuration similarity is very important even for operators that reported using automation to generate changes from templates or synthesize changes from high-level specifications. This is in large part because not all network changes are automated. For example, one operator we interviewed said they recently started synthesizing route filters from specifications, but all other configuration changes were still done manually. Similarly, a prior study showed that in almost all networks operated by a large online service provider, at most two-thirds of the changes were automated [23]. In such networks, ensuring that automated and manual changes are similar helps debugging.

Devices changed. Minimizing the number of devices changed is very important to 38% of operators, and avoiding changes on specific routers (if possible) due to known hardware or software problems is very important to 30% of operators. Interestingly, there is a substantial (50%) correlation in the importance operators assigned to these two factors, suggesting that the former is partially motivated by the latter. For example, one operator we interviewed indicated some

[1] For simplicity of exposition, we ignore equal-cost multipath routing.

[2] We advertised our survey to the North American Network Operators Group (NANOG) and EDUCAUSE Network Management Constituent Group.

	Synthesizer	Management Objectives: Config structure	Update size/scope	Features used	Config components
Clean-slate	Zeppelin [42]	○	○	◐	Most
	Synet [18]	○	○	○	Most
	Propane [10]	○	○	○	BGP only
Incremental	NetComplete [17]	◐	○	◐	Most
	CPR [21]	○	●	○	Many
	Jinjing [44]	○	●	○	Pkt filters only
	Propane/AT [11]	◐	◐	◐	Most
	AED	●	●	●	Most

Table 1: Coverage of management objectives and configuration components (● = supported, ◐ = partially supported, ○ = not supported)

of their routers had flash reliability issues, so they sought to minimize the number of times they changed the configurations on these routers. Furthermore, irrespective of this known issue, the operator indicated there is always a risk for things to go wrong when an updated configuration is pushed to a device. This risk likely underlies the positive (32%) correlation we observe between the importance of minimizing/avoiding devices and minimizing the downtime required to deploy a change.

Feature usage. In our survey, 61% of operators indicated it was very important to avoid using certain protocols/features. Prior studies have found that operators may avoid certain routing protocols due to their additional licensing costs [12], or avoid features that can introduce routing loops (e.g., route redistribution [32]). Furthermore, we observe a substantial (68%) correlation between the importance operators assigned to this factor and making future changes easier.

In summary, operators are concerned with: (1) the structure of configurations, (2) the size and scope of configuration updates, and (3) the features used.

3.2 Limitations of existing tools

Inspired by our operator study, we seek a tool that *automatically generates configuration updates satisfying a range of management objectives and forwarding policies*. Although several types of configuration synthesizers exist, they offer limited support for key management objectives (Table 1).

Templates. According to our survey (Figure 3a) and prior studies [12], one widely used class of tools generate portions of configurations from specialized templates [24, 33, 43]. These tools fill-in-the-blanks in a pre-defined configuration segment with appropriate prefixes, link weights, etc. to satisfy new policies—e.g., enable new hosts to reach the rest of the network and vice versa. Although these tools can support configuration structure and feature usage objectives, they require manual effort to construct suitable templates. Furthermore, templates often cover only part of the configuration; ignoring the structure and semantics of other parts of the configuration can lead to policy violations and suboptimal satisfaction of management objectives.

Clean-slate synthesizers. These tools [10, 42] take as input the network topology, a set of policies [14], and possibly a configuration sketch. They produce brand-new, policy-compliant configurations for every router in the network. Some of them bound the use of certain protocols: e.g., Zeppelin [42] bounds the number of static routes, OSPF domains, etc. However, since these tools ignore current configuration, they cannot satisfy update size and scope objectives (e.g., minimizing the number of devices updated). Furthermore, some of these tools focus only on a narrow swath of configuration components: e.g., Propane [10] only synthesizes BGP configurations.

Incremental synthesizers. These tools [11, 17, 21, 44] take as input a set of policies and the network's existing configurations, and produce configuration patches to fulfill any previously unsatisfied policies without violating policies that were already satisfied by the existing configurations.

CPR [21] creates a graph-based model of a network control plane and produces updates that change the fewest lines of configuration (which is modeled via changes to edges in the graph-based model). However, CPR cannot satisfy configuration structure or feature usage objectives, because CPR's high-level control plane representation only captures configuration semantics, not configuration structure.

NetComplete [17] is another incremental synthesizer that automatically generates portions of configurations. NetComplete models configurations' semantics and syntax at the level of individual route advertisements and filtering policies using SMT constraints. Configuration values (e.g., filter rule actions) are symbolic, to allow the SMT solver to find a set of values that satisfy a network's policies. However, in NetComplete, operators have to manually reason about which values to leave symbolic and how possible values impact different management objectives. This is very challenging given the complexity of today's configurations [12]. Consequently, satisfying objectives requires significant manual guidance.

Other incremental synthesizers have similarly limited support for management objectives. Jinjing [44] focuses on a single configuration component as it only repairs packet filters. Propane/AT [11] uses a high-level abstract topology to generate templates, and it updates only devices whose templates have changed. But it does not allow operators to control the features used and the scope of updates. Additionally, Propane/AT was designed for topology changes and not policy changes.

In summary, existing configuration synthesis tools fall short in satisfying operators' needs w.r.t. configuration changes.

4 OUR APPROACH: AED

Our tool, AED, addresses the aforementioned shortcomings. AED takes as input: (*i*) a set of *forwarding policies*, expressed for specific source/destination subnets using existing high-level languages [17, 41]; (*ii*) a network's current router *configurations*, which violate one or more forwarding policies; and (*iii*) a set of configuration *management objectives*, expressed in a new high-level language (§7.1). AED generates a set of *configuration updates* that rectify forwarding policy violations and optimally satisfy management objectives.

Generating such updates requires a framework for reasoning about: (*i*) the semantics of potential configurations, to ensure policy compliance; (*ii*) the syntax of potential configurations, to satisfy configuration structure and feature usage objectives; and (*iii*) the difference between current and potential configurations, to satisfy update size and scope objectives.

Our key insight is to *model configuration updates as a set of syntax tree additions and removals*. By modeling configurations (and updates) at a syntactic level—instead of a higher-level intermediate

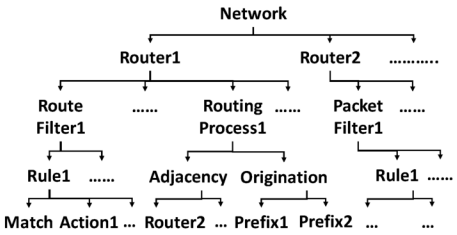


Figure 4: Configuration syntax tree

representation [8, 10, 21, 42]—we can easily reason about the structure and contents of potential configurations (*ii*). By modeling updates as additions and removals—instead of a complete regeneration of (segments of) configurations [10, 17, 42]—we can easily reason about the delta between current and potential configurations (*iii*). Finally, by modeling configurations' influence on route computation and selection, we can reason about configurations' semantics (*i*).

The fundamental challenge is determining which nodes to add and remove from the syntax tree. As illustrated in §2, router configurations specify five key elements that dictate a network's forwarding behavior: (*i*) which routing protocols to use, (*ii*) which neighboring routers to communicate with—also known as *routing adjacencies*, (*iii*) which routes (i.e., prefixes) to originate, (*iv*) which routes to filter/prefer, and (*v*) which packets to filter. The precise organization of this information varies between vendors, but it is generally structured in the manner shown in Figure 4, where each leaf node represents a single line of configuration.

AED automatically derives a symbolic sketch from the current configurations to encode all possible syntax tree additions and removals. We refer to the symbolic variables in the sketch as *delta variables*, because they encode the difference between the current and potential configurations. We create a delta variable for each current and potential node in the syntax tree. (Potential nodes are derived from the physical topology—e.g., potential routing adjacencies—and forwarding policies—e.g., potential router filter rules).

Even with simple forwarding policies—e.g., ensuring traffic is forwarded along a certain path—finding a suitable configuration is not trivial. Satisfying certain simple management objectives, like avoiding static routes, is NP-complete [42]. Consequently, we formulate a system of SMT constraints whose solution is a correct (w.r.t. forwarding policies) and optimal (w.r.t. management objectives) set of syntax tree additions/removals. The system of constraints includes: (*i*) configuration constraints (§5.2), which encode the symbolic sketch; (*ii*) algorithmic constraints (§6.1), which encode configurations' influence on route propagation and selection; (*iii*) policy constraints (§6.2), which encode forwarding policies, and (*iv*) management constraints (§7.2), which encode management objectives.

We introduce a high-level language (§7.1) for operators to express management objectives in terms of restrictions (ELIMINATE, EQUATE, or NOMODIFY) on regions of the syntax tree. These expressions are translated into constraints over delta variables.

The next four sections describe AED in detail.

5 ENCODING CONFIGURATION

In this section, we describe AED's encoding of potential configurations as a symbolic sketch. The sketch is automatically derived from the current configurations and encoded using first-order logic. We use the example network in Figure 1 to help illustrate AED's encoding.

5.1 Symbolic variables

AED's symbolic configuration sketch contains three types of symbolic variables.

Delta variables encode configuration updates. AED creates a delta variable for each current and potential node in the syntax tree. Assigning a true (or non-zero) value to a delta variable indicates the corresponding configuration element was added/removed (or incremented/decremented in the case of a numeric value such as local preference). These variables are used in the configuration sketch to model the impact of changes on protocol parameters. Delta variables are also used in management constraints (§7.2) to model the impact of changes on configuration structure/feature usage and update size/scope. AED maintains a mapping between delta variables and nodes of the syntax tree in order to allow AED to quantify a change's impact on configuration structure (§7.2). For simplicity of exposition, we embed this mapping information in the names of delta variables: e.g., $rm\_R1\_RFilter1\_Rule1$ corresponds to the left-most route filter rule node in Figure 4.

Protocol parameter variables represent configuration values that impact the advertisement and selection of routes: i.e., whether a protocol is enabled, whether a routing adjacency is defined, whether a prefix is originated, or whether a route/packet is allowed. These values must be symbolic, because they depend on the contents of configurations, which partially depends on the value of delta variables. Note that the encoding of protocol parameters in AED is different from NetComplete [17]: NetComplete encodes them as either unconstrained symbolic variables or constants depending on the configuration templates, whereas AED encodes all protocol parameters as symbolic variables constrained on the network's current configurations and the delta variables.

Route advertisement variables represent the route advertisements produced by routing algorithms. AED creates records of symbolic values for each potential routing adjacency: e.g., $outBGP_{A\rightarrow B}$ represents a BGP advertisement sent from A to B and $inBGP_{B\leftarrow A}$ represents an advertisement B receives from A. The fields within each record are similar to the fields in actual protocol messages: e.g., $prefix$, path $cost$, administrative distance (ad), and protocol-specific attributes such as BGP local preference (lp). Also, every record has a boolean field that indicates whether the advertisement is *valid*.

5.2 Configuration constraints

We now describe how AED models each of the five key router configuration elements (Figure 4) using first-order logic.

Routing protocols and adjacencies. AED's symbolic configuration sketch contains protocol parameter variables for each routing protocol a router supports (e.g., BGP_A) and each neighboring router with whom information could be exchanged (e.g., $BGPAdj_{A\rightarrow B}$). If a routing protocol is currently enabled or a routing adjacency

is currently configured, then we introduce a delta variable to represent the potential disabling of the protocol or adjacency (e.g., $rm\_A\_BGP\_Adj\_B$). The protocol parameter variable is constrained to be true as long as the protocol or adjacency isn't disabled. For example, since there is a route adjacency between A and B in Figure 1, $BGPAdj_{A\to B}$ will be constrained as follows:

$$BGPAdj_{A\to B} \iff \neg rm\_A\_BGP1\_Adj\_B$$

Conversely, if a protocol or possible adjacency isn't configured, then we introduce a delta variable to represent the possible enabling of the protocol or adjacency (e.g., add_D_BGP), and we constrain the protocol parameter variable to only be true when the corresponding delta variable is true.

Route filters. Route filters define a set of match-action rules that are applied to route advertisements. AED models the constraints representing filters as if-then-else statements. For example, Figure 5 shows the encoding of B's route filter that is applied to BGP advertisements from A.[3] The constraints modeled as if-then-else statements (1) match the filter rule with the advertisement (line 4), and (2) set the action fields of the rule based on configuration constants. The actions associated with each rule: (i) dictate whether the advertisement is allowed or dropped (line 5), and (ii) set the value of certain metrics (line 6) like local preference, administrative distance, etc (unspecified fields get default value). Next, AED uses delta variables to encode the addition, removal and modification of filter rules. To model rule removal, AED includes a $rm\_*\_rFil\_*$ delta variable in the match conditions (line 4). To modify the actions of existing rules, AED uses (i) boolean delta variables to update filter allow actions (line 5); and (ii) integer delta variables to update preference values assigned to routes (line 6). AED represents preference values as the sum of the current constant and an integer delta variable. Finally, to model rule additions, AED prepends an additional conditional statement (line 1-3) predicated on $add\_*\_rFil\_*$ delta variables.

If the same filter is applied to advertisements from multiple neighbors, then the constraint is replicated for each neighbor.

```
1  if match(outBGP_{A→B}.prefix, policy.dstPrefix) ∧ add_B_rFil_A_new then
2      filter_{B→A}.allow = B_rFil_A_new_allow
3      ...
4  else if match(outBGP_{A→B}.prefix, 1.0.0.0/16) ∧ ¬rm_B_rFil_A_1 then
5      filter_{B→A}.allow = B_rFil_A_1_allow
6      filter_{B→A}.lp = 100 + B_rFil_A_1_lp //by default lp is 100
7      ...
8  else
9      filter_{B→A}.allow = ¬B_rFil_A_2_allow
10     filter_{B→A}.lp = 20 + B_rFil_A_2_lp
```

Figure 5: Encoding of route filter on B

Originated prefixes. Originated prefixes are encoded similar to route filters: a constraint identifies the originated prefix that matches the destination prefix of the target policy (POLICY.DSTPREFIX) and stores the matched prefix in a symbolic variable. Figure 6 encodes prefixes originated by A. It is unnecessary to make the prefix constant a symbolic variable, because we can realize a change in prefix by removing the current originated route and adding a new originated route.

[3] if-then-else is syntactic sugar that can be translated to a conjunction of implications in classic first-order logic.

```
1  if add_A_BGP_B_Orig_new ∧ match(outBGP_{A→B}.prefix, policy.dstPrefix)
     then
2      originate_{A→B}.advertise = true
3      originate_{A→B}.prefix = POLICY.DSTPREFIX
4  else if match(policy.dstPrefix, 1.0.0.0/16) ∧ ¬rm_A_BGP_B_Orig_1 then
5      originate_{A→B}.advertise = true
6      originate_{A→B}.prefix = 1.0.0.0/16
7  else
8      ...
```

Figure 6: Origination of prefix from router A

Packet filters. Packet filter also consist of a set of match-action rules and are encoded similar to route filters. For example, Figure 7 encodes B's packet filter using $rm\_*\_pFil\_*$, $add\_*\_pFil\_*$, and $*\_pFil\_*\_allow$ delta variables.

```
if .. then
else if match(policy.srcPrefix, 3.0.0.0/16) ∧ match(policy.dstPrefix, *) ∧
  ¬rm_B_pFil_D_1 then
    pFil_{B←D}.allow = false ∨ B_pFil_D_1_allow
else if ... then
```

Figure 7: Match-action rules for packet filter on B

Upper bound on delta variables. We now discuss the upper bound on the number of delta variables added in AED's encoding. Note that we add delta variables for different key router configuration elements. We first model adjacency update using delta variables to remove existing route adjacency and add new route adjacency. If the network has R routing processes, then in the worst case scenario, each R process can be a neighbor of the remaining $R-1$ processes. This will create R^2 route adjacencies in the network. Hence, the upper bound on the number of delta variables added for adjacency is R^2. Next, we model route filter updates. For each filter prefix that already exists in the configuration files, we add delta variables to remove the match condition (and hence the rule) for that prefix and modify its filter actions. Additionally, we use delta variables to represent adding a new filter rule w.r.t. the policy being synthesized. Hence, the upper bound on the number of delta variables added for route filters is the number of unique prefixes (P) in AED's configuration files and policies. The same logic applies to packet filters and origination prefixes. Since each adjacency will have its own filters and origination prefixes, the upper bound on the total number of delta variables will be a function of R^2 and P, i.e. $O(R^2.P)$.

6 POLICY COMPLIANCE

Next, we discuss how AED guarantees configuration updates satisfy operator-specified forwarding policies.

6.1 Encoding routing algorithms

The control plane advertises, computes, and selects routes based on routing algorithms. AED's encoding of these algorithms are similar to tools like Minesweeper [8] and NetComplete [17]. For brevity, we explain the encoding at a high-level leaving detailed descriptions to Appendix A.

AED models receiving and sending of route advertisements using constraints expressed over symbolic route advertisements. Certain fields of the symbolic advertisement depend on the route filter that applies to the adjacency (e.g. lp), while other fields are populated from the adjacent process's outgoing advertisement (e.g. $prefix$).

Next, AED creates control and data plane forwarding variables for each physical adjacency: e.g., $controlFwd_{A\rightarrow B}$ and $dataFwd_{A\rightarrow B}$ represent A's decision of whether to forward traffic to B. AED models the route selection algorithm using a preference relation over route metrics (prefer routes with lowest *ad*, highest *lp*, etc). A router's best route determines which interface (e.g., $controlFwd_{X\rightarrow Y}$) is used to reach the destination listed in the policy. AED encodes traffic forwarding using a $dataFwd$ variable whose value is constrained based on the chosen route and any packet filters defined in the configuration.

6.2 Encoding policies

AED can compute updates for a wide range of policies, including: reachability, blocking, isolation, waypointing, path preferences and length constraints, and avoiding loops and black holes. The target policy is expressed using the $dataFwd$ variables. For example, $P2$ in Figure 1 is encoded as:

$$dataFwd_{B\rightarrow C} \wedge dataFwd_{C\rightarrow A}$$

Figure 8: Encoding of waypoint policy $P2$

Handling multiple policies. The above encoding is designed to model a network's behavior w.r.t. a single policy. However, computing separate configuration updates for each policy can lead to an update, and hence network paths, that satisfy one policy but violate another. For example, consider the network and policies in Figure 1. Due to the packet filter on B, the network currently satisfies policy $P1$ and violates policy $P3$. Using the above encoding to compute an update that satisfies $P3$ may result in an update that removes that filter on B, which causes $P1$ to be violated. Thus, when computing updates, AED must consider how a configuration change may impact multiple policies.

The above encoding contains only one set of symbolic route advertisements between each pair of routing processes, so the encoding can only be used to reason about one destination prefix at a time. To reason about multiple policies with different destination prefixes,[4] we must introduce multiple sets of symbolic route advertisements—one for each prefix, e.g., $inBGP^{1.0.0.0/16}_{B\leftarrow A}$ and $inBGP^{2.0.0.0/16}_{B\leftarrow A}$. Furthermore, since the constraints that encode route filters and originated prefixes (e.g., Figure 5 and 6), route sending and receiving functions, and route selection (§6.1 and Appendix A) are predicated on the route advertisements, we also include per-prefix versions of these constraints and variables. The data forwarding decisions (e.g. $dataFwd$) depend on packet filter, which may filter on the basis of source and/or destination prefixes, so we must include per-prefix-pair versions of these constraints and variables. All of these additional constraints and variables—needed for correctness—substantially increases the size of the SMT problem; we address this in §8.

Now that our encoding contains variables and constraints specialized for different prefixes, we must consider how configuration changes—which are modeled by our delta variables (§5.1)—can impact different prefixes. For example, removing a routing adjacency prevents all prefixes from being advertised to a neighboring router, whereas a route filter rule (e.g., a conditional in Figure 5) impacts a specific prefix. When we introduced filter ($add\_*\_rFil\_*$) and origination ($add\_*\_Orig\_*$) delta variables in §5.2, we did not include a symbolic variable for the prefix to the which the addition applies, because the encoding only considered one policy, and hence one prefix. Now that our encoding contains multiple policies, we need to create specialized per-prefix versions of these and related variables (e.g., filter variables $*\_rFil\_*\_allow$ and $*\_rFil\_*\_lp$) to allow network paths to be customized on a per-prefix basis through the addition of prefix-specific configuration constructs. Similarly, we need to create per-prefix-pair versions of packet filter variables: $add\_*\_pFil\_*$ and $*\_pFil\_*\_allow$. With $rm\_*\_rFil\_*$, $rm\_*\_Orig\_*$, and $rm\_*\_pFil\_*$, the impacted prefixes are already included in the constraint, so we do not need multiple versions of these variables. Similarly, routing adjacencies are not prefix specific, so we do not need multiple versions of $add/rm\_*\_Adj\_*$.

By applying the aforementioned transformations to AED's model, we ensure the model faithfully represents the real network's decision processes and constrains update options to the space of correct and valid router configurations.

7 MANAGEMENT OBJECTIVES

In §3.1, we showed that operators consider many factors when updating configurations. To ensure updates computed by AED account for these factors, we introduce a high-level language for operators to express management objectives. Objectives expressed in this language are translated into boolean formulas and appended to the SMT encoding as soft constraints.

7.1 Objective language

We observe that operators' management objectives focus on *restricting how specific (elements of) configurations are updated*. Consequently, in AED, an objective is expressed as a high-level restriction on syntax subtrees. AED's overarching goal is to satisfy as many objectives as possible.

Restrictions. Based on our survey results (§3.1) and review of prior work [21, 42], we identify three primary restrictions: eliminate subtrees (ELIMINATE), make a set of subtrees consistent across devices (EQUATE), or avoid changes altogether (NOMODIFY). AED supports these restrictions and encodes them in SMT constraints using boolean operators (described later in §7.2). AED can easily be extended to support additional restrictions, as long as they can be encoded using boolean operators—e.g., a "prefer changes" restriction is simply the negation of NOMODIFY.

Syntax subtree selection. The objectives in §3.1 apply to various subtrees of the configuration syntax tree (Figure 4): some apply to a specific router—e.g., avoid changing routers with hardware issues—and some apply to a particular feature—e.g., maintain packet filter clones. In AED, the relevant subtrees are expressed using XPath [3]. XPath is designed for selecting nodes of an XML document based on node names, attributes, and relative location in the XML tree. AED uses XPath expressions to select root nodes of syntax subtrees based on node type (e.g., PacketFilter), node name (e.g., *internal*), and node location. For example, all instances of a packet filter called *internal* can be selected using the expression:

```
//PacketFilter[name="internal"]
```

[4] If policies' prefixes partially overlap, we can subdivide policies into non-overlapping packet equivalence classes [26].

Objective	Constraints
Preserve packet filter clones	EQUATE //PacketFilter GROUPBY name
Minimize number of devices changed	NOMODIFY //Router GROUPBY name
Avoid changing devices with HW/SW issues	NOMODIFY //Router[name="B"] NOMODIFY //Router[name="C"]
Avoid protocols/features (e.g. static routes)	ELIMINATE //RoutingProcess [type="static"]/Origination GROUPBY prefix

Table 2: Encoding important management objectives

Multiple objectives. An objective is satisfied if the specified restriction holds true for *all* syntax subtrees selected by the XPath expression. For example, NOMODIFY //Router is satisfied if all routers' configurations are unmodified. To express the objective of minimizing the number of devices changed, an operator must define multiple objectives with different XPath expressions. For example, NOMODIFY //Router[name="A"] is satisfied if router *A*'s configuration is unmodified, NOMODIFY //Router[name="B"] is satisfied if router *B*'s configuration is unmodified, etc. Since such enumeration is tedious and error prone, we introduce a GROUPBY clause whose semantics is to group syntax subtrees based on a specified attribute of the root node and apply the restriction to each group. For example, NOMODIFY //Router GROUPBY name defines a NOMODIFY objective for each router, thus codifying the objective of minimizing the number of devices changed. Note that GROUPBY is syntactic sugar and does not fundamentally change the semantics of AED's objective language.

By default, every objective is assigned equal weight: e.g., avoiding changes on one router is just as desirable as avoiding changes on a different router. However, operators can assign weights to different objectives to express their importance.

Examples. Table 2 shows how to express the management objectives discussed in §3.1. To make AED easier to use, we include a library of pre-defined objectives (including those in Table 2) for operators to choose from. If these objectives do not meet operators' needs, then operators can define their own objectives using restrictions and XPath expressions.

7.2 AED: Encoding management objectives

To ensure AED computes updates that maximally satisfy management objectives, we convert the SMT problem into a maximum satisfiability modulo theories (MaxSMT) problem. A MaxSMT problem contains hard constraints that *must* be satisfied and soft constraints that should be *maximally* satisfied. In AED, hard constraints are the previously presented constraints that encode forwarding policies (§6.2), configurations (§5.2), and control/data plane algorithms (§6.1); these are necessary to ensure the computed updates are correct.

AED creates a soft constraint for each objective (after "desugaring" GROUPBY clauses) expressed by operators in AED's objective language. The constraint encompasses the delta variables associated with the nodes in the syntax subtrees selected by the objective's XPath expression. (As mentioned in §5.1, AED creates a delta variable for each current and potential node in the syntax tree). The selected delta variables are constrained according to the objective's restriction: NOMODIFY is the negation of the disjunction of the variables; ELIMINATE is the conjunction of negated add and non-negated remove variables; and EQUATE is the conjunction of the equality of sets of variables associated with nodes in the same position in each of the subtrees. For example, the objective in the first row of Table 2 translates to the following soft constraint:

$$rm\_D\_pFil_B\_1 = rm\_B\_pFil_C\_1 = ..\wedge$$
$$D\_pFil_B\_1\_allow = B\_pFil_C\_1\_allow = ..\wedge$$
$$rm\_D\_pFil_B\_2 = rm\_B\_pFil_C\_2 = .. \wedge ..$$

8 OPTIMIZATION STRATEGY

The above network model enables AED to compute correct, optimal configuration updates. However, the complexity of the resulting SMT formulation is substantial, and hence the time required to solve it is high. For example, we find that updating a network with just 20 routers and a few hundred policies takes 20 minutes. Moreover, the time to compute updates is ≈40X worse than the state-of-the-art incremental synthesis tool [21]. Next, we propose three distinct strategies that significantly improve AED's speed.

Pruning irrelevant configuration. The parallels between AED's encoding of configuration and configuration's syntactic-structure is essential for realizing many important management objectives (e.g., maintaining configuration similarity). However, a significant fraction of a network's configurations is often irrelevant for a given policy, as they do not overlap with the source and/or destination prefixes associated with that policy. For example, only those packet filter rules that match a given destination will impact reachability to that destination—i.e., lines 4–7 in Figure 5 are irrelevant for policy **P3** from Figure 1.

The inclusion of irrelevant conditionals in origination, route filter, and packet filter constraints, and the delta variables associated with these clauses, increases the computational complexity of the constraint problem, thereby reducing AED's efficiency. Fortunately, we can statically prune a significant fraction of the irrelevant conditionals and delta variables by examining whether a rule applies to (part of) the same traffic class covered by a network policy: if the range of source and destination prefixes matched by the conditional does not intersect with the range of source and destination prefixes covered by a network policy, then the conditional, and its associated delta variable, is a candidate for pruning. For example, Figure 5 is encoded as the following for policy **P3**:

```
1 if match(outBGP_{A→B}.prefix, policy.dstPrefix) ∧ add_B_rFil_A_new then
2       filter_{B→A}.allow = B_rFil_A_new_allow
3       ...
4 else
5       filter_{B→A}.allow = true ∧¬B_rFil_A_2_allow
6       filter_{B→A}.lp = 20 + B_rFil_A_2_lp
```

Grouping policies based on a destination address. As discussed in §6.2, AED considers all policies in unison to compute valid updates. To overcome the resulting performance issue, we formulate multiple MaxSMT problems, one per destination. These per-destination formulations are significantly smaller in size, and having multiple SMT formulations allows us to solve them in parallel.

The solutions to each problem will not conflict, because routing can always be customized on a per-destination basis using route filters and static routes. For example, the problem presented in §6.2 with applying AED separately for $P1$ and P3 in Figure 1, can be

addressed by updating B's packet filter to match both source and destination prefixes. However, the computed updates may be sub-optimal w.r.t. the management objectives, because the management objectives are considered separately for each destination. However, in practice the computed updates are (close to) optimal (§9.3).

Replacing integer variables with booleans. AED uses integer variables for cost and metric (e.g., *ad*, *lp*, *med*) values when computing updates that change which routes are preferred (§5.2). However, each integer variable in the model expands the space of possible updates by a factor of 2^{32}. To reduce the space of possible updates, we constrain the possible integer values to a small set of values represented by boolean variables. For cost and metric values, the set of values we choose is based on our observation that we only need to know the relative *rank* of the route, not its absolute "distance" from another route. In most cases, changing a route's rank to have an equal or in-between rank *relative* to other routes is sufficient. Consequently, if the current configurations contain n distinct values for a cost or metric, we limit the set of possible new values to $(2n+1)$ choices. For example, if the network model currently contains three distinct BGP local preference values (50, 100, and 150), we limit the choice of new values to one of seven choices: LP_{0-49}, LP_{50}, LP_{51-99}, LP_{100}, $LP_{101-149}$, LP_{150}, $LP_{151-inf}$. We replace the integer variable lp in the network model with $(2n+1)$ boolean variables corresponding to the choices in the set.

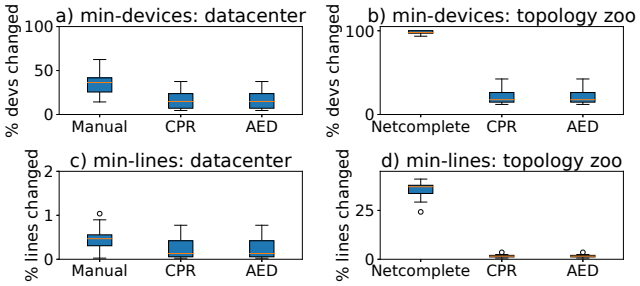


Figure 9: Minimize devices and lines changed

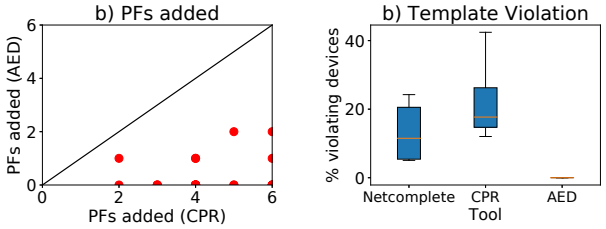


Figure 10: Other management objectives

9 EVALUATION

We prototype AED [1] atop Minesweeper [8]. Minesweeper uses Batfish [20] to parse router configurations, and the Z3 SMT solver [16] to encode and solve the underlying SMT formulation. We add our objective language and modify Minesweeper to incorporate our syntactic-level, update-oriented network model. In all, this amounted to ≈ 4K lines of Java code.

Next, we evaluate AED along a variety of issues:

- How effective is AED at meeting different management objectives? Is AED useful in practice?
- How does AED's performance compare with other incremental synthesis tools? Does AED's generality lead it to be slower than the less general CPR?
- How does AED's performance scale with network size and the set of policies that need to be satisfied?
- How well do AED's optimization techniques work?

We compare AED against two other incremental synthesis tools, CPR [21] and NetComplete [17]. We use NetComplete with all configuration constructs made symbolic.[5]

All our experiments were performed on machines with 10 core 2.4 GHz Intel Xeon Processors and 132 GB RAM.

Dataset. We run extensive experiments on both *real* and *synthetic* network configurations. For the former, we use configuration snapshots from 24 datacenter networks operated by a large online service provider. The dataset does not include operators' intended policies, so we infer all of a network's reachability policies by checking for reachability between every pair of subnets using Minesweeper [8]. These 24 networks have between 2 and 24 routers, and support 50 to 6600 reachability policies. In our experiments, given a network's "before" snapshot, we run AED with different objectives to update the "before" configurations to new configurations that satisfy all policies observed in the "after" snapshot. We then compare AED-generated configurations against the actual "after" configurations. The "after" configurations are the result of operators manually updating, with limited automation [23], the "before" configurations. To compute manual updates, we only consider changes related to routing/forwarding. To compute changes in templates, we group configurations based on their filter rules in the "before" snapshot. We then compare these segments of the configuration across the actual "before/after" snapshots.

To evaluate AED's scalability and performance, we use synthetic BGP configurations generated by NetComplete [17] for 10 network topologies of varying sizes (30-160) from the Internet topology zoo [31].

For brevity, we show most results by updating networks to support new reachability policies. We show results for adding other policy types in §9.2.

9.1 Management objectives and utility

AED allows operators to optimize updates for a variety of management objectives. We study the effectiveness with which AED supports such objectives.

9.1.1 Quantitative Analysis. First, we quantitatively compare updates made by AED against manual and other synthesis tools. We use four objectives: minimize devices changed (*min-devices*), minimize lines changed (*min-lines*), preserve templates (*preserve-templates*), and minimize the use of packet filters (*min-pfs*).

For real data center networks, we compare AED and CPR against manual updates. We cannot use NetComplete for these networks as

[5] NetComplete is an *incremental* tool, but there is no easy way to use it as such to compare against AED, because NetComplete needs manual guidance to be used for incremental synthesis (§3.2).

they cannot model features like packet filters and route redistribution. For the topology zoo networks [31], we first synthesize configurations which support 8 randomly-generated reachability policies. We then generate 8 more policies and run AED, CPR and NetComplete to obtain configurations that support all 16 policies.

In Figure 9, we show the average percentage of changes (*min-devices* and *min-lines*) made in both the real and synthetic networks. In Figure 10, we evaluate for the *min-pfs* and *preserve-templates* objective. The final two objectives are related to filters. To evaluate them, we use synthetic blocking policies on synthetic networks to allow filter updates. Overall, AED performs better for all objectives. We explain the results in detail below.

Comparison with manual updates. In Figure 9, we observe that compared to actual updates, AED significantly reduces the number of devices and lines affected. When executed with the *preserve-templates* objective, AED's updates did not violate any configuration uniformity, and neither did the actual updates. Although we don't know the actual management objective of the network operators when conducting their updates, these experiments show that AED matches or outperforms manual updates for many types of objectives.

Comparison with NetComplete. In Figure 9, we observe that NetComplete makes more changes than the other tools. It modifies almost all devices in the network, whereas AED on average can limit the number of modified devices to less than 30%. Additionally, for *preserve-templates* objective (Figure 10b), NetComplete creates as high as 25% template violation, whereas AED does not violate any template. This happens because NetComplete [17] does not support update size/scope objectives and only partially supports configuration structure objectives.

Comparison with CPR. Recall from 3.2 that CPR only supports update size objective. Hence, as shown in Figure 9, AED and CPR have similar results. However, for configuration structure and feature usage objectives (*preserve-templates* and *min-pfs*), CPR performs poorly. For example, as shown in Figure 10a, with *min-pfs* objective, AED never added more than 2 filters in any network. Whereas in some cases, CPR added 3X as many as AED. Additionally, for *preserve-templates* objective (Figure 10b), CPR creates the most template violations among all the tools.

This shows that a system such as CPR that bakes in a specific objective (*min-lines*) will find updates that may be valid but undesirable for an operator for multiple different management objectives. Whereas, AED's intrinsic expressiveness affords operators much greater flexibility.

9.1.2 Qualitative Analysis. Next, we surveyed operators from four different networks and asked them to rate three anonymized synthesis tools (AED, CPR and NetComplete) based on their coverage of management objectives. First, we showed operators a sample multi-site enterprise network and asked them to choose one or more of the following objectives: *preserve-templates*, *min-devices*, *avoid-redistribution*. Next, we showed them three iterations of the network, each built on top of the other. The first iteration satisfied two new blocking policies, the second satisfied a new reachability policy and the final satisfied a new waypointing policy. We specifically highlighted updates made by each synthesis tool and asked the operators to categorize these updates as good, average, or bad, w.r.t. their chosen objectives. We observe that in 50% of their answers,

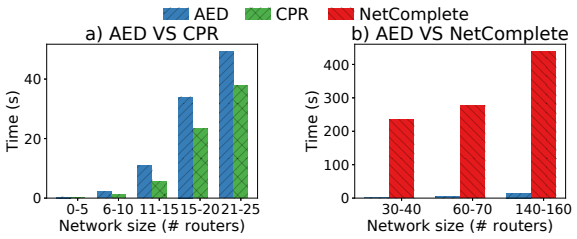


Figure 11: Performance on reachability policy

AED's updates were rated better than the other tools. And in 42% of their answers, AED's updates were rated equal to the other tools. In the remaining 8% of their answers, where AED's updates were rated lower, AED modified intermediate devices whereas operators preferred changing route-originating devices. However, note that by specifying this preference as an extra objective, AED could achieve the same update. We also observe that in 8% of their answers, AED's updates were rated bad. However, in those answers, other tools were also rated bad because all of them violated certain templates.

Overall these qualitative and quantitative results (vs. manual-, NetComplete-, CPR-based updates) show AED's practical utility.

9.2 Performance

Next, we examine AED's performance and scalability. In the remaining experiments, we group networks by their size and show average values of metrics of interest for each group.

Impact of network size. We first compare AED's performance with CPR [21] by running both tools with their intrinsic performance optimizations turned on across the 24 real datacenter networks. Figure 11a shows that for small networks ($\leq$ 10 routers), the time AED takes to compute updates is comparable to CPR. However, with increasing network size, AED's SMT-based control plane encoding becomes more complex, relative to CPR's graph-based encoding. Consequently, the time difference between CPR and AED in computing updates increases with network size. Recall however that CPR has poor management objective coverage (Table 1) and cannot satisfy configuration structure or feature usage objectives. Despite much greater generality, AED's performance does not significantly degrade compared to CPR.

To evaluate AED's scalability on larger networks, we use the NetComplete-generated configurations. We repeat the experiment from §9.1, where we start with configurations which support 8 reachability policies, and update them to satisfy 8 more policies. The objective is *min-devices*. The time taken to create the updated configurations is shown in Figure 11b. We observe that AED significantly outperforms NetComplete (i.e., clean-slate synthesis) by a factor of 10 to 100X. There are at least two reasons for this. The primary one is that, by taking the existing network as input, AED deals with a smaller search space, compared to NetComplete, where we made all configuration constructs symbolic. A secondary reason is that the NetComplete prototype deals with integer variables (e.g., for IP prefix, local-pref etc). It is synthesizing values for these variables in BGP configurations using an SMT solver, which contributes to

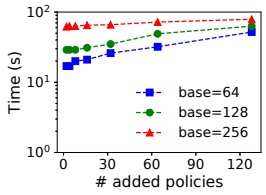


Figure 12: Impact of no. of policies.

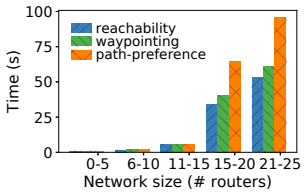


Figure 13: Impact of policy class.

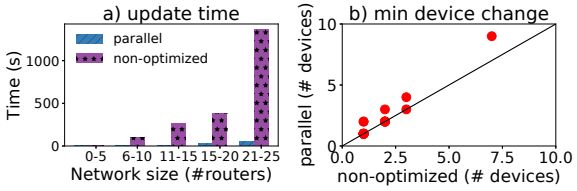


Figure 14: Impact of parallel solvers

slow down. AED's optimization strategy (§8) could improve NetComplete's performance too, but the overall performance achieved will still be poor due to the search space.

Impact of policy set size. Next, we measure how AED scales with the number of old and new policies in a changing network. We consider reachability policies on a 70 router network. For this experiment, we introduce two terms: (1) base policies, which represent the policies already configured in the network, and (2) added policies, which represent new policies to be added to the network. For the first experiment, we vary the number of base policies but keep the number of added policies fixed at 8. NetComplete scales poorly and takes more than 30 hours to handle just 64 base policies. On the other hand, AED scales linearly with the number of base policies and can synthesize a total of 1024+8 policies in 250 seconds. Next, we explore how AED scales as a function of the number of added policies. We run this experiment with three sets of base policies: 64, 128 and 256. Figure 12 shows that AED also scales linearly with an increasing number of added policies, irrespective of the number of base policies.

Impact of policy type. Finally, we evaluate AED's performance as a function of policy type. We evaluate on the real datacenter networks. Here, we assume the operator wants to update the policies supported by all networks by adding 5% new policies. We consider three classes of newly added policies: reachability, waypointing, and, path-preference. From Figure 13, we observe that at larger network sizes (>15), adding path-preference policies is the slowest to generate updates for. These policies need to ensure that (i) a less-preferred path is taken only when a more-preferred path is unavailable, and, (ii) ordering of routers in these paths is valid. This results in adding more variables and constraints to our formulation compared to the other two policies. However, the overall time to compute updates is still reasonable.

9.3 Impact of optimization strategies

We next evaluate the performance benefits and optimality impact of our strategies for improving AED's speed (§8). We assume that the strategies are leveraged in isolation. When employed together, their benefits compound. For brevity, we discuss results only for the real datacenter networks. We see similar and sometimes better speedup on the synthetic networks.

Parallel solvers. Solving the control plane update problem for multiple destinations in parallel yields updates in significantly lower time than solving the entire update instance at once. As shown in Figure 14a, performance speed up ranges from 10X to 300X under the *min-devices* objective. However, by not looking for a "globally" optimal update, we may sacrifice on update quality. In Figure 14b, we observe that there is only one network (with 15 routers) where running AED in parallel results in updates spread across 2 additional devices compared to using a single AED instance. Overall, parallelization performance speedup outweighs optimality loss.

Using boolean variables. A key optimization in our encoding was to replace certain integer variables with approximate boolean equivalents, because searching for suitable assignments for an integer variable can take a significant amount of time. In this experiment, we consider the *lp* variable. We use a synthetic setting (because this construct was not exercised in the networks of our dataset). Specifically, we use the topology shown in Figure 1 and evaluate how quickly AED computes updates for path-preference policies. The policy for all source-destination pairs is to prefer routes through C over routes through A. We set a higher *lp* value on router A (compared to C) in the configurations we provide to AED, such that the preference policies can only be satisfied by changing local preferences. This approach of using boolean variables instead of integers improves AED's performance by 3-10X.

Pruning configuration. Another key optimization was to prune irrelevant parts of the configuration for the given policy. This can simplify the MaxSMT problem by removing irrelevant conditionals and delta-variables from our encoding. We observe that this optimization improves AED's performance by 1.2-1.5X.

10 RELATED WORK

Network verification. Recent work [6–8, 15, 19, 20, 22, 28, 29, 36, 40] has shown how to detect errors in network control planes that lead to violations of important network-wide policies. Tools like Minesweeper [8], Bagpipe [46] and FSR [45] use SMT solvers for verification. However, these tools cannot model network updates.

Intent-based networking. The idea of using policies (network intent) to configure the network has has been well-adapted in both Software-Defined Networks (SDN) [5, 6, 37, 39, 41] and legacy networks [10, 17, 18, 38, 42]. Recently, many synthesis tools [10, 17, 18, 38, 42] automatically generate provably correct distributed control plane configurations, based on a set of high-level policies provided as input. However, these result in clean slate configuration updates.

Centralized control plane update. Wu et al. have designed an update system for a centralized control plane that uses provenance information to identify what caused the control plane to generate forwarding rules that violate some policies and suggests fixes to correct the problem [47]. However, like CPR, this system is far from

a complete solution to the problem of updating centralized control planes. In particular, Wu et al.'s system requires control planes to be written using a declarative programming language [34], and makes no guarantees on the optimality or interpretability of the updates.

Forwarding rule update. Some systems [25, 38] directly update forwarding rules. But this causes the control plane's view of the network to diverge from the current forwarding state. Future actions taken by the control plane may conflict with the updated forwarding rules, resulting in further policy violations and needing frequent forwarding updates.

11 DISCUSSION

Deploying updates. Deploying large number of configuration updates in a live network can lead to routing issues, like transient forwarding loops and black holes. It can also result in significant network downtime. Safely updating configurations in a live network is an important research problem and is part of our future work.

Encoding limitation. We presented our paper in the context of BGP and OSPF because they are very widely used. Although our encoding can be extended to model protocols like RIP and EIGRP, it cannot model stack-based protocols (e.g. MPLS, segment routing, etc) and open-flow rules. Our encoding also does not handle external routes, non-routing/forwarding-related configuration elements (e.g., SNMP, etc), and layer-2 features (e.g. mapping interfaces to VLANs, spanning tree, etc).

SMT output for special cases. If the network has multiple stable states/configurations to satisfy the policies and management objectives, then AED's SMT solver will choose one of those states. If there are conflicting policies or if the network cannot implement all the policies, then the SMT solver will generate an unsat solution. This indicates that the input (configurations and policies) is unsatisfiable.

12 CONCLUSION

Our survey on configuration change practices showed that along with correctness, operators care about a variety of management objectives. To support that, we propose a new synthesis tool called AED. AED encodes current configurations and its potential updates as a novel MaxSMT-based model whose structure is analogous to a syntax tree. AED allows operators to specify their objectives using a novel objective language and it encodes these objectives as soft-constraints. Finally, our evaluations over both real and synthetic network configurations show that AED computes updates fast and covers multiple management objectives.

Acknowledgments. We thank the anonymous reviewers for their insightful comments. We also thank the network engineers at Microsoft that provided us real network configurations. This work is supported by the National Science Foundation grants CNS-1637516 and CNS-1763512.

REFERENCES

[1] [n. d.]. AED source code. https://github.com/anubhavnidhi/batfish/tree/newrepair. ([n. d.]).

[2] [n. d.]. Cisco Network Services Orchestrator (NSO). https://developer.cisco.com/docs/nso. ([n. d.]).

[3] 2017. XML Path Language (XPath) 3.1. https://www.w3.org/TR/xpath-31. (March 2017).

[4] Anubhavnidhi Abhashkumar, Aaron Gember-Jacobson, and Aditya Akella. 2020. Tiramisu: Fast Multilayer Network Verification. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Ranjita Bhagwan and George Porter (Eds.).

[5] Anubhavnidhi Abhashkumar, Joon-Myung Kang, Sujata Banerjee, Aditya Akella, Ying Zhang, and Wenfei Wu. 2017. Supporting Diverse Dynamic Intent-based Policies using Janus. In *Proceedings of the 13th International Conference on emerging Networking EXperiments and Technologies*. ACM, 296–309.

[6] Carolyn Jane Anderson, Nate Foster, Arjun Guha, Jean-Baptiste Jeannin, Dexter Kozen, Cole Schlesinger, and David Walker. 2014. NetKAT: Semantic Foundations for Networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM, 113–126. https://doi.org/10.1145/2535838.2535862

[7] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. 2014. VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 282–293. https://doi.org/10.1145/2594291.2594317

[8] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2017. A General Approach to Network Configuration Verification. In *SIGCOMM*.

[9] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. 2019. Abstract interpretation of distributed network control planes. *Proceedings of the ACM on Programming Languages* 4, POPL (2019), 1–27.

[10] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2016. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 328–341.

[11] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. 2017. Network configuration synthesis with abstract topologies. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 437–451.

[12] Theophilus Benson, Aditya Akella, and David Maltz. 2009. Unraveling the Complexity of Network Management. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[13] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin T. Vechev. 2020. Config2Spec: Mining Network Specifications from Network Configurations. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Ranjita Bhagwan and George Porter (Eds.).

[14] Donald F. Caldwell, Anna C. Gilbert, Joel Gottlieb, Albert G. Greenberg, Gísli Hjálmtýsson, and Jennifer Rexford. 2004. The cutting EDGE of IP router configuration. *Computer Communication Review* 34, 1 (2004), 21–26. https://doi.org/10.1145/972374.972379

[15] Marco Canini, Daniele Venzano, Peter Perešíni, Dejan Kostić, and Jennifer Rexford. 2012. A NICE Way to Test OpenFlow Applications. In *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation (NSDI)*. USENIX Association, 127–140.

[16] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* (2008), 337–340.

[17] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*. USENIX Association.

[18] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. 2017. Network-Wide Configuration Synthesis. In *Computer Aided Verification - 29th International Conference (CAV)*.

[19] Seyed Kaveh Fayaz, Tushar Sharma, Ari Fogel, Ratul Mahajan, Todd D. Millstein, Vyas Sekar, and George Varghese. 2016. Efficient Network Reachability Analysis Using a Succinct Control Plane Representation. In *Symposium on Operating Systems Design and Implementation (OSDI)*.

[20] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. 2015. A General Approach to Network Configuration Analysis. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[21] Aaron Gember-Jacobson, Aditya Akella, Ratul Mahajan, and Hongqiang Liu. 2017. Automatically Repairing Network Control Planes Using an Abstract Representation. In *SOSP*.

[22] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane Analysis Using an Abstract Representation. In *SIGCOMM*.

[23] Aaron Gember-Jacobson, Wenfei Wu, Xiujun Li, Aditya Akella, and Ratul Mahajan. 2015. Management Plane Analytics. In *IMC*.

[24] Joel Gottlieb, Albert Greenberg, Jennifer Rexford, and Jia Wang. 2003. Automated provisioning of BGP customers. *IEEE Network Magazine* (November/December 2003).

[25] Hossein Hojjat, Philipp Rümmer, Jedidiah McClurg, Pavol Cerný, and Nate Foster. 2016. Optimizing horn solvers for network repair. In *Formal Methods in Computer-Aided Design (FMCAD)*.

[26] Alex Horn, Ali Kheradmand, and Mukul R Prasad. 2017. Delta-net: Real-time Network Verification Using Atoms.. In *NSDI*. 735–749.

[27] Siva Kesava Reddy K., Alan Tang, Ryan Beckett, Karthick Jayaraman, Todd D. Millstein, Yuval Tamir, and George Varghese. 2020. Finding Network Misconfigurations by Automatic Template Inference. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, Ranjita Bhagwan and George Porter (Eds.).

[28] Peyman Kazemian, George Varghese, and Nick McKeown. 2012. Header Space Analysis: Static Checking for Networks. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[29] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P. Brighten Godfrey. 2013. VeriFlow: Verifying Network-wide Invariants in Real Time. In *Symposium on Networked Systems Design and Implementation (NSDI)*.

[30] Hyojoon Kim, Theophilus Benson, Aditya Akella, and Nick Feamster. 2011. The evolution of network configuration: a tale of two campuses. In *Proceedings of the 11th ACM SIGCOMM Internet Measurement Conference (IMC)*, Patrick Thiran and Walter Willinger (Eds.).

[31] S. Knight, H.X. Nguyen, N. Falkner, R. Bowden, and M. Roughan. 2011. The Internet Topology Zoo. *Selected Areas in Communications, IEEE Journal on* 29, 9 (october 2011), 1765 –1775. https://doi.org/10.1109/JSAC.2011.111002

[32] Franck Le, Geoffrey G. Xie, Dan Pei, Jia Wang, and Hui Zhang. 2008. Shedding Light on the Glue Logic of the Internet Routing Architecture. In *SIGCOMM*.

[33] Hongqiang Harry Liu, Xin Wu, Wei Zhou, Weiguo Chen, Tao Wang, Hui Xu, Lei Zhou, Qing Ma, and Ming Zhang. 2018. Automatic life cycle management of network configurations. In *Proceedings of the Afternoon Workshop on Self-Driving Networks*. 29–35.

[34] Boon Thau Loo, Tyson Condie, Minos N. Garofalakis, David E. Gay, Joseph M. Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2009. Declarative networking. *Commun. ACM* 52, 11 (2009), 87–95. http://doi.acm.org/10.1145/1592761.1592785

[35] David A. Maltz, Geoffrey Xie, Jibin Zhan, Hui Zhang, Gísli Hjálmtýsson, and Albert Greenberg. 2004. Routing Design in Operational Networks: A Look from the Inside. In *SIGCOMM*.

[36] Tim Nelson, Andrew D. Ferguson, Michael J. G. Scheer, and Shriram Krishnamurthi. 2014. Tierless Programming and Reasoning for Software-Defined Networks. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 519–531. https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/nelson

[37] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. 2015. Pga: Using graphs to express and automatically reconcile network policies. In *ACM SIGCOMM Computer Communication Review*, Vol. 45. ACM, 29–42.

[38] Shambwaditya Saha, Santhosh Prabhu, and P. Madhusudan. 2015. NetGen: Synthesizing Data-plane Configurations for Network Policies. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research (SOSR)*. ACM, Article 17, 17:1–17:6 pages. https://doi.org/10.1145/2774993.2775006

[39] Robert Soulé, Shrutarshi Basu, Parisa Jalili Marandi, Fernando Pedone, Robert Kleinberg, Emin Gun Sirer, and Nate Foster. 2014. Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies*. ACM, 213–226.

[40] Radu Stoenescu, Matei Popovici, Lorina Negreanu, and Costin Raiciu. 2016. SymNet: Scalable symbolic execution for modern networks. In *SIGCOMM*.

[41] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2017. Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL 2017)*. ACM, New York, NY, USA, 572–585. https://doi.org/10.1145/3009837.3009845

[42] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. 2018. Synthesis of Fault-Tolerant Distributed Router Configurations. *Proc. ACM Meas. Anal. Comput. Syst.* (2018).

[43] Yu-Wei Eric Sung, Xiaozheng Tie, Starsky HY Wong, and Hongyi Zeng. 2016. Robotron: Top-down network management at facebook scale. In *Proceedings of the 2016 ACM SIGCOMM Conference*. ACM, 426–439.

[44] Bingchuan Tian, Xinyi Zhang, Ennan Zhai, Hongqiang Harry Liu, Qiaobo Ye, Chunsheng Wang, Xin Wu, Zhiming Ji, Yihong Sang, Ming Zhang, et al. 2019. Safely and automatically updating in-network ACL configurations with intent language. In *Proceedings of the ACM Special Interest Group on Data Communication*. ACM, 214–226.

[45] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. 2012. FSR: Formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Transactions on Networking (ToN)* 20, 6 (2012), 1814–1827.

[46] Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable verification of border gateway protocol configurations with an SMT solver. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*.

[47] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2015. Automated Network Repair with Meta Provenance. In *ACM Workshop on Hot Topics in Networks (HotNets)*.

A ENCODING ROUTING ALGORITHMS

The control plane advertises, computes, and selects routes based on parameterized algorithms. These algorithms reference the protocol parameter variables defined in §5. AED also creates control and data plane forwarding variables for each physical adjacency: e.g., $controlFwd_{A\to B}$ and $dataFwd_{A\to B}$ represent A's decision of whether to forward traffic to B. The value of the latter accounts for packet filters, whereas the value of the former only accounts for route selection.

Route advertisements. Receiving and sending of route advertisements is modeled using constraints expressed over symbolic route advertisements. A pair of constraints is created for each pair of neighboring routers in the physical topology.

```
1 if BGP_Y ∧ BGPAdj_{Y→X} ∧ activeLink_{Y→X} ∧ outBGP_{Y→X}.valid then
2       inBGP_{X←Y}.valid = filter_{Y→X}.allow
3       inBGP_{X←Y}.prefix = outBGP_{Y→X}.prefix
4       inBGP_{X←Y}.lp = filter_{Y→X}.lp
5       ...
6 else inBGP_{X←Y}.valid = false
```

Figure 15: Encoding of BGP receiving a route advertisement

Figure 15 shows the template for constraints that encode BGP's route receiving function. The template is parameterized by two neighboring routers X and Y, where X receives an advertisement from Y. Line 1 represents a set of conditions that must be met to receive a route: the routing protocol is enabled on the router (BGP_X); the routers are configured to have a routing adjacency ($BGPAdj_{Y\to X}$); the physical link connecting the routers is active ($activeLink_{Y\to X}$); and the advertisement sent by the adjacent router is valid. Certain fields of the symbolic advertisement depend on the route filter that applies to the adjacency (lines 2 and 4), while other fields are populated from the adjacent process's outgoing advertisement (line 3), independent of route filters. Note that the constraint references the protocol parameter variables defined in §5. This differs from NetComplete [17] and Minesweeper [8], where such values (e.g., prefixes and local preferences) are included directly in the receive (and send) constraints.

The constraints representing the sending of route advertisements have a similar structure, except matches and assignments are based on the best advertisement for that protocol (described below) and the policy being verified. For example, the constraint in Figure 16 encodes which BGP advertisements a router forwards (lines 1–6) and originates (lines 7–10).

```
1  if bestBGP_X.valid then                                  // Forward advertisement
2       outBGP_{X→Y}.valid = true
3       outBGP_{X→Y}.prefix = bestBGP_X.prefix
4       outBGP_{X→Y}.cost = bestBGP_X.cost + 1
5       outBGP_{X→Y}.lp = bestBGP_X.lp
6       ...
7  else if originate_{X→Y}.advertise then                  // Originate
8       outBGP_{X→Y}.valid = true
9       outBGP_{X→Y}.prefix = originate_{X→Y}.prefix
10      ...
11 else outBGP_{X→Y}.valid = false
```

Figure 16: Encoding of BGP sending a route advertisement

Route selection. To model route selection within and across protocols, the encoding includes an additional symbolic record for each routing process (e.g., $bestBGP_X$) and each router (e.g., $bestOverall_X$). A process's best record is set to the most preferred incoming and valid record: e.g., $bestBGP_X$ is equated from among multiple *in* records from different neighbors, based on which record has the highest lp and lowest $cost$. Similarly, a router's ($bestOverall_X$) best record is equated to whichever of its routing process ($bestBGP_X$ or $bestOSPF_X$) has the lowest ad. A router's best route (e.g., $bestOverall_X$) determines which interface (e.g., $controlFwd_{X\to Y}$) is used to reach the destination listed in the policy:

$$controlFwd_{X\to Y} \iff (inBGP_{X\leftarrow Y} = bestOverall_X) \vee (inOSPF_{X\leftarrow Y} = bestOverall_X)$$

Data forwarding. Finally, AED encodes whether X forwards packets to Y using a $dataFwd$ variable whose value is constrained based on the chosen route and any packet filters defined in the configuration. Again, these filter rules are encoded separately (§5) from the forwarding algorithm, which differs from existing tools [8, 17].

$$dataFwd_{X\to Y} \iff controlFwd_{X\to Y} \wedge pFil_{X\to Y}.allow$$

Figure 17: Encoding of data forwarding rules