Automatically Evolving Distributed Control Planes

Anonymous Author(s)

Abstract

When updating the configurations of distributed control planes, operators often attempt to meet a variety of management objectives, while also ensuring that all policies are correctly satisfied. Our tool, AED, automates this process. It generates correct updates that satisfy objectives specified in a high-level language. AED offers extensive coverage over both configuration and objectives. It models update as a maximum satisfiability (MaxSMT) problem, and uses a variety of domain-specific heuristics to quickly find solutions. Evaluations on both real and synthetic network configurations show that AED can update networks with tens of routers and hundreds of policies in under a minute. AED outperforms both hand-crafted updates and state-of-the-art tools in meeting management objectives.

1 Introduction

Changing, or *evolving*, control plane router configurations is a fact of life for network operators. For example, a study of configuration changes in over 850 networks operated by a large online service provider showed that about half of the networks have at least ten change events per month [16]. But, configuration evolution is a complex and multi-faceted task. First and foremost, operators must ensure that configuration updates made to satisfy new policies (e.g., enabling reachability between new subnets) do not violate existing policies (e.g., required waypoint, or middlebox, traversals). In addition to this basic *correctness* requirement, operators must account for a multitude of other management *objectives* when updating configurations. Important objectives suggested by our survey of 58 network operators (§2.1) include: making configurations easy-to-understand (e.g., by maintaining configuration similarity across devices), minimizing the number of routers affected, avoiding certain features (e.g., static routes), avoiding changes on routers with known hardware issues, etc.

Despite the seeming difficulty of evolving control plane configurations, our survey shows that in many networks today, updates are constructed by hand (§2.1). Even in cases where automation is applied, its scope is limited to specific configuration constructs; thus, manual updates are still common. Given the complexity of network configurations [6], it is unsurprising that manually evolving control plane router configurations often falls short—i.e., it introduces unanticipated bugs, or the updated configurations are not the "best possible" ones with respect to management objectives.

Researchers have developed several recent tools to *automate* evolution. Clean-slate synthesis tools [5, 31] produce brand-new, policy-compliant configurations in response to network evolution events, such as, new customers or new policies being added. On the other hand, incremental tools [10, 14] produce configuration patches. Unfortunately, all these tools suffer from one or more of the following drawbacks (§2.2).

First, clean-slate synthesis tools by definition ignore current configurations, and thus *cannot support objectives that depend on current configurations*, e.g. minimize devices updated. Second, existing tools operate on high-level encodings of network configurations and hence *cannot satisfy objectives tied to the configuration structure*. For example, CPR [14] models networks as hierarchical directed acyclic graphs, which abstracts away low-level details such as which configuration constructs exist on what routers. Finally, some tools, for example, NetComplete [10] require significant manual guidance to encode even simple objectives such as configuration similarity or avoiding certain configuration constructs.

We seek an incremental synthesis tool that can *automatically* compute correct control plane updates while *accommodating diverse configuration management objectives*. In addition, the tool should be *easy to use*, i.e., enable high-level specification of management objectives, and offer *fast performance* when applied to real networks. Our tool, AED, meets these requirements.

Encoding network syntax and semantics: AED's goal is to update the network configurations to be policy-compliant while satisfying various management *objectives*. This requires AED to reason about the syntax and semantics of both the network and its updates, and the difference between the current and the updated configurations. Modeling semantics is necessary for policy compliance. Some configuration updates (e.g., add/remove routing adjacencies) can each impact multiple policies; likewise, multiple configuration updates (e.g., ACLs and route filters) can have a say on whether a single policy is satisfied or not. Modeling syntax and difference in configurations are necessary for satisfying management objectives. Many objectives are defined over the structure (syntax) of router configuration, e.g. maintaining configuration similarity. Many objectives are defined over the scope of configuration updates, e.g. minimize the number of devices changed.

To model syntax, AED first represents the network as a syntax-tree and updates as changes (additions and removals) made to that syntax-tree. Next, AED introduces a novel SMT (Satisfiability Modulo Theory) based encoding of the network configurations. In AED's encoding, the structure of configuration constraints mirror the structure of the syntax tree. To model semantics, AED combines the encoding of configurations with the encoding of (protocol-specific) route computation and selection algorithms.

To model the difference between the current and the updated configurations, AED encodes configurations updates as *symbolic delta variables*. These variables basically encode *updates* w.r.t. specific constructs in the current configuration, e.g., changes to certain constants, and whether certain filters are enabled/disabled.

To ensure updates correctly account for all policies, AED performs careful *specialization* of update variables and related constraints. It clones and renames them depending on whether the variables impact an individual source-destination pair, an individual destination, or all sources and destinations.

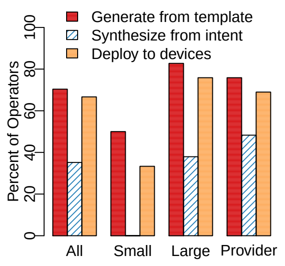
Objective Language: AED provides a high-level language for operators to easily program rich configuration-wide management objectives. This language allows operators to specify objectives as desired updates for specific (segments of) router configurations. These objectives cover the entire configuration space and are specified at various configuration granularities, e.g. rule-level, router-level, network-wide etc. To automatically compute updates that satisfy these objectives, AED compiles objectives into "soft constraints" on relevant delta variables, and creates a maximum satisfiability (MaxSMT) problem out of the aforementioned encoding; maximally satisfying the soft constraints results in updating only those delta variables that help meet the objective at hand.

Speed: AED incorporates several domain-specific heuristics to improve the speed with which a satisfying solution is found. These heuristics include: pruning configurations that are irrelevant to a given policy; leveraging the per-destination semantics of routing protocols to parallelize finding a solution; leveraging the fact that some integer variables (e.g., costs, local preference) are used to determine *ordering* among routes, which allows them to be discretized thereby reducing the complexity of finding satisfying values.

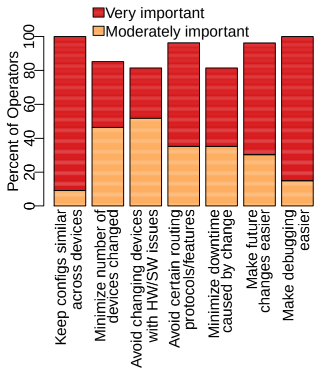
We implement AED in Java using 4K LOC. Our experiments using configurations from real data center networks show that AED effectively supports a variety of management objectives, and generates updates that are better than hand-written ones. We empirically show that state-of-the-art tools either cause 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. AED scales well with network/policy-set size.

2 Motivation

Satisfying all policies, both newly-provided and existing ones, correctly is the central goal of evolving 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 (§2.1). 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 [5, 10, 14, 17, 31] fail to meet operators' needs (§2.2).



(a) Automation



(b) Factors

Figure 1. Operator survey results

2.1 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 then conducted a broader survey of operators from an additional 54 networks[1] 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 intents (Figure 1a). 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 to rate the importance of these seven factors when they change their network. Figure 1b shows for each factor the percentage of operators that reported the factor was moderately or very important for at least one type of change in their network. 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 the changes are easily verified and reversible.

Configuration similarity. In our survey results, the most important factor (very important to 90% of operators) is keeping configurations similar across 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 [6].

Interestingly, our survey results show that configuration similarity is very important even for operators that reported

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

using automation to generate changes from templates or synthesize changes from intents. This is in large part because not all changes in a network are automated. For example, one operator we interviewed said they recently started synthesizing BGP import/export filters from intents, 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 [16]. 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 interview indicated some 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. Prior studies have found that operators may avoid certain routing protocols due to their additional licensing costs [6], or because some features (e.g., route redistribution) can introduce routing loops [23]. In our survey, 61% of operators indicated it was very important to avoid using certain protocols/features. 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.

2.2 Limitations of existing tools

Inspired by our operator study, we seek a tool that automatically generates configuration updates satisfying new and old policies, while accounting for a range of management objectives. Although several classes of configuration synthesis tools exist, they cannot be used effectively because they consider only a subset of management objectives and configuration components (Table 1).

Templates. According to our survey (Figure 1a) and prior studies [6], one widely used class of tools generate portions of configurations from specialized templates [10, 17]. These tools fill-in-the-blanks in a pre-defined configuration segment with appropriate prefixes, link weights, BGP communities, etc. to satisfy new policies—e.g., enable a new subnet 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 configurations; ignoring the semantics of other parts of the configuration can lead to violations of other policies.

		Management Objectives			
	Synthesizer	**Config structure**	**Update size/scope**	**Features used**	**Config components**
Clean-slate	Zeppelin [31]	○	○	◐	Most
	Synet [11]	○	○	○	Most
	Propane [5]	○	○	○	BGP only
Incremental	NetComplete [10]	◐	○	◐	Most
	Jinjing [32]	○	●	○	ACLs only
	CPR [14]	○	●	○	Many
	AED	●	●	●	Most

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

Clean-slate synthesizers. These tools [5, 31] take as input the network topology, a set of policies [7], 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 [31] 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 [5] only synthesizes BGP configurations.

Incremental synthesizers. These tools [10, 14, 32] 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 [14] uses a graph-based model of the network's control plane and produces updates that change the fewest lines of configuration (which is modeled via changes made to edges in the graph-based model). However, CPR cannot meet structural (e.g., maintaining similarity across devices) or feature usage objectives. The underlying reason is that CPR uses a high-level model that captures the paths that materialize in a network, but not the low level configuration *syntax* nor how various configuration statements impact said paths.

NetComplete [10] 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 satisfiability modulo theory (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, which is very challenging given the complexity of today's configurations [6]. Consequently, satisfying even simple objectives such as maintaining configuration similarity, or minimizing the use of certain features, requires significant manual guidance.

Some incremental synthesizers focus on a narrow swath of configuration components: e.g., Jinjing only repairs ACLs [32].

In summary, existing configuration synthesis tools do not fully meet operators' needs w.r.t. configuration changes.

3 Our Approach: AED

Automatically generating configuration updates that satisfy forwarding policies and management objectives 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 collection of syntax-tree additions and removals*. By modeling configurations (and updates) at a syntactic level—instead of a higher-level intermediate representation [4, 5, 14, 31]—we can easily reason about the structure and contents of potential configurations (*ii*). By modeling updates as additions and removals—instead of a collection of values [10]—we can easily reason about the delta between current and potential configurations (*iii*). Finally, by combining our model of configuration with a model of (protocol-specific) route computation and selection algorithms, we can reason about configurations' semantics (*i*).

The fundamental challenge is *determining which nodes to add and remove* from the syntax-tree. 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, 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 2, where each leaf node represents a single line of configuration.

The semantics of these elements is complex and partially depends on (protocol-specific) route computation and selection algorithms. For example, which routes a protocol instance advertises to its neighbors depends on: which routes the protocol is configured to originate (*iii*), which routes the protocol is configured to filter (*iv*), how the protocol selects (i.e., ranks) routes, whether the routes must be the most preferred among all protocols (e.g., BGP and EIGRP only advertise routes that are installed in a router's main routing information base), etc. Reconciling configuration (and protocol) semantics with forwarding policies is non-trivial.

Consequently, we formulate a system of satisfiability modulo theory (SMT) constraints whose solution is a correct (w.r.t. forwarding policies) and optimal (w.r.t. management objectives) set of syntax-tree additions and removals. The system of constraints includes: (*i*) configuration constraints (§4.2), which encode the structure and partial semantics of

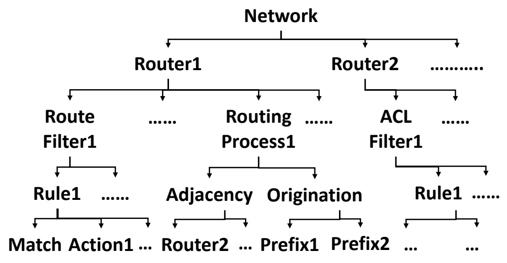


Figure 2. Configuration syntax tree

the five configuration elements described above; (*ii*) protocol constraints (§4.3), which fully encode the semantics of configuration elements in the context of route propagation and selection algorithms; (*iii*) policy constraints (§5), which encode forwarding policies, and (*iv*) management constraints (§6), which encode management objectives.

Syntax tree additions and removals are modeled using a special category of free variables we call *delta variables*. 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.) The delta variables are included in configuration constraints to model the semantic impact of changes. For example a packet filter can be encoded using nested if-then-else statements, where each statement's conditional expression includes a filter rule's match criteria and a delta variable corresponding to the possible addition/removal of that rule. Assigning a true (or non-zero) value to one of these variables effectively enables/disables the rule, which impacts whether (a subset of) packets are blocked. The delta variables are used in management constraints to model the impact of changes on configuration structure, update size/scope, and feature usage. We introduce a high-level language (§6.1) for operators to express management objectives in terms of actions (ELIMINATE, EQUATE, or NOMODIFY) on regions of the syntax tree. These expressions are translated into SMT constraints over delta variables.

The next four sections describe AED in detail. §4 explains how AED encodes the structure and semantics of network configurations. §5 describes how AED handles a multitude of forwarding policies. §6 introduces our management objective language and explains how AED accounts for these objectives. Finally, §7 presents domain-specific optimizations for speeding up AED.

4 Encoding network

In this section, we describe the symbolic variables and constraints AED uses to model the structure and semantics of current and potential configurations. We use the example network in Figure 3 to help illustrate AED's design.

4.1 Symbolic variables

AED relies on three categories of symbolic variables.

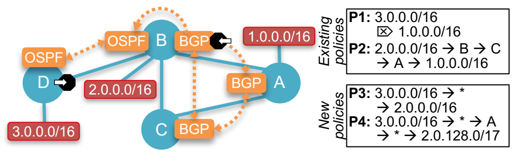

OSPF
B
BGP
1.0.0.0/16
OSPF
D
2.0.0.0/16
BGP
A
3.0.0.0/16
C
BGP
Existing policies
P1: 3.0.0.0/16 ⌧ 1.0.0.0/16
P2: 2.0.0.0/16 → B → C → A → 1.0.0.0/16
New policies
P3: 3.0.0.0/16 → * → 2.0.0.0/16
P4: 3.0.0.0/16 → * → A → * → 2.0.128.0/17

Figure 3. 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; D blocks outgoing packets from 3.0.0.0/16

Delta variables encode configuration updates, making them most important variables in AED. 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 configuration constraints (§4.2) to model the semantic impact of changes and in management constraints (§6.2) to model the impact of changes on configuration structure, update size/scope, and feature usage. A delta variable's name is determined by the corresponding node's location in the syntax-tree: e.g., $rm.R1.RFilter1.Rule1.Match$ corresponds to the left-most leaf node in Figure 2. This naming convention enables AED to quantify a change's impact on configuration structure; we discuss this in more detail in §6.

Protocol parameter variables represent configuration values that impact the advertisement and selection of routes: e.g., whether a protocol is enabled, whether a routing adjacency is defined, whether a prefix is originated, whether a route/packet is allowed, etc. 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 [10]: NetComplete encodes them as either 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.

Control plane output variables represent the route advertisements and forwarding rules produced by routing protocols. These are partially influenced by configuration values, which are partially influenced by delta variables. AED creates records of symbolic values for each potential routing adjacency: e.g., $outBGP_{A \to B}$ represents BGP advertisement sent from A to B and $inBGP_{B \leftarrow A}$ represents 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). Additionally, every symbolic record has a boolean field that indicates whether the advertisement is *valid*. AED creates control and data plane forwarding variables for each physical adjacency: e.g., $controlFwd_{A \to B}$ and $dataFwd_{A \to B}$ represents 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. Forwarding policies are expressed as constraints on data forwarding variables (§5).

4.2 Encoding configuration

We now describe how AED models each of the five key router configurations elements (Figure 2).

Routing protocols and adjacencies. AED introduces 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., $BGPAdj_{A \to B}$). Each variable is constrained to be true (false) if the protocol/neighbor relationship is (not) included in the configuration. AED models adjacency updates using $add.*.Adj.*$ and $rm.*.Adj.*$ delta variables for each possible adjacency that is absent or present, respectively, in the current configurations. For example, since there is a route adjacency between A and B in Figure 3, $BGPAdj_{A \to B}$ will be constrained as follows:

$$\boxed{BGPAdj_{A \to B} \iff \text{true} \wedge \neg rm.A.BGP1.Adj.B}$$

Route filters. Route filters define a set of match-action rules that are applied to advertisements a router imports/exports. Since our goal is to model configuration updates as syntax-tree additions/removals, we structure filter constraints in a manner that mirrors the structure of route filters as embodied by the syntax tree. For example, Figure 4 shows the encoding of B's route filter that is applied to BGP advertisements from A. Each rule specifies a prefix to match with the BGP advertisement from A. The actions associated with each rule: (i) dictate whether the advertisement is allowed or dropped, and (ii) set the value of certain fields in the BGP advertisement (unspecified fields get default value). AED encodes filters using filter variables ($filter_{B \to A}$), and these match-action rules basically assign values to the fields of these variables.

AED can add, remove and modify 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 cost and metric values assigned to routes (e.g., ad and lp) in the BGP advertisement (line 6). AED represents these filter cost constraints as the sum of the current constant and the integer delta variable. These integer variables determine which routes are preferred. Finally, to model rule additions, AED prepends an additional conditional statement (line 1-3) predicated on $add.*.rFil.*$ delta variables.

```
if $match(outBGP_{A\to B}.prefix, \text{POLICY.DSTPREFIX}) \land$
  $add.B.rFil_A.new.Match$ then
    $filter_{B\to A}.allow = B.rFil_A.new.Allow$
    ...
else if $match(outBGP_{A\to B}.prefix, 1.0.0.0/16) \land$
  $\neg rm.B.rFil_A.1.Match$ then
    $filter_{B\to A}.allow$ = false $\lor B.rFil_A.1.Allow$
    $filter_{B\to A}.lp$ = 100 + $B.rFil_A.1.lp$ //by default lp is 100
    ...
else
    $filter_{B\to A}.allow$ = true $\land \neg B.rFil_A.2.Allow$
    $filter_{B\to A}.lp$ = 20 + $B.rFil_A.2.lp$
```

Figure 4. Encoding of import filter on B

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

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 5 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.

```
if $add.A.BGP_B.Orig.new \land$
  $match(outBGP_{A\to B}.prefix, \text{POLICY.DSTPREFIX})$ then
    $originate_{A\to B}.advertise$ = true
    $originate_{A\to B}.prefix$ = POLICY.DSTPREFIX
else if
  $match(\text{POLICY.DSTPREFIX}, 1.0.0.0/16) \land \neg rm.A.BGP_B.Orig.1$
  then
    $originate_{A\to B}.advertise$ = true
    $originate_{A\to B}.prefix$ = 1.0.0.0/16
else
    ...
```

Figure 5. Origination of prefix from router A

Packet filters. ACLs also consist of a set of match-action rules and are encoded similar to route filters. For example, Figure 6 shows the encoding for the outbound ACL on D that blocks traffic from $3.0.0.0/16$. To encode updates related to ACLs, we use an approach similar to our handling of route filters: we introduce $rm.*.aFil.*$, $add.*.aFil.*$, and $*.aFil.*.Allow$ delta variables.

```
if .. then
else if $match(\text{POLICY.SRCPREFIX}, 3.0.0.0/16) \land$
  $\neg rm.D.aFil_B.1.Match$ then
    $acl_{D\to B}.allow$ = false $\lor D.aFil_B.1.Allow$
else if ... then
```

Figure 6. Match-action rules for ACL on D

4.3 Encoding control/data plane algorithms

The control plane advertises, computes, and selects routes based on parameterized (protocol-specific) algorithms. These algorithms reference the protocol parameter variables defined in §4.2.

Route advertisements. Route import and export functions are modeled using constraints expressed over symbolic route advertisements. A pair of import and export constraints is generated for each pair of neighboring routers in the physical topology.

```
1 if $BGP_Y \land BGPAdj_{Y\to X} \land activeLink_{Y\to X} \land$
    $outBGP_{Y\to X}.valid$ then
2     $inBGP_{X\leftarrow Y}.valid = filter_{Y\to X}.allow$
3     $inBGP_{X\leftarrow Y}.prefix = outBGP_{Y\to X}.prefix$
4     $inBGP_{X\leftarrow Y}.lp = filter_{Y\to X}.lp$
5     ...
6 else $inBGP_{X\leftarrow Y}.valid$ = false
```

Figure 7. Encoding of BGP incoming record

Figure 7 shows the template for constraints that encode BGP's import 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 import 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 configuration-related variables defined in §4.2. This differs from NetComplete [10] and Minesweeper [4], where configuration values (e.g., prefixes and local preferences) are included directly in the import (and export) constraints.

The constraints representing export functions 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 8 encodes which BGP advertisements a router forwards (lines 1–6) and originates (lines 7–10).

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

Figure 8. Encoding of BGP export policies

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., $bestOverall_X$). Similar to NetComplete and Minesweeper, a process's best record is set to the most preferred, valid incoming record: e.g., $bestBGP_X$ is equated from among multiple in from different neighbors, based on which record has the highest lp and lowest $cost$ values. Similarly, a router's best record is set to the best record from the most preferred protocol: e.g., $bestOverall_X$ is equated with whichever of $bestBGP_X$ 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:

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

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 ACLs defined in the configuration. Again, the ACL rules are encoded separately (§4.2) from the forwarding algorithm, which differs from Minesweeper.

$$\boxed{dataFwd_{X\to Y} \iff controlFwd_{X\to Y} \land acl_{X\to Y}.allow}$$

Figure 9. Encoding of data forwarding rules

5 Encoding policies

AED can compute updates for a wide range of policies, including: reachability, 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 3 is encoded as:

$$\boxed{dataFwd_{B\to C} \land dataFwd_{C\to A}}$$

Figure 10. Encoding of waypoint policy $P2$

5.1 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 3. Due to the ACL on D, 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 the ACL on D, 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,[2] 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 4 and 5), import and export functions (e.g., Figure 7 and 8), and route selection (not shown) are predicated on the route advertisements, we also include per-prefix versions of these constraints and variables. The data forwarding decisions (e.g., Figure 9) depend on ACLs, 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. We refer to the above as *specialization*. All of

[2]If policies' prefixes partially overlap, we can subdivide policies into non-overlapping packet equivalence classes [19].

```
objective := action | objective objective
           | for var in selector: objective
           | objGroup: objective
   action := NoModify(name) | Eliminate(name)
           | Equate (name for var in selector)
 selector := RouteFilters(nameRegex,deviceRegex)
           | Rules(filterName) | ...
```

Figure 11. Grammar for management objectives

these additional constraints and variables for specialization—needed for correctness—substantially increases the size of the SMT problem; we address this in §7.

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 (§4.2)—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 4) impacts a specific prefix. When we introduced filter ($add.{*}.rFil.{*}$) and origination ($add.{*}.Orig.{*}$) delta-variables in §4.2, 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 ACL variables: $add.{*}.aFil.{*}.Match$ and ${*}.aFil.{*}.Allow$. With $rm.{*}.rFil.{*}$, $rm.{*}.Orig.{*}$, and $rm.{*}.aFil.{*}$, 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.

6 Management Objectives

In §2.1, we showed that in addition to policy correctness, operators consider many other factors during configuration updates. These include maintaining configuration similarity across devices, avoiding the use of certain protocols/features, etc. These factors are termed as management objectives. To ensure the configuration updates computed by AED satisfy these additional factors (§2.1), we introduce a high-level language (Figure 11) for expressing management objectives. These objectives are then compiled into boolean formulas and appended to the SMT encoding (§4.2 and §4.3) as soft constraints.

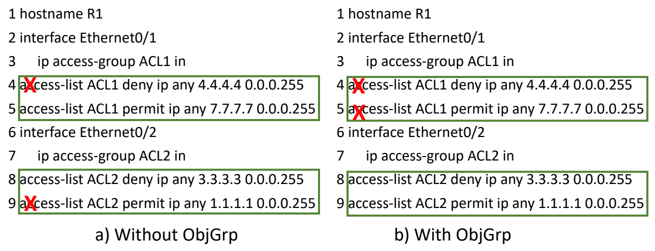
6.1 Objective Language

Objectives as actions. The management objectives basically dictate what type of configuration updates are preferred. AED's objective language allows operators to specify objectives as desired *action* (update) for specific (segments of) router configurations. AED's overarching goal is to satisfy as many objectives as possible.

Based on our survey results (§2.1) as well as the capabilities of other tools [14], we identify that objectives can be satisfied using three actions. These desired actions are: eliminate segments of configuration (Eliminate), make a set of segments consistent across different devices (Equate), or avoid changes altogether (NoModify). AED currently supports these actions and it encodes them in its SMT model using boolean operators (described later in §6.2). As long as an action can be encoded using boolean operators, AED can be easily extended to support them. For example, an action that "prefers changes" is the opposite of NoModify. Since its SMT encoding is similar to NoModify with an additional negation operator, AED can easily support it.

Selector Functions. In §3, we represent the network as a syntax tree. The objectives in §2.1 apply to various segments of the network, and hence the syntax tree. Some apply to a specific router, e.g. avoid changing routers with hardware issue. And some to a particular feature, e.g. avoid protocol with additional licensing costs. To model objectives that span multiple nodes of the tree, an naive approach will be to list each objective on each node as a separate expression. This is a tedious process. To make AED simpler to use, we introduce multiple $Selector$ functions. These functions are used by operators to select groups of related nodes of the syntax tree. For example, RouteFilters($nameRegex$, $deviceRegex$) provides the names of all route filters whose name matches $nameRegex$ and which are defined on one or more devices whose names match $deviceRegex$. Similarly, Rules($filterName$) provides the identifiers for all rules associated with a route or packet filter named $filterName$, and Actions($ruleName$, $filterName$) provides the names of actions in a specific rule in a specific filter. For simplicity, we assume that if two different routers have a filter with the same name, the filter's rules will be the same across the two routers.

Objective Group. As mentioned in §3, AED restricts updates (and hence actions) to the leaf nodes of the syntax tree. However, operators may desire objectives at a higher granularity, i.e. feature-level, router-level etc. Modeling these objectives at leaf-level granularity is incorrect. For example, assume an operator has two main objectives - avoid changing $ACL1$ and $ACL2$. To model them at leaf-level granularity, the operator has to write multiple NoModify actions on each match and action rules of $ACL1$ and $ACL2$. AED aims to maximally satisfy objectives by minimizing the number of violated NoModify. However, it does not differentiate rules from $ACL1$ and $ACL2$. An update that violates X number of NoModify from only $ACL1$ is equally preferable to an update that violates X number of NoModify spread across $ACL1$ and $ACL2$. However, the former violates only one objective, whereas the latter violates both.



```
1 hostname R1
2 interface Ethernet0/1
3    ip access-group ACL1 in
4 access-list ACL1 deny ip any 4.4.4.4 0.0.0.255
5 access-list ACL1 permit ip any 7.7.7.7 0.0.0.255
6 interface Ethernet0/2
7    ip access-group ACL2 in
8 access-list ACL2 deny ip any 3.3.3.3 0.0.0.255
9 access-list ACL2 permit ip any 1.1.1.1 0.0.0.255
```

a) Without ObjGrp

```
1 hostname R1
2 interface Ethernet0/1
3    ip access-group ACL1 in
4 access-list ACL1 deny ip any 4.4.4.4 0.0.0.255
5 access-list ACL1 permit ip any 7.7.7.7 0.0.0.255
6 interface Ethernet0/2
7    ip access-group ACL2 in
8 access-list ACL2 deny ip any 3.3.3.3 0.0.0.255
9 access-list ACL2 permit ip any 1.1.1.1 0.0.0.255
```

b) With ObjGrp

Figure 12. Impact of objective group ($objGroup$). **X** represents violation of NoModify. In the left figure, NoModify is applied without grouping. In the right figure, it is applied with grouping.

Objective	Constraints

Keep configs (e.g. ACLs) similar across devices	for f in PacketFilters(.*, .*):
	objGroup:
	  for r in Rules(f):
	   Equate($d.f.r.match$ for d in Devices(.*))
	   for a in Actions(.*, r, f):
	    Equate($d.f.r.a$ for d in Devices(.*))

Minimize number of devices changed	for d in Devices(.*):
	objGroup:
	  for f in RouteFilters(.*, d)):
	   for r in Rules(f)):
	    NoModify($d.f.r.match$)
	    for a in Actions(.*, r, f)):
	     NoModify($d.f.r.a$)
	  for f in PacketFilters(.*, d):
	   // repeat code from route filters
	  for p in Protocols(static, d):
	   for n in Adjacencies(p, d):
	    NoModify($d.p.n$)
	   for o in Originated(p, d)):
	    NoModify($d.p.o$)

Avoid changing devices with HW/-software issues	Similar to previous objective but replace first line with: for d in Devices(R1—R2—R3):

Avoid protocols/features (e.g., static routes)	for d in Devices(.*):
	objGroup:
	  for p in Protocols(Static, d)):
	   for n in Adjacencies(p, d)):
	    Eliminate($d.p.n$)
	   for o in Originated(p, d)):
	    Eliminate($d.p.o$)

Table 2. Encoding important management objectives

To model objectives at higher-granularity, AED uses a special objective group ($objGroup$) operator. $objGroup$ groups actions applied to individual configuration segments as a single objective. An $objGroup$ is satisfied if every action within the $objGroup$ is satisfied; if one or more actions are not taken, then the $objGroup$ is unsatisfied. For loops can be used to define multiple actions within a $objGroup$ and/or to define multiple $objGroups$. Figure 12 shows how $objGroups$ impacts the aforementioned ACLs examples.

Table 2 shows how to express the management objectives discussed in §2.1. Depending on how actions are grouped, they result in different semantics for the update that AED computes. Suppose "avoiding static routes" (row 4 in Table 2) was expressed by moving `objGroup` to the beginning:

```
objGroup: for d in Devices(.*):
  // include last 5 lines from row 4 in Table 2
```

The difference between these two expressions is the granularity at which we seek to eliminate static routes. The expression in the fourth row of Table 2 seeks to (maximally) eliminate all static routes on a per-router basis while the above expression seeks to eliminate static routes across the entire network. With the former, even if all static routes cannot be eliminated on one router, AED will still try to eliminate all static routes on other routers. In contrast, the latter is all-or-nothing: either AED eliminates static routes on all routers or it leaves (some) existing static routes in place and potentially even adds new static routes. To eliminate as many instances of static routes across the network as possible, even if they are spread across devices, an operator could put the *objGroup* directive immediately before each of the ELIMINATE actions. The overall point is that AED's language offers operators great flexibility to control the scope of their objectives (per-feature, per-router, or network-wide).

Moreover, operators can assign weights to different objectives–overriding AED's default of assigning equal weight to each objective–if some objectives are preferred over others.

Finally 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 the grammar in Figure 11.

6.2 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 the target policies (§5) along with configurations (§4.2) and control/data planes' behavior (§4.3); these are necessary to ensure that the computed updates are correct. Soft constraints are operator management objectives that are translated into boolean formulas.

As shown in §4.2, delta variables, like $add/rm.*.Adj.*$, represent configuration updates. Given these delta variables, we encode each action as: NOMODIFY is the negation of the disjunction of every add/remove variable; ELIMINATE is the conjunction of every remove variable and the negation of every add variable; EQUATE is a conjunction of equality comparisons over a set of variables.

For example, the objective in the first row of Table 2 is transformed into the following soft constraints:

$$rm.D.aFil_B.1.Match = rm.B.aFil_C.1.Match = .. \wedge$$
$$D.aFil_B.1.Allow = B.aFil_C.1.Allow = .. \wedge$$
$$rm.D.aFil_B.2.Match = rm.B.aFil_C.2.Match = .. \wedge ..$$

Recall that *objGroup* are used to group actions in an all-or-nothing manner. This is realized by unrolling the `for` loops and conjuncting the relevant variables into a single boolean formula.

7 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 [14]. 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. For example, only those packet filter rules that match a given destination will impact reachability to that destination. I.e., line 4 to 7 in Figure 4 are irrelevent for policy **P3** from Figure 3.

The inclusion of irrelevant conditionals within origination, route filter, and packet filter constraints and 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 4 is encoded as the following for policy **P3**:

```
1 if $match(outBGP_{A\to B}.prefix, \text{POLICY.DSTPREFIX}) \wedge add.B.rFil_A.new.Match$ then
2     $filter_{B\to A}.allow = B.rFil_A.new.Allow$
3     ...
4 else
5     $filter_{B\to A}.allow = \text{true} \wedge \neg B.rFil_A.2.Allow$
6     $filter_{B\to A}.lp = 20 + B.rFil_A.2.lp$
```

Grouping policies based on a destination address. As discussed in §5, 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 §5 with applying AED separately for $P1$ and P3 in Figure 3, can be addressed by updating D's ACL 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 (§8.3).

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 (§4.2). However, each single 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.

8 Evaluation

We prototype AED atop Minesweeper [4]. Minesweeper is a control plane verification tool. Minesweeper uses Batfish [13] to parse router configurations, and the Z3 SMT solver [9] to encode and solve the underlying SMT formulation. We add our objective language and modify Minesweeper to incorporate our detailed (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 [14] and NetComplete [10]. We use NetComplete with all configuration constructs made symbolic.[3]

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.

[3]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 (§2.2).

To evaluate AED's ability to update real networks, we use configuration snapshots from 102 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 [4]. We only infer reachability policies, because inferring more complex policies, e.g., waypointing, has substantial complexity, rendering Minesweeper unable to terminate. Minesweeper was able to infer all reachability policies for 47 of the 102 networks in our dataset; the remaining 55 networks heavily use VLANs, which Minesweeper cannot currently model. Minesweeper detected at least one policy change across snapshots in 24 of the 47 networks. These 24 networks have between 2 and 24 routers, and support 50 to 6600 reachability policies.

To evaluate AED's scalability and performance, we use synthetic BGP configurations generated by NetComplete [10] for 10 network topologies of varying sizes from the Internet topology zoo [22]. In all our experiments with these configurations, we group networks according to their size (30-40, 60-70, and, 140-160 routers) and show average values of metrics of interest for each group.

For brevity, we show most results by evolving networks to support new reachability policies. We show results for adding other policy types in §8.2.

8.1 Management Objectives and Utility

AED's grammar (§6.1) and update-oriented network model (§4.2 and §4.3) allow operators to optimize updates for a variety of management objectives. We study the effectiveness with which AED supports such objectives and its practical utility.

Comparison with manual updates: We first compare the optimality of AED's updates to manual (or semi-automated) updates w.r.t two management objectives: minimize the number of $devices$ changed and preserve configuration $homogeneity$. We use the configuration snapshots from the 24 real datacenter networks where Minesweeper detected differing reachability policies. Given a network's "before" snapshot, we run AED with each of these objectives to evolve 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 [16], the "before" configurations.

We compute the number of devices changed across the actual snapshots by "diff'ing" the two snapshots. We only count devices where changes were related to routing/forwarding, and ignore changes due to SNMP and authentication. To compute changes in homogeneity, we group configurations based on their ACL and Filter rules in the "before" snapshot. We then compare these segments of the configuration across the actual "before/after" snapshots.

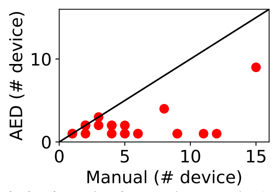


Figure 13. Minimize devices changed: AED vs Manual

The results for the *min-devices* objective is shown in Figure 13. We observe that compared to actual updates, AED's updates significantly reduce the number of devices affected. When executed with the *preserve-homogeneity* objective, AED's updates did not violate any configuration uniformity, and neither did the actual updates. We observed qualitatively similar results for other objectives.

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 *any* objective.

Comparison with NetComplete. Next, we show why NetComplete [10], with all configuration constructs made symbolic, is a poor choice for updating configurations. Here, we use synthetic configurations. To ensure fair comparisons, we limit AED to update only those configuration constructs that can be supported by NetComplete. For each network, we synthesize configurations which support 8 randomly-generated reachability policies. We then run AED on these configurations to satisfy 8 more policies, and we run NetComplete with all 16 policies. In this experiment, we assume the operator's objective is *min-devices*. The number of configurations changed by each tool is shown in Figure 14a. We observe that NetComplete modifies almost all devices in the network, whereas AED can limit the number of modified devices to 25 even on a 158 router network.

Comparison with CPR. Next, we show the effectiveness of accommodating different configuration management objectives in the AED encoding, relative to CPR [14], a state-of-the-art incremental synthesis tool. Recall from 2.2 that CPR has limited objective expressiveness on account of relying on a higher-level control plane abstraction. In this experiment, we assume the operator's objective is to minimize the use of ACLs (*min-acls*). We assume the operator wants to evolve each of the 24 real data center networks to block 20% of previously reachable subnets. We run AED separately with the objectives *min-acls* and *min-lines* (minimize lines of configuration changed, which is CPR's sole objective). We measure the number of ACLs added to configurations to satisfy the newly added policies. From Figure 14b, we observe that using *min-lines* always results in updates that add more ACLs than using *min-acls* directly. Optimizing for *min-acls* never added more than 2 ACLs in any network, whereas in some cases *min-lines* added 3X as many as *min-acls*. 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 from a management perspective. AED's intrinsic

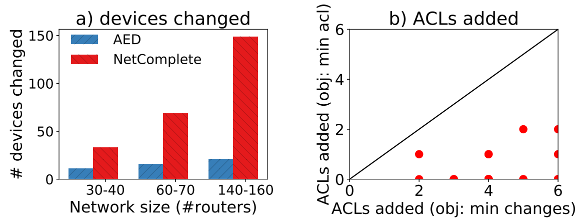


Figure 14. Update Quality: AED vs NetComplete and CPR

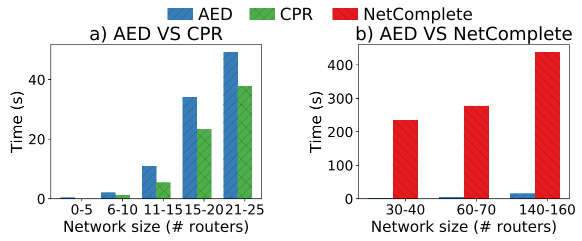


Figure 15. Performance on reachability policy

expressiveness affords operators much greater flexibility in controlling the structure of their updated configurations.

Overall these comparative results (vs. manual-, NetComplete-, CPR-based updates) shows AED's great practical utility.

8.2 Performance

Next, we examine AED's performance and scalability.

Comparison with CPR. We first compare AED's performance with CPR [14] by running both tools with their intrinsic performance optimizations turned on across the 24 real datacenter networks. Henceforth, for all our experiments with real networks, we group networks according to their size and show the average value for that group.

Figure 15a shows 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 sacrifices coverage (over objectives and configuration constructs). Despite much greater generality, AED takes only a few tens of additional seconds to compute updates.

Impact of network size. To evaluate AED's scalability on larger networks, we use NetComplete-generated configurations. We repeat the experiment from §8.1, 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 15b. 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

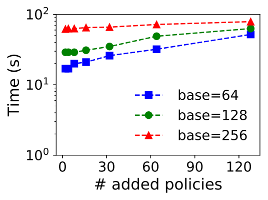


Figure 16. Impact of no. of policies.

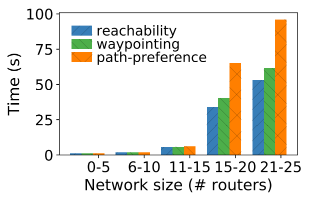


Figure 17. Impact of policy class.

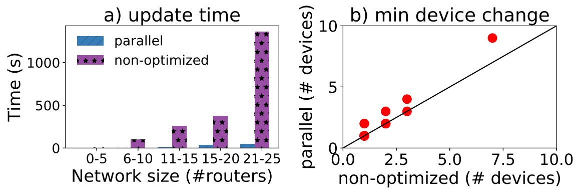


Figure 18. Impact of parallel solvers

using an SMT solver, which contributes to slow down. AED's optimization strategy (§7) 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 an evolving 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 16 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 47 real datacenter networks, where Minesweeper was able to infer policies. 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 17, 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.

8.3 Impact of Optimization Strategies

We next evaluate the performance benefits and optimality impact of our strategies for improving AED's speed (§7). We assume that the strategies are leveraged in isolation of each other. When employed together, their benefits compound.

Parallel solvers. Again, we use the same 24 real data center networks from §8.1. 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 18a, 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. We see that for only one network (with 15 routers) running AED in parallel results in updates spread across 2 additional devices compared to using a single AED instance (Figure 18b). 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 3 and evaluate how quickly AED 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 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. To evaluate its speedup, we use the same 24 real data center networks. We observe that this optimization improves AED's performance by 1.2-1.5X.

9 Related Work

Network verification. Recent work [2–4, 8, 12, 13, 15, 20, 21, 25, 29] has shown how to detect errors in network control planes that lead to violations of important network-wide policies. Tools like Minesweeper [4], , Bagpipe [34] and FSR [33] 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) [1, 2, 26, 28, 30] and legacy networks [5, 10, 11, 27, 31]. Recently, many synthesis tools [5, 10, 11, 27, 31] 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 [35]. 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 [24], and makes no guarantees on the optimality or interpretability of the updates.

Data plane update. Some systems [18, 27] directly update the data plane. 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 data plane, resulting in further policy violations and needing frequent data plane updates.

10 Conclusion

We developed AED, a system for automatically updating distributed network configurations. It has rich configuration, policy coverage and supports a variety of configuration management objectives that operators care about. AED uses a novel MaxSMT-based formulation and domain-specific speed-up heuristics. Detailed evaluations over both real and synthetic network configurations show that AED computes updates fast. Also, AED-computed updates are comparable, if not better, than manually-authored ones.

References

[1] 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.

[2] 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

[3] 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

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

[5] 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.

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

[7] 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

[8] 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.

[9] 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.

[10] 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.

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

[12] 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)*.

[13] 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)*.

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

[15] Aaron Gember-Jacobson, Raajay Viswanathan, Aditya Akella, and Ratul Mahajan. 2016. Fast Control Plane

Analysis Using an Abstract Representation. In *SIGCOMM*.

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

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

[18] 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)*.

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

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

[21] 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)*.

[22] 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

[23] 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*.

[24] 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

[25] 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

[26] 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.

[27] 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

[28] 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.

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

[30] 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

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

[32] 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.

[33] 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.

[34] 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)*.

[35] 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)*.