# Automatic Repair of Network Configurations using a Modular Verifier and Symbolic Templates

Dexin Zhang[0009−0005−9009−9503], David Walker[0000−0003−3681−149X], and Aarti Gupta[0000−0001−6676−9400]

Princeton University, Princeton, NJ 08544, USA

**Abstract.** Verification of network control planes answers whether certain properties (such as reachability or fault tolerance) hold for the given network control configurations or not, giving formal guarantees when the verification succeeds. However, when verification fails, the task to *repair* the network configuration is often left to the user. Although networks in practice utilize a large set of configurations, if there is a small mistake, repair may require making changes in a relatively small fragment. The main challenge in such cases is to *localize* the error to a small fragment, and decide the *repair action* to fix the problem.

We present an algorithm to automatically repair the network configuration after an error is reported by CB-Ver, an existing modular network control plane verifier. Here, CB-Ver identifies the local component router that is incorrect, and provides feedback in the form of a counterexample to a verification condition that must be valid, but is violated. Our repair algorithm first constructs a *symbolic repair template* that captures a large set of syntactic mutations of the erroneous configuration. It then uses a Satisfiability Modulo Theories (SMT) solver to find a repair candidate that is *guaranteed to be correct*, using a counterexample-guided loop to avoid quantifier alternation during the synthesis procedure. Furthermore, thanks to optimizing SMT solvers, our algorithm can also *minimize* the cost of repair, with respect to a user-provided cost model. We have implemented a prototype and our experiments show that it can find correct repair actions in seconds for multiple errors injected in practical configurations of fattree networks (used in data centers) and for Internet2, a wide-area network with more than 100K lines of configuration.

## 1 Introduction

Networks use distributed routing protocols (e.g., the Border Gateway Protocol, BGP) and vendor-specific configurations on the routers to configure the protocols that make routing decisions in the network control plane. Errors in these network configurations can cause costly outages [40, 1]. This has motivated several research efforts in network control plane verification in the last decade or so [27, 39, 10, 25, 4, 31, 43, 11, 32, 38, 36, 6, 45]. Typically, network verifiers report whether a network is *correct or incorrect*, leaving the task of *repairing* the configuration to the user. However, networks in practice maintain a large set of configurations. For example, a wide-area network called Internet2 [2] contains

more than 100K lines of configurations. Thus, even if there is a small mistake in some router's configuration, and repairing it requires changes in a small fragment only, the problems of error localization and identifying a correct repair become very challenging.

To address these challenges, there have been some related research efforts [24, 26, 23]. They have proposed systematic frameworks to localize and repair errors. However, they have some limitations: they either take a *monolithic* approach over the entire network, which does not scale well on large networks, or they place restrictions on network features that they support. For example, CPR [23] represents the control plane in a graph-based abstract representation [25] that does not model local preferences in BGP, and CEL [24] targets only ACL errors.

In a different but related direction, there have been recent developments in *modular verification* [6, 36, 45] that provide a potentially useful platform for error localization and network repair in a scalable fashion. In modular verifiers, the network is decomposed into modules, such as individual routers (or some subsets of routers), each of which is annotated by users with specifications (e.g., invariants) and module-level correctness properties. Then, verification conditions (VCs) are generated per module and its interacting neighbor(s), based on user-specified annotations. The verifier, typically a Satisfiability Modulo Theories (SMT) solver [18], checks the validity of each verification condition independently (and possibly in parallel). Importantly, any errors reported by the verifier are thereby localized to a module and its interacting neighbor(s). Thus, their repair involves the configurations of these localized modules only. Furthermore, since the VCs are already localized and independent, they can be utilized to modularly verify the correctness of repair. To check whether the error is correctly fixed, only the affected VCs are re-checked — VCs in other locations that are unaffected and have already passed the check do not need to be re-run.

In this paper, we propose an algorithm to automatically repair network configurations on top of CB-Ver, an existing SMT-based modular verifier [45]. It solves a specific problem: given a (localized) error reported by the verifier, find syntactic mutations of the configuration that eliminate the error and re-validate the VCs required by CB-Ver. Although we use CB-Ver in this paper, our approach of configuration repair applies to a general SMT-based modular verifier with localized VCs. Examples of other such modular verifiers include Timepiece [6] and Lightyear [36]. However, these tools do not currently support any automated repair procedures.

Even with a localized error, the search space of possible mutations in repair is very large, which is challenging. To address this, we represent a large set of potential candidates as a *symbolic repair template*, where each assignment of the symbolic variables corresponds to a concrete candidate in the search space. This enables use of SMT solvers to search for correct candidates, within an iterative counterexample-guided approach for synthesis of the repair. Moreover, thanks to optimizing SMT solvers [15], this algorithm can also *minimize* the cost of repair, under a user-provided cost model.

We have implemented a prototype CB-Repair and our experiments show that it can find correct repair actions within 2 seconds for Internet2, a network of 10 routers with more than 100K lines of configuration in total, but where about 100 lines are involved in each error repair. For a synthetic fattree network injected with a small number (1-3) of different types of errors, CB-Repair successfully corrects all errors within 2 iterations by using a symbolic template with ∼170 symbolic variables. Our experiments on scalability of CB-Repair show that it can easily handle a greater number of (injected) faults (when the configuration size does not change), largely due to our symbolic template, with the repair time growing slowly. It can similarly scale well with growing size of a configuration for a practical policy of interest.

*Scope and Limitations.* Our approach depends on the modular verifier to generate VCs per-module and report the location of the failed errors. The verifier may itself require user-provided annotations and properties for generating the VCs – this holds for all existing modular verifiers [6, 36, 45]). Our approach does not restrict how VCs are generated, as long as they are sound and can be solved by an optimizing SMT solver [15].

Particularly with respect to CB-Ver, CB-Repair can be viewed as a companion repair engine. It inherits its assumption of user-provided interfaces and properties and modular VC generation. Unlike CB-Ver which can synthesize a CB-graph, CB-Repair assumes that a connected CB-graph is available (either via CB-Ver or user-provided).

CB-Repair has the new capability for automatic localized repair of an erroneous network configuration that results in a failed VC in CB-Ver. The new technical contributions (on top of CB-Ver) include the symbolic template for repair candidates, the use of an optimizing SMT solver, and the counterexample-guided search loop for synthesis of a repair. We also note that other modular verifiers [6, 36] do not support automated repair.

Our approach is guaranteed to not produce any *incorrect* repair that violates a user-specified property. However, the repair found by our approach may not suffice to prove correctness when the annotations and/or properties are not accurate enough; a correct repair may not exist when the annotations are too strong or too weak. In these cases, the user may also need to refine the annotations and/or the properties. A current limitation is that our approach does not consider refinement of user-provided annotations – this is a topic for future work.

*Artifact availability.* An artifact including our tool and our evaluations is published at https://doi.org/10.5281/zenodo.22691256 [44]. It includes scripts to reproduce all experiments we have made in this paper.

*AI disclosure.* AI was used to proofread the paper, correct grammatical errors, and improve formatting. AI was also used in developing the artifact.

## 2 Background and Overview

In this section, we give a brief overview of our configuration repair approach, including relevant background on network control plane verification and CB-Ver, along with a motivating example.

### 2.1 Background: control plane verification and CB-Ver

*The network control plane.* In a network *control plane*, *routers* maintain states (routes to a destination), and exchange messages with their neighbors to announce their routes according to the configured protocols and policies. These message exchanges lead to state updates, where routers select the best routes available to them from their neighboring routers, until (hopefully) all routers converge to some stable state. The converged states in the network control plane determine how the routers deliver network packets in the *data plane*.

In terms of a formal model of the network control plane: We use $S$ to denote the set of routes and $M$ to denote the set of messages. They are represented as records of routing fields that depend on the *routing protocol* (e.g., BGP, OSPF, RIP). For simplicity, we assume that $M$ comprises the same routing fields as $S$. The messages a router can send are based on the route it currently selects and the *export policy* it is configured with. We write $\text{export}_{u,v} : S \to M$ to denote the export policy of router $u$ exporting routes to its neighbor $v$, and mapping a route to a message. Similarly, when a router $v$ receives a message, it applies an *import policy* $\text{import}_{u,v} : M \to S$ on the message received from $u$. Upon receiving a message, a router updates its state using a *merge operator* $\text{merge} : S \times S \to S$ that selects the preferred route between two routes. The composition of an export policy and an import policy is called the *transfer function* $\text{transfer}_{u,v}(s) = \text{import}_{u,v}(\text{export}_{u,v}(s))$, which is associated with the transfer of a message from $u$ to $v$.

*SMT encoding of control plane model.* Modeling routing protocols using SMT theories is a well-understood process; the reader can refer to Minesweeper [10] or CB-Ver [45] for a more detailed explanation.

For exposition purposes, we focus mainly on a model of the Border Gateway Protocol (BGP) [29] – other protocols can be modeled using a general stable routing path (SRP) model, similar to prior work [10, 11]. A BGP route $s \in S$ is encoded as a quadruple: $(\text{prefix}, \text{lp}, \text{path}, \text{comm})$, where prefix is a 32-bit vector (assuming IPv4) indicating the destination prefix, lp is an integer indicating local preference, path is an ordered sequence of AS numbers for each router this message has traversed (called AS-path in BGP), and comm is a finite set of community tags. We use $\text{prefix}(s)$ to denote the prefix field of a route $s$. Moreover, we use a special value $\infty \in S$ to represent "no route", a placeholder when no route has been chosen by the router [45]; $\infty$ is the least preferred route.

In BGP, the merge function encodes the preference order between two routes, checking whether one of the routes is $\infty$ first, then the local preference (higher

is preferred), and if it is the same, then the length of the AS-path (shorter path is preferred). This can be encoded in SMT theories as:

$$\text{merge}(s_1, s_2) = ite(s_1 = \infty, s_2, ite(s_2 = \infty, s_1, ite(\text{lp}(s_1) > \text{lp}(s_2), s_1, \cdots)))$$

where $ite(p, a, b)$ represents an if-then-else expression that evaluates to $a$ when $p$ is true and to $b$ otherwise.

The export/import policy and transfer functions are modeled based on the configuration language (supported by a vendor) and its semantics. For example, in Cisco's configuration language, a statement `set community 100:2` sets the comm field to a singleton set {100:2}. This can be modeled as a function $f : S \to S$ as

$$f(s) = (\text{prefix}(s), \text{lp}(s), \text{path}(s), \{100{:}2\}).$$

In real configurations, an export/import policy is a combination of such statements; we will present a core language for these configurations in §3.1.

*SMT-based verification of network control planes.* In SMT-based verification of control planes [10, 6, 36, 45], a user can express correctness properties as SMT formulas over the SMT-based network model. Examples of such properties include reachability to certain/all destinations of interest, fault tolerance, isolation, etc. In a monolithic approach, the verifier encodes the entire network model and the specified property (its negation) as an SMT formula, and uses an SMT solver to find a bug or a proof. Due to a single large formula for the entire network, monolithic verification often does not scale to large sized networks; moreover, if verification fails, it is very challenging to localize the error to a small fragment in the network.

In a *modular* verifier [6, 36, 45], the overall verification task is decomposed into multiple subtasks over smaller fragments in a network, e.g., a router or a link between two routers. Typically, a modular verifier requires a user to provide annotations (e.g., invariants) — Lightyear [36] requires invariants and path constraints, Timepiece [6] requires temporal invariants (with an explicit representation over all time instants), and CB-Ver [45] requires invariants and abstract convergence sets (denoted $\mathcal{I}, \mathcal{Q}$ respectively). Given the user-provided annotations and correctness properties, a verifier generates a set $\phi$ of verification conditions (VCs)

$$\phi = \phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_N,$$

where each $\phi_i$ depends on only a small fragment of the network. If any $\phi_i$ is not valid, then verification fails, and the verifier reports the error. It usually provides feedback to the user about which $\phi_i$ fails (providing *localization* of the error) and why $\phi_i$ is not valid (via a counterexample generated by the SMT solver).

*Modular verification and error reporting in CB-Ver.* CB-Ver [45] requires a user to provide an invariant annotation $I$ and an abstract convergence annotation $Q$ for each router in the network, along with a node-local correctness property $Y$. It automatically generates VCs for routers and links in the network, and checks their validity independently and in parallel. The complete set of VCs is shown in

| VC Set | VC Formula |
|---|---|
| $VC_{\text{Init}}(v)$ | $\text{init}(v) \in \mathcal{I}(v)$ |
| $VC_{\text{Inv}}((u,v))$ | $\forall s_u, s_v.\ s_u \in \mathcal{I}(u) \wedge s_v \in \mathcal{I}(v) \Rightarrow s_v \oplus f_{(u,v)}(s_u) \in \mathcal{I}(v)$ |
| $VC_{\text{Prop}}(v)$ | $\forall s_v.\ s_v \in \mathcal{Q}(v) \Rightarrow s_v \in \mathcal{Y}(v)$ |
| $VC_{\text{CBroot}}(v)$ | $\text{init}(v) \in \mathcal{Q}(v) \wedge \bigwedge_{u \in V,(u,v) \in E} \begin{pmatrix} \forall s_u, s_v.\ s_u \in \mathcal{I}(u) \wedge s_v \in \mathcal{Q}(v) \\ \Rightarrow s_v \oplus f_{(u,v)}(s_u) \in \mathcal{Q}(v) \end{pmatrix}$ |
| $VC_{\text{CBedge}}((u,v))$ | $\forall s_u, s_v.\ s_u \in \mathcal{Q}(u) \wedge s_v \in \mathcal{I}(v) \Rightarrow s_v \oplus f_{(u,v)}(s_u) \in \mathcal{Q}(v)$ |

Fig. 1: CB-VER: Verification Condition (VC) Formulas.

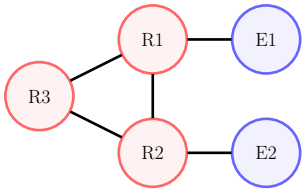
Figure 1. Here, $VC_{\text{Init}}, VC_{\text{Prop}}$ and the first conjunct of $VC_{\text{CBroot}}$ are checked per node ($v \in V$ of the network graph $G = (V, E)$), while $VC_{\text{Inv}}, VC_{\text{CBedge}}$ and the second conjunct of $VC_{\text{CBroot}}$ are checked per edge ($(u, v) \in E$). To understand our repair algorithm, it is not necessary to understand the exact formulations of these VCs. Briefly, $VC_{\text{Init}}, VC_{\text{Inv}}$ together check the correctness of the user-provided $\mathcal{I}$, $VC_{\text{Prop}}$ checks that the abstract convergence annotation $\mathcal{Q}$ implies $\mathcal{Y}$, and $VC_{\text{CBroot}}, VC_{\text{CBedge}}$ together identify a maximal CB-graph, which provides a structure for well-founded inductive reasoning about $\mathcal{Q}$. CB-VER reports an error if any one of $VC_{\text{Init}}, VC_{\text{Inv}}, VC_{\text{Prop}}$ is not valid, or if it cannot find a connected CB-graph with the nodes and links identified by $VC_{\text{CBroot}}$ and $VC_{\text{CBedge}}$, respectively (readers may refer to the related paper [45] for more details). It has been shown that the CB-VER verification algorithm is sound with respect to a general asynchronous network semantics [45].

In this paper, we focus on the violations of $VC_{\text{Inv}}$ and $VC_{\text{CBedge}}$ in CB-VER, which are the most common types of VC violations in practical networks. Importantly, each of these violations is localized to a specific link $(u, v)$ in the network, with a potentially erroneous transfer function $f_{(u,v)} : S \to S$ due to an erroneous configuration in node $u$, or $v$, or both. We aim to automatically repair such erroneous configurations. We also note that reparing the local transfer function $f_{(u,v)}$ of one link does not affect VCs in other links. Thus, there is no need to re-check the other VCs. Furthermore, because the repair of the transfer function of one link has no effect on other links, our repair is guaranteed not to introduce new errors.

### 2.2 Motivating example with an erroneous configuration

We consider a small network example[1] shown in Fig. 2a. The network consists of 3 routers, R1, R2, and R3, connected with 2 external neighbors, E1 and E2. Suppose the intended property is that any route originating from E1 should not

[1] This example is inspired by a motivating example in Lightyear [36].




(a) Example network.

```
ip prefix-list some_list permit 10.0.0.0/8

route-map IMPORT_R3 permit 10
match ip address prefix-list some_list
set community 100:3
...
```

(b) Config. of R2 for importing routes from R3.

Fig. 2: Example network and its erroneous configuration.

be sent to E2 (a blocking property); to ensure this, the network operator uses a community 100:1 that is added to a route, when imported from E1, and any route with this community tag will be dropped when exported to E2. Moreover, the community 100:1 should not be dropped within R1, R2, and R3.

To verify this property using CB-Ver, a user would provide the invariants for each router; the invariant for R1, R2, and R3 is the same: any route from E1 is tagged with community 100:1. The verifier will generate VCs based on these invariants and the property. For instance, the following VC checks that the link $R3 \rightarrow R2$ does not drop the community tag 100:1:

$$\begin{aligned} \forall s, s'.\ & s \neq \infty \wedge 100{:}1 \in \mathrm{comm}(s) \wedge s' = \mathrm{import}_{R3,R2}(\mathrm{export}_{R3,R2}(s)) \\ & \Rightarrow s' \neq \infty \wedge 100{:}1 \in \mathrm{comm}(s') \end{aligned} \tag{1}$$

(here $s = \infty$ indicates that the route $s$ is dropped).

If every VC check passes, the soundness of CB-Ver ensures that the invariants hold at each router. However, suppose the operator made a mistake in configuring the import policy $\mathrm{import}_{R3,R2}$, as shown in the snippet in Fig. 2b. Here the mistake is in the statement that says `set community 100:3`, instead of `set community 100:3 additive`. In Cisco's configuration format, `additive` adds a community to the existing communities, while a plain `set community` would nullify all existing communities before setting the new one. This means that when a route from E1 is tagged by 100:1, this community would be dropped when R2 imports a route from R3, thereby leading to a *VC violation*.

Our repair algorithm is based on a simplified representation of the original configuration in a core language, as shown for the given configuration in Fig. 3a. It first matches whether the route has destination prefix 10.0.0.0/8. If the route is matched, this import policy will set its community to 100:3; otherwise, it will leave the route unchanged and accept it as-is. (The core language specification is described in §3.1.)

The goal of configuration repair is to change the original configuration to a correct one. For this example, the goal is to find a correct $\mathrm{import}'_{R3,R2}$ such that the VC shown in (1) is valid, with $\mathrm{import}_{R3,R2}$ replaced by $\mathrm{import}'_{R3,R2}$.

CB-Repair considers different syntactic mutations of the given configuration, where each mutation provides a *candidate* for repair. For example, the mutation in Fig. 3b replaces **SetComm** with **AddComm**; Fig. 3c replaces the literal in **MatchPrefix** with some other one, while Fig. 3d replaces the final action from **permit** to **deny**. Note that among these candidates, only Fig. 3b

**MatchPrefix**(10.0.0.0/8);
**SetComm**(100:3);
**permit**;

(a) Simplified representation of Fig. 2b.

**MatchPrefix**(10.0.0.0/8);
**AddComm**(100:3);
**permit**;

(b) Candidate 1

**MatchPrefix**(⟨some literal⟩);
**SetComm**(100:3);
**permit**;

(c) Candidate 2

**MatchPrefix**(10.0.0.0/8);
**SetComm**(100:3);
**deny**;

(d) Candidate 3

**MatchPrefix**(
  **if** $\text{activate}_1$ **then** $\text{prefix}_1$
  **else** 10.0.0.0/8);
**if** $\text{activate}_2$ **then**
  **AddComm**(100:3);
**else** **SetComm**(100:3);
**if** $\text{activate}_3$ **then**
  **deny**;
**else** **permit**;

(e) Symbolic template

Fig. 3: Simplified representation, repair candidates, and symbolic template.

is a correct mutation, since Fig. 3c would still drop the community 100:1, while Fig. 3d would drop the entire route.

A naive way for performing repair would be to try the candidates in Fig. 3b, 3c, and 3d one by one, and verify whether the new VC is valid. However, this can be inefficient if there are multiple possible changes in various lines, and there are multiple combinations of such cases to consider. Therefore, we propose using a *symbolic template* in CB-Repair. It combines *all candidate changes* together and introduces symbolic parameters, where instantiations of these parameters correspond to different changes and their combinations. For our example, the symbolic template is shown in Fig. 3e. It contains four parameters: $\text{activate}_1$, $\text{activate}_2$, $\text{activate}_3$ and $\text{prefix}_1$. When $\text{activate}_1 = \text{activate}_3 = \text{False}$ and $\text{activate}_2 = \text{True}$, the template is instantiated as in Fig. 3b.

We denote the vector of four parameters in Fig. 3e as $\pi$. The problem of configuration repair is now formulated as finding an instantiation of $\pi$, such that the VC with $\pi$, shown below, is valid.

$$\exists \pi.\ \forall s, s'.\ s \neq \infty \wedge 100{:}1 \in \text{comm}(s) \wedge s' = \text{import}'_{R3,R2}(\text{export}_{R3,R2}(s), \pi) \Rightarrow s' \neq \infty \wedge 100{:}1 \in \text{comm}(s')$$

However, a new challenge emerges due to difficulties in SMT solvers for handling above formulas of the form $\exists \pi.\ \forall s, s'.\ \cdots$, which contain a quantifier alternation. To address this challenge, CB-Repair uses a counterexample-guided approach, where $\forall s, s'$ is replaced by considering a finite set of counterexamples $(s_1, s'_1), \cdots, (s_n, s'_n)$. Once a candidate is found that makes the VC valid on the finite set of counterexamples, then CB-Repair performs an additional step of

checking whether the candidate also works for the VC with $\forall s, s'$. If it does, then the iterative loop terminates with a correct repair, otherwise we add to our set of counterexamples and try again. Our experiments show that in practice a correct repair can be found in a few iterations via a small number of counterexamples.

Finally, we note that this approach also admits *minimization of the cost of repair*, by using a cost model and an optimizing SMT solver [15]. We assume that the cost of a candidate is a linear combination of the costs of each change, and express it as a cost function over the symbolic parameters in the template, to be minimized when searching for a candidate repair (details in §4).

## 3 Route Maps and Symbolic Repair Templates

Real world network configurations are usually described in some domain specific language (DSL), such as Cisco's configuration format [16], Juniper's configuration format [30], etc. In this paper, we consider a *core language of route maps* suitable as a proof-of-concept for automated repair. It is formal, targets the BGP protocol, and provides a useful subset of both Cisco and Juniper configuration formats [16, 30]. We start by describing this core language, and how it is extended to design a symbolic repair template. We also present a procedure to automatically generate a symbolic template by traversing the syntax tree and considering different kinds of mutations.

### 3.1 Language of route maps

The syntax of our language is shown (as unboxed text) in Fig. 4. The topmost construct is a route map, which consists of a list of clauses; each clause consists of several matchers and modifiers, and an action (permit/deny). The matchers and modifiers operate on the BGP attributes of routes, including BGP communities, IP prefixes, or local preferences.

A route map $rm$ can be interpreted as a function $\text{apply}(rm) : S \to S$ with the following semantics: The input route $s$ is checked against each successive clause for a match (with all matchers). When a match is found, the route map applies the clause's modifiers and action. For example, a route map with a single **permit** action does not modify the routes, so the transfer function for this route map is the identity function $f(s) = s$. A route map with a single **deny** action will drop any route, which corresponds to the transfer function that maps any route to $\infty$ (a special route used as "no route"). The full semantics of our language of route maps can be found in Appendix A.

### 3.2 Symbolic repair templates

As described in §2, a VC violation involves an edge $(u, v)$ with export route map $e$ and import route map $i$ that together violate the validity condition. The goal of repair is to find two correct route maps $e^*$ and $i^*$ to replace $e$ and $i$, such that the repaired transfer function $f^*_{(u,v)}$ satisfies all the generated VCs. Rather than

$$
\begin{aligned}
rm \in \text{RouteMap} &:= (cl;)^* \\
cl \in \text{Clause} &:= (mt;)^*(md;)^* a \mid \boxed{\textbf{if } b \textbf{ then } cl} \\
mt \in \text{Matcher} &:= \textbf{MatchPrefix}(l^*) \mid \textbf{MatchComm}(l^*) \mid \boxed{\textbf{if } b \textbf{ then } mt} \\
md \in \text{Modifier} &:= \textbf{AddComm}(l^*) \mid \textbf{SetComm}(l^*) \mid \textbf{RemoveComm}(l^*) \\
&\quad \mid \textbf{SetLP}(l) \mid \boxed{\textbf{if } b \textbf{ then } md} \mid \boxed{\textbf{if } b \textbf{ then } md \textbf{ else } md} \\
a \in \text{Action} &:= \textbf{deny} \mid \textbf{permit} \mid \boxed{\textbf{if } b \textbf{ then permit else deny}} \\
&\quad \mid \boxed{\textbf{if } b \textbf{ then deny else permit}} \\
l \in \text{Literal} &:= \text{PrefixLiteral} \mid \text{CommunityLiteral} \mid \text{LPLiteral} \\
&\quad \mid \boxed{z \mid \textbf{if } b \textbf{ then } z \textbf{ else } l} \\
&\quad \boxed{b \in \text{BoolSym},\ z \in \text{LiteralSym}}
\end{aligned}
$$

Fig. 4: The language of route maps (unboxed) and repair templates (boxed and blue); $*$ denotes a list.

directly explore all possible $e^*$ and $i^*$ for repair (which would be expensive and not scalable), we limit the search space to some mutations of $e, i$, considering the following types of misconfiguration errors likely to happen in the real world:

1. typos in literals, including community tag, IP prefix lists, or local preferences;
2. using the wrong expressions, e.g., **permit** instead of **deny**, or **SetComm** instead of **AddComm**; and
3. missing matchers, modifiers and clauses in a route map.

In our evaluations (§5.2), we consider varying numbers and types of errors from the above list.

For finding a repair that addresses the above errors, with flexibility for fixing other errors, we use a *template route map*, denoted $R$. Templates are route maps that are *parameterized* with symbolic variables. By instantiating $R$'s symbolic variables with exact values, we can obtain a concrete route map $r$.[2]

The syntax of template route maps (boxed and blue, Fig. 4) extends the syntax of route maps. First, templates may contain symbolic boolean variables $b$ and symbolic literal variables $z$ (matching the appropriate type). Second, templates may wrap clauses, matchers, and modifiers in an **if** $b$ **then** conditional statement, for conditionally inserting (when $b$ is true) or deleting (when $b$ is false) these elements in a template. Finally, templates can change the modifier and action with a symbolic condition.

*Symbolic Template Generation.* We use an automated procedure that takes as input an incorrect route map $r$ and returns a template route map $R$ as output. The procedure traverses the syntax tree of $r$, and performs the following changes:

[2] We use lowercase $r, e, i$ for route maps and uppercase $R, E, I$ for templates.

- *modifies* all existing constructs (clauses, matchers and modifiers) by replacing them with conditionally-activated ones (examples shown in Fig. 3e);
- conditionally *inserts* new conditional constructs, up to some user-specified bounds; and
- conditionally *removes* existing constructs.

Each new conditional construct introduces fresh boolean and literal symbolic variables; the number of variables and the range of literal values determine the complexity of the search space for repair. In practice, we expect the user to specify a small set of choices for the literals (extracted from the configurations). The numbers and sizes of conditionally inserted new constructs can be optionally controlled by the user; the default values in our tool are shown in Table 1. More constructs are added with higher values, which allows a larger search space for repair candidates, but may potentially worsen performance.

| **Generation Option** | **Default Value** |
|---|---|
| # new clauses to add | 2 |
| # matchers/modifiers in new clauses | 2 of each type |
| # new matchers/modifiers to add | 2 of each type |

Table 1: Default values of user-settable options in template generation.

*Benefits of symbolic templates.*

- By *allowing modification, deletion, or insertion* of every construct in $r$, a template route map provides a rich space of corrective repairs (within some user-defined bounds) without needing to guess the kind of errors, the exact number of errors, or which combinations of repair actions to try.
- Symbolic variables enable use of an SMT solver, and instantiating the template with the solution yields a corrected route map.
- Repair *cost* can be minimized using a cost function $c$ over symbolic variables, with different costs associated with different repair actions. For example, we can replace a matcher $m$ with **if** $b$ **then** $m$, using cost function $c(b) = ite(b, 0, 1)$. This assigns cost 0 for an unmodified candidate and cost 1 for removing $m$ (we expect lower cost for a smaller repair). We write $c(R)$ for the cost value of a template route map $R$.

## 4 Repair Algorithm with Symbolic Templates

The template generation procedure (§3) takes the configuration $e, i$ as input and returns the templates $E, I$. We use $\pi$ to represent the list of symbolic variables (and write $E[\pi], I[\pi]$ to indicate that $E$ and $I$ include variables in $\pi$). Following the semantics of route maps, we interpret the templates $E$ and $I$ together as a symbolic transfer function parameterized by $\pi$:

$$\begin{aligned} \text{export}(\pi, s) &= \text{apply}(E[\pi])(s) \\ \text{import}(\pi, s) &= \text{apply}(I[\pi])(s) \\ F_{(u,v)}(\pi, s) &= \text{import}(\pi, \text{export}(\pi, s)) \end{aligned}$$

Recall that in Fig. 1, the VCs $VC_{\text{Inv}}$, $VC_{\text{CBedge}}$ (and part of $VC_{\text{CBroot}}$) have the form

$$\forall s_u, s_v. P(s_u, s_v, f_{(u,v)}(s_u))$$

which depends on the transfer function $f_{(u,v)}(s) = \text{import}_{u,v}(\text{export}_{u,v}(s))$ (and where we have replaced the concrete formulas with some predicate $P$). If such a VC fails, the counterexample generated by the SMT solver includes an assignment of $s_u$ and $s_v$ such that the formula is false.

The goal of the repair algorithm is to find an assignment of $\pi$ such that the VC is valid when $f_{(u,v)}(s)$ in the VC is replaced by the symbolic transfer function $F_{(u,v)}(\pi, s)$. (Note that the VC used for repair includes all verification conditions that involve the local configuration, i.e., it is a conjunction of $VC_{\text{Inv}}, VC_{\text{CBedge}}$ and part of $VC_{\text{CBroot}}$.)

The repair problem can now be formulated as solving:

$$\exists \pi. \forall s_u, s_v. P(s_u, s_v, F(\pi, s_u)),$$

where $\pi$ is existentially quantified, and we write $F_{(u,v)}(\pi, s)$ more simply as $F(\pi, s)$. Note that this formula involves a quantifier alternation (similar to synthesis in general). However, quantifier alternation is often difficult for SMT solvers [22]. Therefore, to avoid this quantifier alternation, we use an iterative counterexample guided approach, in the style of counterexample guided abstraction refinement (CEGAR) [17] and synthesis (CEGIS) [8].

In the loop for counterexample guided repair, we start from the counterexample $cex_0 = (s_{u0}, s_{v0})$ generated by CB-Ver. In each iteration, we use an SMT solver to find a candidate $\pi^*$ such that $\bigwedge_{i=0}^{n} P(s_{ui}, s_{vi}, F(\pi^*, s_{ui}))$ is true, where each $cex_i = (s_{ui}, s_{vi})$ is collected from previous iterations (starting from $cex_0$). If we cannot find such a $\pi^*$, then we report failure; otherwise we check whether $\pi^*$ is a *correct* candidate by checking the validity of $\forall s_u, s_v. P(s_u, s_v, F(\pi^*, s_u))$. If it is valid, we report a correct repair as $E, I$ instantiated with $\pi^*$; otherwise, we get a new counterexample $cex_{n+1}$ and repeat the loop.

Furthermore, we can minimize the cost of repair by using an optimizing SMT solver [15] in the search for $\pi*$. Here, the cost function $c$ is also parameterized by $\pi$, and expressed in terms of costs of $E$ and $I$: $c(\pi) = c_E(\pi) + c_I(\pi)$. Now, in each step of the iteration, we search for a $\pi$ that also minimizes the cost:

$$\min_\pi\ c(\pi).\ \text{s.t.}\ \bigwedge_{i=0}^{n} P(s_{ui}, s_{vi}, F(\pi, s_{ui})).$$

Our algorithm is shown as pseudo-code in Algorithm 1. The main iterative loop (lines 2–15) tries to find a fix with a finite set of concrete counterexamples $CEX$, and the user can specify a maximum number of iterations $k$ to try (default $k = 100$). In each iteration, we first replace the $\forall s_u, s_v$ in $VC$ with counterexamples in $CEX$, obtaining a *repair condition* $\text{RC}(s_u, s_v, F(\pi, s_u))$ for each counterexample $(s_u, s_v) \in CEX$. Then we use an optimizing SMT solver (line 4) to find a fix (i.e., a satisfying solution of $\pi$ with minimum cost $c$) that repairs all the counterexamples. If there is no such fix, we declare failure (line 6). Otherwise, we check whether the fix might work in general for all other values of

**Input :** Network $N$ and annotations $(\mathcal{I}, \mathcal{Q})$, failing edge $(u, v)$, initial counterexample $cex_0$, repair templates $E, I$ with transfer function $F(\pi, s)$ and cost function $c(\pi)$, maximum number of iterations $k$
**Output :** Success with route maps $e^*, i^*$, or Fail

```
1  CEX ← {cex_0};
2  for i ← 1 to k do
3  |  φ ← ⋀_{(s_u,s_v)∈CEX} RC(s_u, s_v, F(π, s_u));
4  |  (res_1, π*) ← OPT(φ, c);
5  |  if res_1 = unsat then
6  |  |  return Fail;
7  |  else
8  |  |  φ' ← VC(s_u, s_v, F(π*, s_u));
9  |  |  (res_2, cex) ← SAT(¬φ');
10 |  |  if res_2 = sat then
11 |  |  |  CEX ← CEX ∪ {cex};
12 |  |  else
13 |  |  |  e* ← instantiate(E, π*);
14 |  |  |  i* ← instantiate(I, π*);
15 |  |  |  return Success(e*, i*);
16 return Fail;
```

**Algorithm 1:** Repair Algorithm.

$s_u$ and $s_v$ via a standard VC check (line 8). If the VC is valid, we have found a correct and minimal repair $e^*, i^*$ (lines 13–15). Otherwise, we get another counterexample that we add into the set $CEX$ (line 11) and generate a stronger repair condition $RC$ in the next iteration. The repair algorithm is guaranteed to find the correct solutions $e^*, i^*$ if they exist and the generation options and $k$ are large enough. Note also that the finite range of values of $\pi$ (due to symbolic literals having a finite set of choices) guarantees termination of Algorithm 1.

Our repair algorithm can be viewed as an instance of Syntax-Guided Synthesis (SyGuS) [7], where the grammar is defined by our symbolic template, and a counterexample-guided search is similar to CEGIS [8]. Similar to many SyGuS solvers, we too use an SMT solver to find a satisfying instance. However, the widely used SyGuS solver CVC5 [9] does not directly support optimization under user-defined repair costs, so we developed our own synthesis procedure for repair.

## 5 Implementation and Evaluation

We have implemented CB-Repair as an extension of the CB-Ver tool [45]. We use Batfish [21] to parse network configurations in Cisco's and Juniper's format and convert them into our core language (Fig. 4) as an intermediate representation (IR), and then perform verification using CB-Ver (along with user-provided annotations). When verification fails at some edge $(u, v)$, CB-Repair uses the corresponding IR of the import and export policy of that edge to generate repair templates and to perform repair. The user can control the

search space of template generation (Table 1) and the maximum number of iterations as command line options.

*Evaluation Goals.* We would like to answer the following questions: 1. Is CB-Repair able to repair different kinds of configuration errors, including errors in a real-world network? 2. What is the performance of repair?

### 5.1 Benchmarks and setup

We experimented with two sets of benchmarks: (1) synthetic fattree networks (parameterized by size), and (2) Internet2, a real wide-area network with over 100,000 lines of Juniper configurations. (These benchmarks are the same as in [45]; readers can check more details there.)

*Synthetic fattrees.* These benchmarks consider the following two properties[3]:

- `ValleyFree`: selected route between nodes (all pairs) must not be a valley (up-down-up);
- `Hijack`: certain routes from an external "hijacker" node must be blocked from the internal network. We also evaluate `Hijack` conjoined with a reachability property, where routes to a certain whitelist prefix must not be blocked.

*Internet2.* The Internet2 benchmark tests CB-Repair on a real-world wide-area network [2]. We consider a `BlockToExternal` property (first presented in Bagpipe [39]) to check that internal routes are not advertised to Internet2's peers. We use a version of configurations [39] that have been reported to have violations of this property at some locations. (Some features used by Internet2, e.g., IPv6, next-hop self are not modeled, which do not affect our evaluation.)

All experiments were run on a Linux server with 16 Intel(R) Xeon(R) CPUs and 252GB memory. The verification procedure utilizes all 16 cores in the modular setting to verify the VCs independently. The repair procedure runs on a single thread sequentially and does not utilize the multi-core parallelization.

### 5.2 Evaluation results for repair

*Options in template generation and repair costs.* In our evaluations, we allowed 2 insertions for each of clause/matcher/modifier. In addition, we specified a small set of 2-4 literals (extracted from the configurations or manually specified) for choices of a symbolic literal. For costs of repair, we assigned a cost of 1 for modifying literals, 3 for toggling action, and 5 for insertion and deletion of a modifier/match/clause. (The total cost of inserting a clause is the sum of the clause insertion cost and the cost of inserted modifiers/matchers in it.)

[3] We did not evaluate repair for `Reachability` and `PathLength` properties in [45] because their policies are too simple to be injected with different fault types.

*Fault injection in fattree networks.* To mimic misconfiguration errors in real networks, we experimented with mutations in the synthetic fattree benchmarks. We considered the following four types of mutations:

- `ToggleAction`: toggles permit/deny actions.
- `ChangeLiteral`: changes literals to a random value.
- `Deletions`: removes matchers, modifiers, or clauses.
- `Insertions`: adds randomly-generated matchers, modifiers, or clauses.

CB-Repair's repair algorithm (Algorithm 1) repairs specific edges reported by the modular verifier CB-Ver. For our experiments, we randomly pick an edge in a 20-node network[4] and inject faults only on that edge. For a given edge, we inject faults: (1) of different types, (2) varying in number, and (3) on random locations in the configurations. Since faults on random locations may not lead to an error, we report results only for faults that result in errors. We report the average values over 20 runs for each fault (after filtering out the no-error cases).

*Baseline and comparison.* Existing network-configuration repair systems include Marham [26] and CPR [23] (discussed in more detail later, §6), but they solve very different repair problems. Marham targets SDN forwarding behavior encoded as Horn clauses, while CPR performs whole-network repair over an abstract control-plane representation. Neither tool accepts localized configuration errors or verifiable modular interfaces, as required in our setting. Applying either system would require substantial effort in developing new encodings and extensions, rather than a direct execution on our benchmarks. We therefore do not use these systems as evaluation baselines.

*Evaluation Results on fattree networks.* We evaluated repair for `ValleyFree` and `Hijack` properties. The results are shown in Table 2. Here, we injected 1-3 faults of various types and in combination (Injected Faults), on a single edge in a 20-node fattree network. The second column shows the verification time (in milliseconds, ms) taken by CB-Ver to find a VC violation and report a buggy edge, after which we use CB-Repair to perform repair. The number of symbolic parameters in the template route map is ∼170, which provides a fairly large search space. Note that our template route map is unaware of the type or number of injected faults, and all mutations are repaired successfully in less than a second (Repair time, in milliseconds, ms), demonstrating the effectiveness of our template route map. Furthermore, our counterexample-guided approach in Algorithm 1 finds a solution successfully within 2 iterations (#Repair Iterations) on average, while avoiding quantifiers. In all case where an error is injected, a repair is found, and therefore the success rate is 100%. The last column lists the most frequent repair solution found by our procedure over 20 runs, which matches the expected corrective action based on the fault(s) injected.

We performed additional experiments to evaluate the scalability of repair on a parameterized benchmark for the `Hijack` property. In `Hijack`, routes from the

[4] For larger networks with 2000 nodes, only the verification time by CB-Ver would grow, the repair time per buggy edge would not change much due to localized repair.

| Injected Faults | CB-Ver Verif. Time (ms)* | #Symbolics in Template | Repair Time (ms) | #Repair Iterations** | Repair Action (most frequent) |
|---|---|---|---|---|---|
| **ValleyFree** | | | | | |
| `ChangeLiteral` × 1 | 661.7 | 186.5 | 166.9 | 1.00 | `ChangeLiteral` × 1 |
| `ToggleAction` × 1 | 683.2 | 166.5 | 167.6 | 1.00 | `ToggleAction` × 1 |
| `ToggleAction` × 1, `ChangeLiteral` × 1 | 676.3 | 165.2 | 188.5 | 1.20 | `ToggleAction` × 1 |
| `Remove` × 1 | 669.7 | 151.6 | 163.1 | 1.00 | `Add` × 1 |
| `Add` × 1 | 666.8 | 175.9 | 173.2 | 1.15 | `Remove` × 1 |
| `Remove` × 1,`Add` × 1 | 668.5 | 161.4 | 166.7 | 1.00 | `Add` × 1 |
| `ToggleAction` × 1, `Remove` × 1, `Add` × 1 | 665.5 | 165.1 | 161.2 | 1.00 | `Add` × 1 |
| **Hijack** | | | | | |
| `ChangeLiteral` × 1 | 337.4 | 199.7 | 130.0 | 1.50 | `ChangeLiteral` × 2*** |
| `ToggleAction` × 1 | 336.9 | 171.2 | 90.5 | 1.00 | `ToggleAction` × 1 |
| `ToggleAction` × 1, `ChangeLiteral` × 1 | 338.9 | 200.2 | 134.8 | 1.55 | `ToggleAction` × 1 |
| `Remove` × 1 | 339.7 | 144.2 | 94.2 | 1.10 | `Add` × 1 |
| `Add` × 1 | 344.4 | 171.8 | 83.7 | 1.00 | `Remove` × 1 |
| `Remove` × 1,`Add` × 1 | 333.4 | 151.0 | 93.8 | 1.05 | `Add` × 1 |
| `ToggleAction` × 1, `Remove` × 1, `Add` × 1 | 339.4 | 153.1 | 89.2 | 1.00 | `Add` × 1 |

* Reports wall clock time for verifying all VCs for 20 nodes in parallel.
*** All repairs succeed after the iterations; the repair success rate is 100%.
*** We count remove and add literal as two `ChangeLiteral` actions.

Table 2: Repair for different types/numbers of faults (averaged on 20 runs).

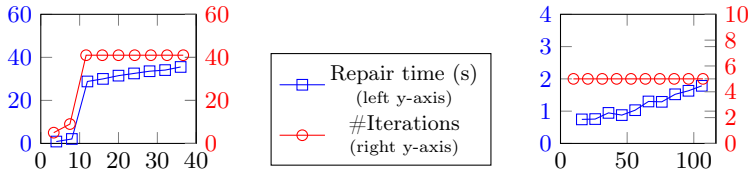
external network with an internal destination prefix are dropped. We replaced this blocked prefix with a prefix list of variable length, and injected a variable number of `ChangeLiteral` faults in the list. The results are shown in Fig. 5 with two graphs. In each graph, we report the repair time (in seconds, shown on the left $y$-axis) and the number of iterations (shown on the right $y$-axis). The left graph shows how these statistics change for a fixed configuration with an increasing number of faults, and the right graph shows how these change for a fixed number of faults with an increasing number of configuration lines.[5]

In the left graph, the number of iterations grows from 5 to 9, where CB-Repair fixes the faulty literals in 4-8 faults. For 12-36 faults, CB-Repair prefers to remove the whole prefix matcher and inserts a new one, where it learns to allow one prefix and block 40 prefixes via counterexamples in 41 iterations, regardless

[5] Configuration lines are estimated based on their core language representations, §3.1.

of the number of faults. The repair time is almost proportional to the number of iterations, and all repairs complete in less than 50 seconds.

In the right graph, the number of iterations is a constant (5) for all configuration sizes, because the optimal repair is to fix the 4 injected faulty literals, which requires only 5 iterations for any configuration size. The repair time grows slowly due to the larger SMT formula to be solved, but is still quite small (less than 5 seconds). We note that while this shows the ability of CB-Repair to scale well for a large configuration with a small number of faults, the specific performance may vary for different policies and other configurations.




#Faults (Config. Size=46) Config. Size (#Faults=4)

Fig. 5: Results for Evaluating Scalability of Repair.

*Evaluation on Internet2.* As a real-world case study, we evaluated CB-Repair on repair of the 3 violations reported by others for the `BlockToExternal` property. In our first attempt, CB-Repair picks a safe but impractical fix where *no* routes are exported to neighbors, which satisfies the `BlockToExternal` safety property. To find a more useful repair, we conjoin the `BlockToExternal` property with a reachability property that routes originated from internal nodes without the BTE community tag are advertised to neighbors. CB-Ver takes 37.7 seconds to verify this, and CB-Repair takes 1.3 seconds to successfully repair all 3 violations with 100% success rate (by adding a new clause that blocks routes with the BTE community).

## 6 Related Work

*Control plane verification.* There are many prior efforts in network control plane verification [39, 10, 5, 36, 25, 4, 31, 43, 11, 12, 32, 38, 45]. Our work is mostly related to approaches that are SMT-based and modular, including Timepiece [6], Lightyear [36] and CB-Ver [45].

*Error localization.* CEL [24] is a tool for localizing configuration errors through MCS (Minimal Correction Set), but it targets only ACL errors. Scalpel [41] is a tool to localize root causes of errors through sequential program analysis, on top of an existing simulation tool [21].

*Network Configuration synthesis and repair.* Configuration synthesis tools [13, 14, 19, 20, 35, 37, 3, 34, 33] are able to synthesize configuration for a whole network, given the properties specified by a user. However, these tools focus on network-wide properties and do not leverage verifiable modular interfaces. Marham [26],

CPR [23], ACR [28] and $S^2Sim$ [42] are tools to repair network configurations. Marham focuses on software defined networks and utilizes Horn clauses for repair, with a lattice-based search algorithm to ensure minimality of repair. CPR is based on ARC [25], a graph-based abstract representation of network control plane and ensures minimality of repairs through MaxSMT. ACR [28] considers a syntax driven generate-and-validate method, which makes localization-aware syntactic changes to network configurations and validates the effect of the repair using an external incremental verifier. $S^2Sim$ [42] is a more recent work on both diagnosing and repairing configuration errors. It derives local contracts of routers from the intended data plane and instantiates repair templates based on contracts violation. Snowcap [34] is a tool to find a safe ordering of reconfigurations, given the initial and final configurations.

## 7 Conclusions

In this paper, we present CB-Repair, a tool to automatically repair the network configurations when violations are found by the CB-Ver verifier. We use a symbolic template to enable candidate search for repairs via an optimizing SMT solver, and an iterative counterexample-guided approach in the repair algorithm to avoid solving formulas with quantifier alternation. Our evaluations show that this algorithm can automatically find different kinds of repair using a small number of iterations; it is also able to repair real-world Internet2 configurations in a few seconds, thanks to our modular and symbolic approach.

## References


1. More details about the october 4 outage. https://engineering.fb.com/2021/10/05/networking-traffic/outage-details/
2. Internet2. https://internet2.edu (2013)
3. Abhashkumar, A., Gember-Jacobson, A., Akella, A.: Aed: Incrementally synthesizing policy-compliant and manageable configurations. In: Proceedings of the 16th International Conference on emerging Networking EXperiments and Technologies. pp. 482–495 (2020). https://doi.org/10.1145/3386367.3431304
4. Abhashkumar, A., Gember-Jacobson, A., Akella, A.: Tiramisu: Fast multilayer network verification. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20). pp. 201–219 (2020)
5. Alberdingk Thijm, T., Beckett, R., Gupta, A., Walker, D.: Kirigami, the verifiable art of network cutting. In: 2022 IEEE 30th International Conference on Network Protocols (ICNP). pp. 1–12 (2022). https://doi.org/10.1109/ICNP55882.2022.9940333
6. Alberdingk Thijm, T., Beckett, R., Gupta, A., Walker, D.: Modular control plane verification via temporal invariants. Proceedings of the ACM on Programming Languages **7**(PLDI), 50–75 (2023). https://doi.org/10.1145/3591222
7. Alur, R., Bodik, R., Juniwal, G., Martin, M.M., Raghothaman, M., Seshia, S.A., Singh, R., Solar-Lezama, A., Torlak, E., Udupa, A.: Syntax-guided synthesis. In: 2013 Formal Methods in Computer-Aided Design. pp. 1–8. IEEE (2013). https://doi.org/10.1109/FMCAD.2013.6679385
8. Alur, R., Singh, R., Fisman, D., Solar-Lezama, A.: Search-based program synthesis. Communications of the ACM **61**(12), 84–93 (2018). https://doi.org/10.1145/3208071
9. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., et al.: cvc5: A versatile and industrial-strength smt solver. In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer (2022)
10. Beckett, R., Gupta, A., Mahajan, R., Walker, D.: A general approach to network configuration verification. In: Proceedings of the Conference of the ACM Special Interest Group on Data Communication. pp. 155–168 (2017). https://doi.org/10.1145/3098822.3098834
11. Beckett, R., Gupta, A., Mahajan, R., Walker, D.: Control plane compression. In: Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication. pp. 476–489 (2018). https://doi.org/10.1145/3230543.3230583
12. Beckett, R., Gupta, A., Mahajan, R., Walker, D.: Abstract interpretation of distributed network control planes. Proceedings of the ACM on Programming Languages **4**(POPL), 1–27 (2019). https://doi.org/10.1145/3371110
13. Beckett, R., Mahajan, R., Millstein, T., Padhye, J., Walker, D.: Don't mind the gap: Bridging network-wide objectives and device-level configurations. In: Proceedings of the 2016 ACM SIGCOMM Conference. pp. 328–341 (2016). https://doi.org/10.1145/2934872.2934909
14. Beckett, R., Mahajan, R., Millstein, T., Padhye, J., Walker, D.: Network configuration synthesis with abstract topologies. In: Proceedings of the 38th ACM SIGPLAN conference on programming language design and implementation. pp. 437–451 (2017). https://doi.org/10.1145/3062341.3062367
15. Bjørner, N., Phan, A.D., Fleckenstein, L.: $\nu$z-an optimizing smt solver. In: Tools and Algorithms for the Construction and Analysis of Systems: 21st International

Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015, Proceedings 21. pp. 194–199. Springer (2015). https://doi.org/10.1007/978-3-662-46681-0_14
16. CISCO: Route maps (Dec 2017), https://www.cisco.com/c/en/us/td/docs/security/asa/asa99/configuration/general/asa-99-general-config/route-maps.html
17. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement. In: CAV. pp. 154–169 (2000). https://doi.org/10.1007/10722167_15
18. De Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: International conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer (2008). https://doi.org/10.1007/978-3-540-78800-3_24
19. El-Hassany, A., Tsankov, P., Vanbever, L., Vechev, M.: Network-wide configuration synthesis. In: Computer Aided Verification: 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part II 30. pp. 261–281. Springer (2017). https://doi.org/10.1007/978-3-319-63390-9_14
20. El-Hassany, A., Tsankov, P., Vanbever, L., Vechev, M.: Netcomplete: Practical network-wide configuration synthesis with autocompletion. In: 15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18). pp. 579–594 (2018)
21. Fogel, A., Fung, S., Pedrosa, L., Walraed-Sullivan, M., Govindan, R., Mahajan, R., Millstein, T.: A general approach to network configuration analysis. In: 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15). pp. 469–483 (2015)
22. Ge, Y., de Moura, L.M.: Complete instantiation for quantified formulas in satisfiabiliby modulo theories. In: Computer Aided Verification (CAV). Lecture Notes in Computer Science, vol. 5643, pp. 306–320. Springer (2009). https://doi.org/10.1007/978-3-642-02658-4_25
23. Gember-Jacobson, A., Akella, A., Mahajan, R., Liu, H.H.: Automatically repairing network control planes using an abstract representation. In: Proceedings of the 26th Symposium on Operating Systems Principles. pp. 359–373 (2017). https://doi.org/10.1145/3132747.3132753
24. Gember-Jacobson, A., Shrestha, R., Sun, X.: Localizing router configuration errors using minimal correction sets. arXiv preprint arXiv:2204.10785 (2022). https://doi.org/10.48550/arXiv.2204.10785
25. Gember-Jacobson, A., Viswanathan, R., Akella, A., Mahajan, R.: Fast control plane analysis using an abstract representation. In: Proceedings of the 2016 ACM SIGCOMM Conference. pp. 300–313 (2016). https://doi.org/10.1145/2934872.2934876
26. Hojjat, H., Rümmer, P., McClurg, J., Černỳ, P., Foster, N.: Optimizing horn solvers for network repair. In: 2016 Formal Methods in Computer-Aided Design (FMCAD). pp. 73–80. IEEE (2016). https://doi.org/10.1109/FMCAD.2016.7886663
27. Jayaraman, K., Bjørner, N., Padhye, J., Agrawal, A., Bhargava, A., Bissonnette, P.A.C., Foster, S., Helwer, A., Kasten, M., Lee, I., Namdhari, A., Niaz, H., Parkhi, A., Pinnamraju, H., Power, A., Raje, N.M., Sharma, P.: Validating datacenters at scale. In: Proceedings of the ACM Special Interest Group on Data Communication. p. 200–213. SIGCOMM '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3341302.3342094
28. Liu, X., Zhang, P., Abhashkumar, A., Chen, J., Jiang, W.: Automatic configuration repair. In: Proceedings of the 23rd ACM Workshop on Hot Top-

ics in Networks. p. 213–220. HotNets '24, Association for Computing Machinery, New York, NY, USA (2024). https://doi.org/10.1145/3696348.3696895, https://doi.org/10.1145/3696348.3696895
29. Lougheed, K., Rekhter, Y.: Rfc1267: Border gateway protocol 3 (bgp-3) (1991). https://doi.org/10.17487/RFC1267
30. Networks, J.: Policy framework overview (Nov 2023), https://www.juniper.net/documentation/us/en/software/junos/routing-policy/topics/concept/policy-routing-overview.html
31. Prabhu, S., Chou, K.Y., Kheradmand, A., Godfrey, B., Caesar, M.: Plankton: Scalable network configuration verification through model checking. In: 17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20). pp. 953–967 (2020)
32. Raghunathan, D., Beckett, R., Gupta, A., Walker, D.: Acorn: Network control plane abstraction using route nondeterminism. In: FMCAD. pp. 261–272 (2022). https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_33
33. Ramanathan, S., Zhang, Y., Gawish, M., Mundada, Y., Wang, Z., Yun, S., Lippert, E., Taha, W., Yu, M., Mirkovic, J.: Practical intent-driven routing configuration synthesis. In: 20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23). pp. 629–644 (2023)
34. Schneider, T., Birkner, R., Vanbever, L.: Snowcap: Synthesizing network-wide configuration updates. In: Proceedings of the 2021 ACM SIGCOMM 2021 Conference. pp. 33–49 (2021). https://doi.org/10.1145/3452296.3472915
35. Subramanian, K., D'Antoni, L., Akella, A.: Synthesis of fault-tolerant distributed router configurations. Proceedings of the ACM on Measurement and Analysis of Computing Systems **2**(1), 1–26 (2018). https://doi.org/10.1145/3179425
36. Tang, A., Beckett, R., Benaloh, S., Jayaraman, K., Patil, T., Millstein, T., Varghese, G.: Lightyear: Using modularity to scale bgp control plane verification. In: Proceedings of the ACM SIGCOMM 2023 Conference. pp. 94–107 (2023). https://doi.org/10.1145/3603269.3604842
37. Tian, B., Zhang, X., Zhai, E., Liu, H.H., Ye, Q., Wang, C., Wu, X., Ji, Z., Sang, Y., Zhang, M., Yu, D., Tian, C., Zheng, H., Zhao, B.Y.: Safely and automatically updating in-network acl configurations with intent language. In: Proceedings of the ACM Special Interest Group on Data Communication, pp. 214–226 (2019). https://doi.org/10.1145/3341302.3342088
38. Wang, D., Zhang, P., Gember-Jacobson, A.: Expresso: Comprehensively reasoning about external routes using symbolic simulation. In: Proceedings of the ACM SIGCOMM Conference. pp. 197–212. ACM (2024). https://doi.org/10.1145/3651890.3672220
39. Weitz, K., Woos, D., Torlak, E., Ernst, M.D., Krishnamurthy, A., Tatlock, Z.: Scalable verification of border gateway protocol configurations with an smt solver. In: Proceedings of the 2016 acm sigplan international conference on object-oriented programming, systems, languages, and applications. pp. 765–780 (2016). https://doi.org/10.1145/2983990.2984012
40. Wikipedia contributors: 2022 Rogers Communications outage. https://en.wikipedia.org/wiki/2022_Rogers_Communications_outage (2022), accessed: 2026-09-10
41. Yang, R., Fang, X., You, L., Xiang, Q., Shao, H., Han, G., Wang, Z., Zhang, Z., Shu, J., Kong, L.: Diagnosing distributed routing configurations using sequential program analysis. In: Proceedings of the 7th Asia-Pacific Workshop on Networking. pp. 34–40 (2023). https://doi.org/10.1145/3600061.3600065

42. Yang, R., Han, G., Shao, H., Zheng, X., Fang, X., Wang, Z., You, L., Zhou, R., Kong, L., Zhai, E., Xiang, Q., Shu, J.: Diagnosing and repairing distributed routing configurations using selective symbolic simulation. In: 23rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 26). pp. 1635–1652. USENIX Association, Renton, WA (May 2026), https://www.usenix.org/conference/nsdi26/presentation/yang
43. Ye, F., Yu, D., Zhai, E., Liu, H.H., Tian, B., Ye, Q., Wang, C., Wu, X., Guo, T., Jin, C., She, D., Ma, Q., Cheng, B., Xu, H., Zhang, M., Wang, Z., Fonseca, R.: Accuracy, scalability, coverage: A practical configuration verifier on a global wan. In: Proceedings of the Annual conference of the ACM Special Interest Group on Data Communication on the applications, technologies, architectures, and protocols for computer communication. pp. 599–614 (2020). https://doi.org/10.1145/3387514.3406217
44. Zhang, D.: Artifact of automatic repair of network configurations using a modular verifier and symbolic templates (Sep 2026). https://doi.org/10.5281/zenodo.22691256
45. Zhang, D., Thijm, T.A., Walker, D., Gupta, A.: Cb-ver: A stable foundation for modular network control plane verification. In: Proceedings of the 26th Conference on Formal Methods in Computer-Aided Design–FMCAD 2026: Conference Series: Formal Methods in Computer-Aided Design. p. 402. TU Wien Academic Press (2026). https://doi.org/10.34727/2026/isbn.978-3-85448-093-8_45

## A Semantics of Core Language

We present the semantics of our core language of route maps here. Recall the syntax of our language (without templates):

$$
\begin{aligned}
r \in \text{RouteMap} &:= (cl;)^* \\
cl \in \text{Clause} &:= (mt;)^*(md;)^* a \\
mt \in \text{Matcher} &:= \mathbf{MatchPrefix}(l^*) \mid \mathbf{MatchComm}(l^*) \\
md \in \text{Modifier} &:= \mathbf{AddComm}(l^*) \mid \mathbf{SetComm}(l^*) \\
&\quad \mid \mathbf{RemoveComm}(l^*) \mid \mathbf{SetLP}(l) \\
l \in \text{Literal} &:= \text{PrefixLiteral} \mid \text{CommunityLiteral} \\
&\quad \mid \text{LPLiteral}
\end{aligned}
$$

The semantics of a matcher $mt$ is a function $\text{match}(mt) : S \to \{\mathbf{true}, \mathbf{false}\}$ ($S$ is the set of routes), mapping a route $s$ to either **true** (if matched) or **false** (if not matched). The function match is defined as:

$$
\begin{aligned}
&\text{match}(\mathbf{MatchPrefix}(L))(s) \\
&\quad \Leftrightarrow s \neq \infty \wedge \bigvee_{l \in L} \text{prefix}(s) = l \\
&\text{match}(\mathbf{MatchComm}(L))(s) \\
&\quad \Leftrightarrow s \neq \infty \wedge \bigvee_{l \in L} l \in \text{comm(s)}
\end{aligned}
$$

Here, $L$ is a set of literals (corresponding to the list $l^*$ in the syntax), $\infty \in S$ is a special route for "no route" (and therefore cannot be matched), $\text{prefix}(s)$ is the prefix of a route $s \in S$, and comm(s) is the set of communities of a route $s \in S$. Moreover, we interpret **MatchPrefix** as an exact match between prefixes (we require $\text{prefix}(s) = l$ in our semantics). We plan to support more patterns of match in the future.

The semantics of a modifier $md$ is a function $\text{modify}(md) : S \to S$, mapping a route $s$ to a new route that is modified. The function modify is defined as:

$$
\begin{aligned}
&\text{modify}(\mathbf{AddComm}(L))(s) \\
&\quad = \text{WithComm}(s, \text{comm}(s) \cup L) \\
&\text{modify}(\mathbf{SetComm}(L))(s) \\
&\quad = \text{WithComm}(s, L) \\
&\text{modify}(\mathbf{RemoveComm}(L))(s) \\
&\quad = \text{WithComm}(s, \text{comm}(s) - L) \\
&\text{modify}(\mathbf{SetLP}(l))(s) = \text{WithLP}(s, l)
\end{aligned}
$$

Here,

$$\text{WithComm}(s, L) = (\text{prefix}(s), \text{lp}(s), \text{path}(s), L)$$

and

$$\text{WithLP}(s, l) = (\text{prefix}(s), l, \text{path}(s), \text{comm}(s))$$

are functions that update the communities or local preference of $s$ with new value $L$ or $l$.

The semantics of a clause $cl$ is a binary function $\text{apply}(cl) : S \times S \to S$; the first argument of $\text{apply}(cl)$ is the input route to be matched or modified by this clause; the second argument is the *continuation*, the route that is applied by the remaining clauses coming after the current clause. The interpretation of apply is:

$$\text{apply}(mt_1; \cdots; mt_n; md_1; \cdots; md_m; a)(s_1, s_2)$$
$$= \begin{cases} \text{modify}(md_m)(\cdots \text{modify}(md_1)(s_1)) & a = \textbf{permit} \wedge \bigwedge_{i=1}^{n} \text{match}(mt_i)(s_1) \\ \infty & a = \textbf{deny} \wedge \bigwedge_{i=1}^{n} \text{match}(mt_i)(s_1) \\ s_2 & \neg \bigwedge_{i=1}^{n} \text{match}(mt_i)(s_1) \end{cases}$$

If $s_1$ is not matched by this clause, the continuation $s_2$ is returned. If $s_1$ is matched but the action is to **deny**, an invalid route (dropped route) $\infty$ is returned. If $s_1$ is matched and the action is to **permit**, the modified route is returned.

The semantics of a route map $r$ is a transfer function $\text{transfer}(r) : S \to S$, interpreted as follows:

$$\text{transfer}(cl_1; \cdots; cl_n)(s)$$
$$= \text{apply}(cl_1)(s, \text{apply}(cl_2)(s, \cdots \text{apply}(cl_n)(s, \infty))).$$

Note that if no clause is matched for input route $s$, the default action of a route map is to deny it.