Network Configuration Synthesis with Abstract Topologies

Ryan Beckett* Ratul Mahajan†‡ Todd Millstein†§ Jitendra Padhye‡ David Walker*

Princeton University, USA* Intentionet, USA† Microsoft, USA‡ UCLA, USA§

Abstract

We develop Propane/AT, a system to synthesize provably-correct BGP (border gateway protocol) configurations for large, evolving networks from high-level specifications of topology, routing policy, and fault-tolerance requirements. Propane/AT is based on new abstractions for capturing parameterized network topologies and their evolution, and algorithms to analyze the impact of topology and routing policy on fault tolerance. Our algorithms operate entirely on abstract topologies. We prove that the properties established by our analyses hold for every concrete instantiation of the given abstract topology. Propane/AT also guarantees that only incremental changes to existing device configurations are required when the network evolves to add or remove devices and links. Our experiments with real-world topologies and policies show that our abstractions and algorithms are effective, and that, for large networks, Propane/AT synthesizes configurations two orders of magnitude faster than systems that operate on concrete topologies.

CCS Concepts •**Networks** → *Network manageability*; *Network management*; •**Software and its engineering** → *Domain specific languages*

Keywords Propane/AT; Domain-specific Language; BGP; Fault Tolerance; Compilation; Network Management

1. Introduction

Computer networks run many critical services, and every second of downtime is costly at best and dangerous at worst. Yet, keeping these systems running 24/7 is an enormous challenge [10, 11, 15, 26]. While hardware faults, backhoes, power failures and natural disasters all pose problems, studies have shown that human error in network configuration is one of the leading causes of outages [21, 29].

To configure large networks, instead of considering individual devices, operators classify devices into *roles*. A role refers to specific functionality and is served by one or more devices. For instance, in a data center, roles may be "top-of-rack," "aggregation," and "spine" routers; and in a backbone network, they may be "core" and "border" routers. While the network may have hundreds or thousands of devices—a scale that is impossible for humans to handle—there tend to be only a handful of roles. Operators author a configuration *template* for each role. Templates are macros that, given a network topology, can be instantiated with different concrete values to generate device configurations.

Unfortunately, templates use the same low-level constructs as ordinary router configurations (*e.g.*, adding or removing tags from announcements). In both templatized and non-templatized configurations, ensuring that a collection of low-level, local configurations achieves some important network-wide policy, such as a guarantee of connectivity in the face of failures, is extremely challenging. To validate their templates, operators will typically first instantiate a template with appropriate concrete parameters and then test it under various scenarios. In general, like in any complex software system, such testing is inherently incomplete.

Moreover, even if network operators were to instantiate their templates using the initial network topology and verify key properties using tools such as Bagpipe [36], the guarantees would not hold as the network topology evolves. Evolution of the topology is a frequent event for large networks, as devices and links are taken offline for maintenance and added to expand capacity. Templates that work for the current topology may or may not work for future topologies. Ensuing problems may cause operators to make non-uniform changes to routers' configurations, which defeats the purpose of a template system. An even worse situation is when operators must update many devices to evolve their network. Such network-wide configuration changes entail a great deal of risk and can be highly disruptive to live traffic.

Given the challenges of generating and validating network configurations, one might think that operators will be

PLDI'17, June 18–23, 2017, Barcelona, Spain
ACM. 978-1-4503-4988-8/17/06...$15.00
http://dx.doi.org/10.1145/3062341.3062367

receptive to systems that can generate provably-correct configurations from high-level policy specifications [5, 8, 28]. However, our conversations with two major cloud providers reveal that operators of large networks are reluctant to use such systems. While they think of their network abstractly, in terms of roles, current synthesis systems operate over concrete topologies. Even if two devices play the same role, operators cannot specify policy in terms of this role; and even if specifications for the two devices are similar, there is no guarantee that the systems will generate (syntactically) similar configurations. Perhaps most importantly, if the operators want to debug or analyze system output, they will have to consider hundreds of device configurations instead of just a handful of role configurations. Current synthesis systems are also brittle in the face of network evolution. Any change in network topology requires re-execution of the engine, from scratch, on the new topology, and the result may be a completely different set of configurations. No operator can shut down a large, production network, upgrade policy on all devices and then restart their network.

To address the challenge of configuration synthesis in the presence of abstract roles, we develop Propane/AT. Propane/AT allows operators to input *abstract topologies* in terms of roles and their connectivity. For instance, they may specify roles for "top-of-rack" and "aggregation" routers and specify that every top-of-rack router connects to at least two aggregation routers (to tolerate the failure of a link to an aggregation router). Propane/AT takes two additional inputs. The first is a high-level specification of routing policy, for which we borrow notation from the original Propane system [5]. While Propane policies refer to concrete devices, Propane/AT policies refer to abstract roles. The final input to Propane/AT is the fault-tolerance requirements of the network, such as the number of simultaneous link failures it can tolerate without loss of connectivity for any traffic flow.

Based on these inputs, Propane/AT generates one template per role. These templates specify routing policy using BGP (border gateway protocol). BGP is the standard protocol for interdomain routing and is also commonly used within data centers because of its scalability and support for rich policies.[1] Our templates are correct for *any* concrete topology that complies with the abstract topology. They are also evolution friendly. When the network evolves from one compliant concrete topology to another, only the configurations of devices that acquire or lose a neighbor need to change. This guarantee is the best that any system can give as neighbor relationships are explicitly configured in devices. We achieve it in part by leveraging BGP features that allow arbitrary tags on routing announcements and expressing policy using such tags (instead of router identifiers).

During synthesis, our compiler analyzes abstract topologies to determine the fault tolerance properties of the specified routing policy. This analysis yields a lower bound on the number of link failures required to disconnect one abstract location from another. Any concrete instance of the abstract topology will adhere to the given fault tolerance property.

We use Propane/AT for a range of real-world topologies and routing policies. We find that it can effectively encode the topologies and their evolution, and the fault tolerance bounds that it computes are often precise. We also find that Propane/AT scales substantially better than its nearest competitor, the original Propane system, which operates over concrete topologies. As the number of devices in a network grows, the number of distinct roles will often stay constant. Consequently, Propane/AT can be two orders of magnitude faster than Propane, taking less than 10 seconds to synthesize templates for large networks.

Contributions: To summarize, our contributions are:

- New topology abstractions for network programming based on graph homomorphisms and connectivity rules.
- New algorithms for analyzing routing policy and fault tolerance over abstract topologies.
- Implementation and evaluation of a Propane/AT compiler that generates BGP templates and is much faster than compilers based on concrete topologies.

2. Background

In this work, we focus on synthesizing configurations for a network that runs BGP, the standard for routing across independent organizations and inside large data centers. It scales effectively and allows operators to define flexible policies. We provide a brief background on BGP focused on aspects relevant to our work.

The way in which a BGP-based network forwards traffic depends on how the routers exchange and process *routing announcements*. To invite traffic from a neighbor to a destination, routers announce the destination's address *prefix* (e.g., 10.1.1.0/24[2]) to the neighbor. Traffic flows in the opposite direction to such route announcements.

All route announcements carry an *AS path*, the sequence of BGP networks, called autonomous systems (ASes), that will be traversed to reach the destination. Optionally, they may also include one or more *communities*, which are arbitrary tags whose semantics are agreed upon out of band.

When a BGP router receives one or more announcements for a prefix from different neighbors, it selects which one to use (if any) based on its configuration. This selection can depend on the AS path, communities, or the neighbor that sent the announcement, and it can even discard certain announcements (*i.e.*, never use them for sending traffic). Then, based again on its configuration, the router forwards the selected route to one or more neighbors, after adding itself to the AS path, and optionally, modifying communities.

[1] However, BGP is not the only routing protocol used in existing networks. In the future, we will extend our work to other protocols such as OSPF.

[2] An IP address such as 10.1.1.0 is 32 bits, with each of the four values representing 8 bits. The prefix 10.1.1.0/24 denotes the set of addresses that share the first 24 bits with 10.1.1.0, *i.e.*, all addresses that begin with 10.1.1.

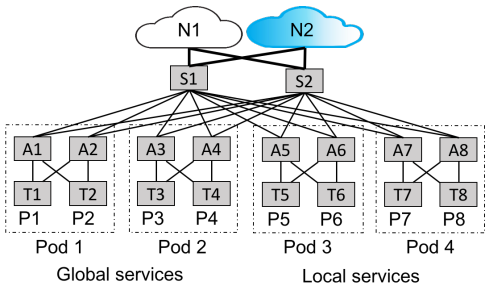


Figure 1: An example data center network.

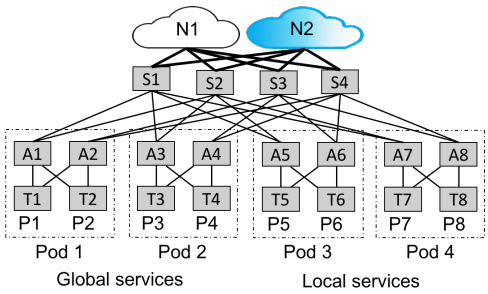


Figure 2: A modified version of the network in Figure 1.

When announcing destinations to certain neighbors, a BGP router may perform *route aggregation*, *i.e.*, announce a single prefix that covers multiple prefixes (*e.g.*, announce 10.1.0.0/16 instead of 10.1.[1–255].0/24). The covering prefix is announced if the router has a valid route for *any* of the covered prefixes. Aggregation helps reduce the memory consumption of routers, but it can lead to traffic black holes [24]. A router that announces 10.1.0.0/16, because it has a route to 10.1.1.0/24, may also get traffic for 10.1.2.0/24 to which it has no route.

3. Motivation

We demonstrate the difficulty of configuring networks today using the data center example in Figure 1. The boxes denote routers. Using terminology for fat tree networks [3], S[1–2] are spine routers, A[1–8] are aggregation routers,[3] and T[1–8] are top-of-rack (ToR) routers. The spine routers connect to the Internet through neighbors N[1–2]. The ToR routers attach to a set of servers ("a rack") that host services with address prefixes P[1–8].

The intended policy for this network is: (1) complete internal connectivity, *i.e.*, all routers should be able to reach each other; (2) services in Pods[1–2] should be accessible from outside; (3) prefixes for global service should be aggregated into a covering prefix PG when announced outside; (4) services in Pods[3–4] should not be externally accessible; (5) traffic paths should be valley-free (*e.g.*, a path through S1 should not go down through Ai and then back up through S2, for instance, creating an up-down-up path); (6) prefer neighbor N1 over N2, *i.e.*, when both neighbors announce a prefix, send traffic through N1; (7) routers should not transit traffic between N1 and N2; and (8) no loss in connectivity after any single-link failure.

[3] This term is not related to BGP route aggregation.

```
%{for $n in neighbors do}%
interface $n.interface
  ip address $n.interfaceIP
!
%{end for}%

router bgp %{$bgpASN}%
  no synchronization
  bgp router-id %{$routerID}%
  bgp bestpath compare-routerid
%{for $n in neighbors do}%
  neighbor $n.IP remote-as $n. ASN
  neighbor $n.IP send-community both
  neighbor $n.IP route-map peer-in in
  %{if $n.isLocalPeer then}%
  neighbor $n.IP route-map local out
  %{else}%
  neighbor $n.IP route-map global out
  %{end if}%
%{end for}%
```

```
%{for $p in localPrefixes do}%
ip prefix-list localPL permit $p
%{end for}%
!
ip prefix-list other permit 0.0.0.0/0 le 32
!
route-map local permit 10
   match ip address prefix-list localPL
!
route-map local deny 20
   match ip address prefix-list other
!
route-map peer-in permit 10
   match as-path list WAN1
   set community additive 64512:3200
!
route-map peer-in permit 20
  match as-path list WAN2
  set local-preference 90
  set community additive 64512:3201
```

Figure 3: Idealized small configuration component for the data center spine based on templates from a cloud provider.

To correctly configure this policy, operators must generate configurations for each router, which implies ensuring, for instance, that all routing adjacencies are correctly configured (*e.g.*, T1's configuration includes A1 as neighbor and vice-versa); the ToRs announce the correct prefixes for their services; all routers forward the prefix announcements that they should to each neighbor and not forward others (*e.g.*, the spines should forward prefixes for local services to internal neighbors but not to external neighbors); and the spines announce externally only the covering global prefix. Such configuration tasks are highly complex [1, 5, 11, 29].

To simplify them, operators are adopting a template-based approach [19, 35]. Instead of authoring a configuration per router, operators author a template per role. A *role* is a specific function that is served by one or more routers. For example, the network in Figure 1 might have five roles: spine, global aggregator, global ToR, local aggregator, and local ToR. Figure 3 shows an example of what a small component of a template for the spine role in the two data centers might look like. The template has parameters for various aspects of the configuration (*e.g.*, neighbor list, local prefixes) and is compiled to low-level device configurations by instantiating the parameters using the network topology and a database of network information.

As described earlier (§1), templates are hard to author and hard to validate. Worse, templates that work for one topology may not work for seemingly-inconsequential variations which may arise after the network evolves. Consider the network in Figure 2, which is similar to Figure 1; it has the same five roles, connected in a similar hierarchy. One might think that the same templates, with different database entries, can be used for both cases. However, if the templates are configured to disallow "valley" paths (per policy (5) above), they

will work for Figure 1 but silently violate the fault tolerance policy (8) when used for Figure 2. Specifically, in Figure 2, an aggregation-based black hole (§2) will occur when the link S1–A1 fails; after this failure, S1 has no valley-free path to P[1–2] even though it will continue to get traffic for these prefixes as it announces the covering prefix PG (because it gets routes for P[3–4]). Such a black hole will not occur in Fn Figure 1 because spine routers have two links to each pod.

When operators discover that an old template no longer works, they may consider changing it, which may cause a change to all devices that use it—an unacceptable disruption in many cases. As a result, operators may abandon the template entirely and revert to hand-crafting configuration patches to accommodate the change. Such patches reintroduce the complexity and the risk of errors that templates were meant to prevent.

4. Propane/AT Overview

Propane/AT takes as input the network's abstract topology and its policy. The policy consists of the routing policy that describes how traffic should flow and the fault-tolerance policy that describes how many simultaneous link failures the network should withstand without losing connectivity.

Abstract Topology Abstract topologies in Propane/AT define structural and role-based invariants that compactly describe all concrete networks that can emerge as the network evolves. They are encoded in the form of a graph homomorphism annotated with logical constraints about node and edge multiplicities. We designed the abstractions to be able to precisely capture real network topologies, while being amenable to fault-tolerance analysis in the abstract domain.

Our topology abstractions consist of several concepts. The primary one is a role-based abstraction that allows an operator to map routers in the concrete network to roles in the abstract network. Figure 4 shows an example of this abstraction for both networks from §3. In the example, the concrete networks are abstracted into a new topology with 5 different roles: local ToR (TL), global ToR (TG), local aggregator (AL), global aggregator (AG), and spine (S).

More specifically, a network topology is a graph $G = (V, E)$, which consists of a set of vertices V and a set of directed edges $E \colon V \times V$. A role-based abstraction is a graph homomorphism from G to an abstract graph $G^A = (V^A, E^A)$. A graph homomorphism $f : G \to G^A$ maps each node in the concrete graph to a node in the abstract graph such that, whenever $(u, v) \in E$, then $(f(u), f(v)) \in E^A$. The role-based abstraction therefore over-approximates the connectivity of the underlying concrete graphs.

On its own, this abstraction loses a lot of information about the concrete network's structure, making it difficult to reason precisely about fault-tolerance. For example, with this abstraction any spine router may or may not connect to any aggregator router. To capture concrete networks more

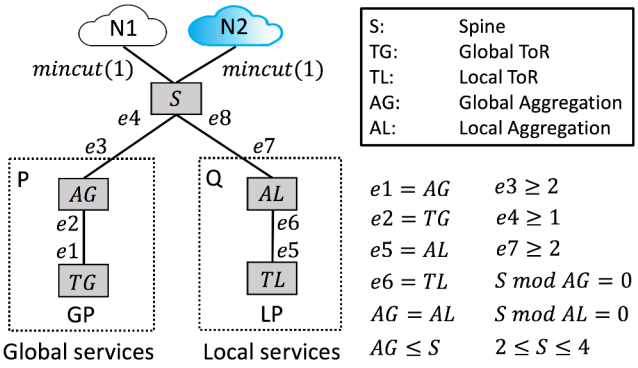


Figure 4: An abstraction for the network in Figure 1.

precisely, we introduce additional concepts. The first is topology hierarchy, captured by P and Q, which indicate that nodes in the ToR and aggregator roles are grouped into pods. The second is node and edge *multiplicity*. Each edge (and node) is labeled with a symbolic variable (*e.g.*, $e1$) that denotes a constraint on the number of edges (and nodes) that may appear in any valid concrete network. Operators can capture concrete network invariants by adding constraints on the symbolic variables using logical formulas.

For example, in Figure 4, the first constraint ($e1 = AG$) states that, within any pod P, the number of outgoing edges from a node in the TG role (*i.e.*, $e1$) to a node in the AG role equals the number of nodes in the AG role. Similarly, the constraint ($e2 = TG$) states that the number of outgoing edges from a node in the AG role (*i.e.*, $e2$) to a node in the TG role equals the number of nodes in the TG role. Together these constraints capture the fact that, within any pod, the global aggregators and ToRs are in a full mesh. Furthermore, the constraints $AG = AL$ and $AG \leq S$ ensure that, within pods P and Q, the AG and AL roles have the same number of routers, which is less than or equal to the number of routers in the spine role S. The constraint $e3 \geq 2$ says that, in each pod, each aggregator node has at least 2 outgoing edges to nodes in the spine role. Symmetrically, the constraint $e4 \geq 1$ says that, for each pod P, each node in the spine role has at least one outgoing edge to a node in the AG role. Similar constraints appear for the local aggregator role. The constraint $2 \leq S \leq 4$ makes explicit the possibility for growth, for example, by growing the network from Figure 1 to Figure 2. In general, we need not bound the number of spine routers to admit more concrete topologies, potentially at the expense of analysis precision. We also include the constraints $(S \mod AG) = 0$, and $(S \mod AL) = 0$ simply to show that constraints do not have to be in the form of inequalities. Operators can use logical formulas from any theory supported by modern SMT solvers.

A final concept is the $mincut(1)$ constraint between the spine role S and N[1–2]. It says that any spine router has at least one path to any node in the neighbor N1 (and N2) role. Such annotations are useful for a "one big switch"

abstraction [7] in which a complex, unstructured network is represented as a single node. As another use case, an ISP backbone can be modeled by dividing the network into separate geographic regions with two roles per region—one for the border routers and another for the network core. Mincut annotations can describe the degree of fault tolerance both within regional cores and across regions.

Our topology abstractions can also capture concrete topologies by using a one-to-one correspondence between abstract and concrete nodes/edges. This allows operators to define complex networks in which some (e.g., legacy) parts of the network cannot evolve while others can.

Routing Policy Routing policies in Propane/AT consist of an ordered sequence of i) a predicate that matches a class of traffic; and ii) paths that the traffic should take through the network, ranked per their relative preference. Propane/AT borrows syntax from Propane [5], but instead of concrete predicates and paths, uses abstract predicates and paths.

Let us see how to express the routing policy of the networks in §3 over the abstract topology. We can capture the basic routing behavior, constraints (1, 2, 6), as follows:

```
define Routing =
    $GP  => end(TG)
    $LP  => end(TL)
    true => end(out) & exit(N1 >> N2)
```

The second line introduces a prefix *template* variable `$GP`. Template variables represent multiple instances of a rule for different concrete prefixes that can be provided by an external source (*e.g.*, a database). The line says that traffic for each global prefix associated with the variable should follow a path that ends at its corresponding destination router in the `TG` role. As we will see in §5, constraints like **`end`**`(TG)` are just syntactic sugar for regular expressions describing network paths. The second line has a similar policy for local prefixes. The final rule matches all other IP prefix destinations and allows traffic to follow a path that leaves the data center, ending at some external role (**`out`**), through neighbors `N1` or `N2` with a preference for leaving through `N1`. The `>>` symbol indicates that traffic should satisfy the constraint on the left and resort to the backup (right) only when that is not possible due to network failures.

Next, we can capture constraint (4) that traffic for local prefixes must stay within in the data center:

```
define Local =
    $LP => always(in)
```

The `Local` policy adds the **`always`**`(in)` constraint for each local prefix described by the template variable `$LP`. This constraint ensures that traffic follows a path that matches an internal role (*i.e.*, in the data center) at each hop of the path. The constraint to prevent "valleys" (5) is written as:

```
define NoValley =
    true => novalley({TG,TL},{AG,AL},{S})
```

This policy applies to all traffic and prevents valley paths by adding the **`novalley`** constraint with arguments corresponding to each level in the data center. Constraint (7) to prevent transit traffic between neighbors is expressed as:

```
define Peer = {N1,N2}
define NoTransit =
    true => !(enter(Peer) & exit(Peer))
```

We define a `Peer` as `N1` or `N2` and disallow paths where traffic both enters and exits the data center through a peer.

Finally, we can combine these constraints together as:

```
Routing & Local & NoTransit &
NoValley & agg(GP_AGG, in -> out)
```

This instructs Propane/AT to satisfy the conjunction of all the constraints. It also declares that we want to perform route aggregation, with covering prefix GP_AGG, for global prefixes at the border of the data center (*i.e.*, along any edge that connects the inside of the data center to the outside). In this case, GP_AGG is declared as a concrete prefix rather than a template because a single aggregate prefix will be used to summarize all less-specific prefixes.

Fault-Tolerance Policy This policy specifies how many link failures the network should be able to withstand before traffic experiences connectivity loss. Operators can specify different tolerance levels for different pairs of abstract nodes. For instance, they may say that ToR to spine connectivity should be robust to 2 failures, *i.e.*, no ToR-spine pair should lose connectivity as long as the number of simultaneous link failures is 2 or fewer; and ToR-to-ToR connectivity should be robust to 1 failure.

Synthesis Propane/AT generates templates from the inputs above in three phases. First, it combines the abstract topology and routing policy into a *product graph* (PG) that compactly captures the flow of routing information over the topology in a policy-compliant manner. The algorithm for this phase builds on Propane (for concrete networks); we show that it can be extended to abstract inputs.

Second, Propane/AT checks if the fault-tolerance policy can be met by considering the joint impact of the abstract topology and routing policy. Joint analysis is needed, because connectivity depends on both: traffic will not flow along valid topological paths if the routing policy disallows it. We develop a sound analysis based on computing the minimum number of edge-disjoint, policy-compliant paths between pairs of nodes over abstract topologies. If this number is less than the desired fault-tolerance level for a pair of nodes, Propane/AT declares that the policy cannot be met.

If it can be met, Propane/AT generates templates for each node in the abstract topology. Per-device BGP configurations can in turn be generated from the templates using information in the concrete topology. To ensure the process is sound, Propane/AT checks that the given concrete topology is an instance of the abstract topology.

d	$\in$	*Integers*	
x	$\in$	*Variables*	
l	$\in$	*Topology Locations*	
pol	::=	$p_1, \ldots, p_n$	Policy
p	::=	t => r_1>>...>>r_m | a	Path Preferences
t	::=	$\$x$ | $d.d.d.d/[d..d]$	Predicate
a	::=	$agg(t, r_1 \to r_2)$	Aggregation
r	::=	l | $\emptyset$ | `in` | `out` | $r_1 \cup r_2$ | $r_1 \cap r_2$ | $r_1 \cdot r_2$ | `!`r | r^*	Regular Path

Figure 5: Propane/AT core syntax.

The following sections detail each of the three phases.

5. Product Graph Generation

The first step for the Propane/AT compiler is to build the product graph (PG), a data structure that is amenable to joint analysis of the topology and routing policy. As a precursor to this step, we convert the routing policy syntax in §4 to a core language based on regular expressions, shown in Figure 5. A policy has one or more constraints, each an aggregation or a path constraint. Path constraints have a test on a destination prefix and a list of regular expressions describing network paths. Regular paths are defined over network locations, where each location is either a router inside the operator's network, or an external neighbor connected to that network. Values **`in`** and **`out`** will match any internal and external location respectively. There are two types of predicates, as shown. A test for a concrete prefix $d.d.d.d/[d..d]$ matches a range of IP prefixes (*e.g.*, 10.0.1.0/[24..32]) where metavariable d represents an integer. A prefix template test $x, for variable x, represents a collection of many tests, one for each prefix represented by the template.

Converting from the high-level syntax from §4 to the core syntax is straightforward. The predicate `true` becomes the prefix range $0.0.0.0/[0..32]$. Each constraint desugars to a regular expression. For example, the constraint **`always`**`(`**`in`**`)` becomes $\mathbf{in}^*$ and the constraint **`end`**`(TG)` becomes $\Sigma^* \cdot$ `TG`. Preferences are lifted to the top level of the regular expression when their use is unambiguous, and separate sets of constraints are joined prefix-by-prefix by taking the regular expression intersection of their constraints.

We are now ready to generate the PG. Intuitively, the PG captures topology and routing constraints by "intersecting" the finite automata associated with the policy regular expressions with the graph structure of the topology. Because there can be a separate routing policy for each predicate t in the policy, we construct one PG for each predicate.

For each regular path constraint r_i from r_1 >> ... >> r_k, we construct a DFA for the reverse of r_i. A DFA for r_i is defined as a tuple $(\Sigma, Q_i, F_i, q_{0_i}, \sigma_i)$. The alphabet Σ is the set of topology locations (*i.e.*, routers or roles), Q_i is the set of states for automaton i, F_i is the set of final states, q_{0_i} is the initial state, and $\sigma_i \colon Q_i \times \Sigma \to Q_i$ is the state transition function. The PG for a graph (*i.e.*, a topology) $G = (V, E)$ is a tuple (G', $start$, rank) where $G' = (V', E')$ has vertices $V' \colon V \times Q_1 \times \cdots \times Q_j$, edges $E' \colon V' \times V'$, a unique starting vertex $start$, and a ranking function $\mathrm{rank} \colon V' \to 2^{\{1,\ldots,j\}}$, mapping nodes in the PG to a set of path ranks.

The PG is constructed by adding an edge from state $m = (l_m, q_{m_1}, \ldots, q_{m_k})$ to $n = (l_n, q_{n_1}, \ldots, q_{n_k})$ whenever $\sigma_i(q_{m_i}, l_n) = q_{n_i}$ for each i and $(l_m, l_n) \in E$. We add edges from the $start$ node to any $m = (l, q_{m_1}, \ldots, q_{m_k})$ when $\sigma_i(q_{0_i}, l) = q_{m_i}$ for each i. The ranking function $\mathrm{rank}(m)$ denotes the rank of paths through the PG ending at node m and is defined as $\mathrm{rank}(m) = \{i \mid q_{m_i} \in F_i\}$. Finally, we write $\mathrm{topo}(m) = l$ to extract the topology location from a PG node, when $m = (l, q_{m_1}, \ldots, q_{m_k}) \in V'$. For the remainder of the paper, we use the term *location* to refer to a router in the case of a concrete topology/PG, or a role in the case of an abstract topology/PG.

Figure 6 shows the PG for the data center policy that applies to all external traffic (`true=>`**`exit`**`(N1 >> N2)`). The first automaton represents the more preferred constraint **`exit`**`(N1)` and the second automaton represents the less preferred constraint **`exit`**`(N2)`. The PG is shown for both an instance of a simple concrete network matching the abstraction from §4 as well as for the abstract topology.

Paths through the PG represent paths through the topology that BGP announcements may use to ensure policy-compliance. For example, in the concrete PG, all messages from N1 are not blocked at spine routers S1 and S2. Paths for a destination learned through N1 will end in an accepting state for the first automaton (*e.g.*, node (T1,1,0)). Similarly, paths for a destination learned through N2 will end in an accepting state for the second automaton.

Interestingly, the concrete and abstract PGs have a similar structure, which leads to the following observation:

Lemma 5.1. *If we have a graph homomorphism $f : G \to G^A$, concrete product graph $PG = (G', start, \mathrm{rank})$ and abstract product graph $PG^A = (G'^A, start^A, \mathrm{rank}^A)$, then there is a homomorphism $f_{pg} : G' \to G'^A$ where:*

$$\begin{aligned} f_{pg}(start) &= start^A \\ f_{pg}((l, q_1, \ldots, q_n)) &= (f(l), q_1, \ldots, q_n) \end{aligned}$$

As shown in §7, our generation strategy commutes with template instantiation, meaning that we obtain the same results if we instantiate the abstraction early, or if we defer the instantiation until after template generation.

6. Fault-Tolerance Analysis

The possibility of network failures exacerbates the difficulty of constructing correct configurations. Link failures in networks occur frequently; it is not uncommon for a large network to experience dozens of failures in any given day [15]. However, existing tools [5, 33] reason about fault-tolerance only for concrete topologies. Propane/AT provides stronger guarantees: all possible concrete instantiations of an abstract topology satisfy the fault-tolerance policy.

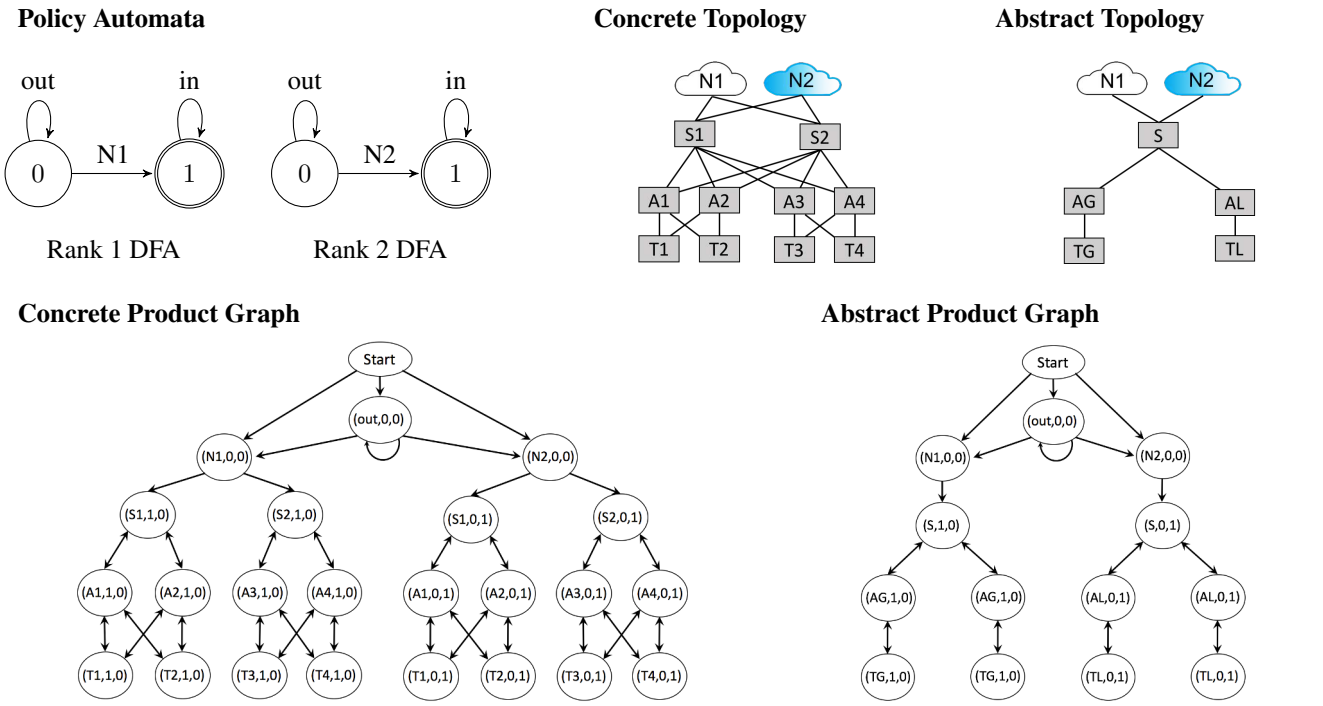


Figure 6: Product Graph construction for policy `true => exit(N1 >> N2)`.

We frame satisfying the fault-tolerance requirements as an analysis problem over the structure of the PG. In particular, we develop an analysis that uses information embedded in the abstract topology to infer bounds on the number of edge-disjoint paths between pairs of concrete nodes.

For each node in the abstract PG, the idea is to infer facts learned about the number of edge-disjoint paths to groups of concrete routers in the node's role. More specifically, we maintain fact of the form:

$$L_{X_1}, \ldots, L_{X_n}(j, k)$$

where each label $L \in \{S, A\}$ is either S, which stands for "some" or is A, which stands for "all". There is one label for each pod in the abstraction pod hierarchy under which the abstract node appears. For a given node, L_{X_1} corresponds to the outermost pod, $L_{X_{n-1}}$ corresponds to the innermost pod, and L_{X_n} to the node itself. Semantically, $L_{X_1}, \ldots, L_{X_n}(j, k)$ means that starting from some concrete source node, for some/all pods $X_1, \ldots, X_{n-1}$ and for some/all groups of nodes in the role X_n of size j, there are k paths to each such that all $j * k$ paths are edge-disjoint.

For example, an inference of the form $A_Q A_{TL}(2, 3)$ states that, from the given source location, for *all* pods Q, and *all* groups of 2 nodes in the TL role, there are 6 disjoint paths to the group—3 for each of the 2 nodes.

6.1 Inference Rules

Figure 7 displays the collection of rules used to infer facts about disjoint paths. Each rule is read from bottom to top. The label on the bottom left is a known fact. We use L to represent a rule that is parametric over the label (S or A). Labels on other nodes correspond to facts learned after applying the rule. The box shows the conditions that must be valid, given the abstraction constraints, for the rule to apply.

Some of the inference rules (*e.g.*, I-out2 and I-mesh2) try to learn about the largest number of disjoint paths to any single node in an abstract role, while others (*e.g.*, I-out1 and I-mesh1) try to learn about the largest reachable group in a particular role with at least one disjoint path to each node in that group. Both kinds of rules are useful.

The first rule, I-out1 applies to a learned fact of the form $L_m(j, k)$ where the number of outgoing edges from any concrete node in the m role is greater than 0. In the worst case, the largest group of concrete nodes we could hope to reach at the n role would be e_1 since all j nodes at the bottom may have outgoing edges to the same concrete nodes at the top. Furthermore, the total number of disjoint paths to the j nodes at the bottom is equal to $j * k$. Since extending the existing disjoint paths with disjoint edges keeps the paths disjoint, and since we cannot exceed the current number of disjoint paths to the concrete nodes in role m on the bottom, the largest reachable group for the role n on the top will be $min(j * k, e_1)$. We conservatively use 1 for the number of disjoint paths to each node in role n, since when n is very large, all reachable nodes in role m might only have a single edge to completely different nodes in n.

Consider rule I-out2, and consider any node in the role n. There are e_2 incoming edges to that node. Due to the

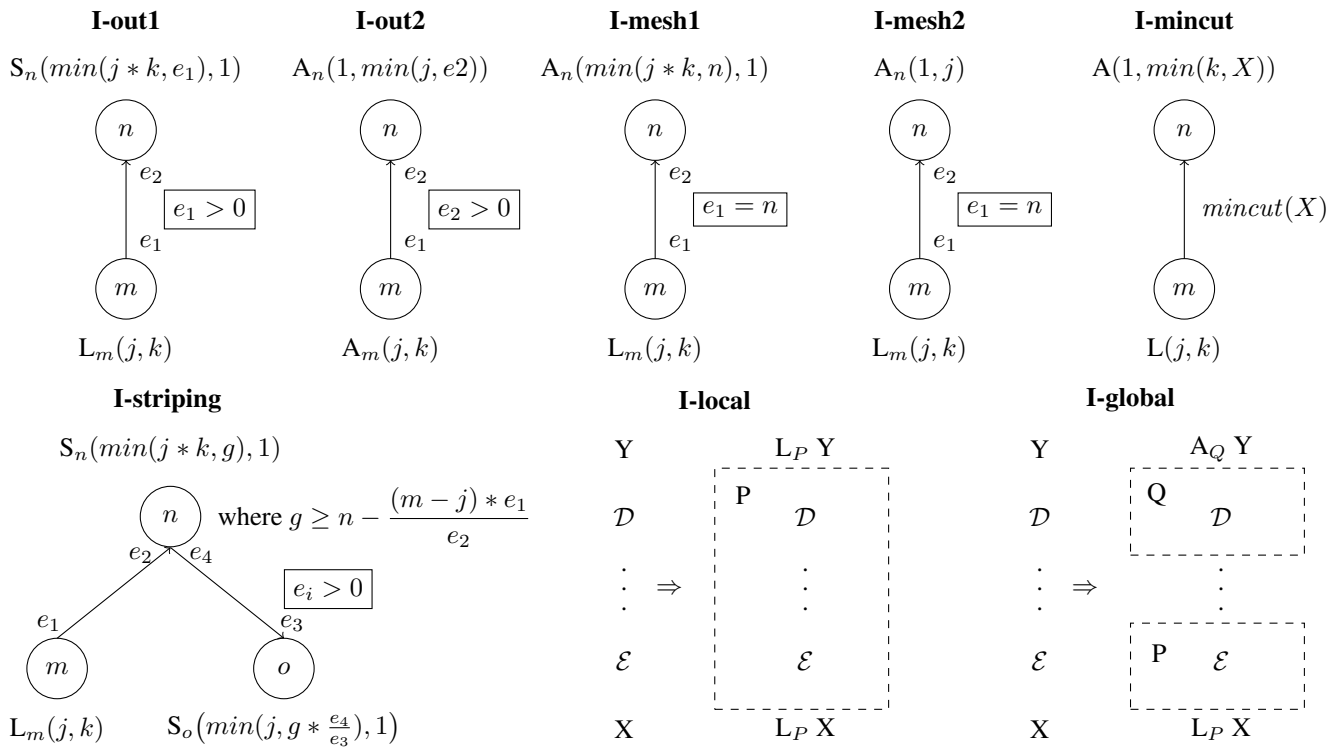


Figure 7: Abstract k-disjoint path analysis inference rules.

fact that $A_m(j,k)$, we know that at least j of those e_2 edges are connected to nodes with disjoint paths from the origin. Hence we infer $A_n(1, min(j, e_2))$.

The two rules I-mesh1 and I-mesh2 handle the case where there is a full mesh between the two roles. This happens when the number of outgoing edges (e_1) from nodes in role m equals the number of nodes on top (n). I-mesh1 says that we can find disjoint paths to each node in the top role restricted to the number of disjoint paths we started with. I-mesh2 uses the fact that each node in the top role is connected to each node in the bottom role to infer that there can be j disjoint paths to any single node in the top role.

The annotation $mincut(X)$ appearing on an edge is an assertion about the fault tolerance between nodes in two different roles. The rule I-mincut uses such assertions. To each node in role n, from a node in role m, we can construct at least the minimum of X and k disjoint paths.

The rule I-striping is the most complicated case. It starts with the invariant $L_m(j,k)$ at role m and can be applied if each edge multiplicity $e_i > 0$ is valid given the constraints. The first inference for role n tries to find the largest reachable group with disjoint paths to each. The idea is similar to the rule I-out1, but is able to use the fact that $e_2 > 0$ to learn more about the structure of the concrete topology. In particular it uses the following inequality, where g represents the size of the group for the role n:

$$(m-j)*e_1 \geq (n-g)*e_2$$

The remaining nodes $(m-j)$ that are not part of the reachable group in the bottom role, each have e_1 outgoing edges and must be able to at least "fill" the incoming edges for the remaining nodes not in the reachable group at the top $(n-g)$, which each have e_2 incoming edges. Solving the inequality gives the lower bound for g used in Figure 7.

The second part of the rule uses a similar idea to reason about the overlap between roles m and o with respect to role n. This rule is particularly useful for data center topologies where routers in one tier of the data center often have a uniform striping pattern with another tier.

Finally, rules I-local and I-global reason across pod hierarchies. I-local says that if there is an inference from X to Y, then the derivation can be used inside pod P by leaving the P-label unchanged. I-global says that when the edge goes across pods, we can infer the fact for all pods Q since the multiplicities apply uniformly for each pod.

6.2 Inference Algorithm

The abstract disjoint path analysis starts from a fixed source location src and repeatedly tries to apply every inference rule from Figure 7 until it reaches a fixed point. The algorithm applies an inference rule when the rule's condition is valid given the abstract topology constraints. Because the inference rules may continue to yield larger and larger symbolic expressions, we make the following observation to ensure termination: for any invariant learned of the form $L(j,k)$, it is sound to instead infer $L(j',k')$ if $j' \leq j$ and

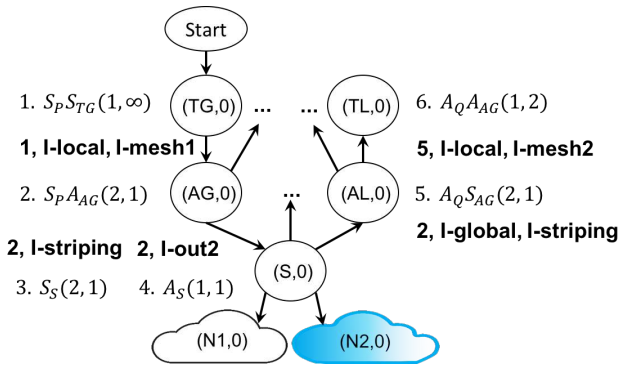


Figure 8: Abstract disjoint path analysis for global prefixes.

$k' \leq k$. Therefore, for each inference $L(j,k)$ we minimize the symbolic expressions for j and k subject to the topology constraints using the optimizing SMT solver νZ [6].

At a higher level, what is happening is that each inference rule is attempting to learn the maximum fault tolerance information possible as a function of the symbolic inputs. The νZ [6] solver will then minimize this maximum by accounting for all possible topologies that meet the abstraction. Facts learned with $j = 0$ or $k = 0$ are discarded.

Recall the policy for global prefixes in the data center.

```
$GP => end(TG) ∩
       novalley({TG,TL},{AG,AL},{S}) ∩
       !(enter(Peer) & exit(Peer))
```

Figure 8 shows part of the abstract PG representation for this routing policy. The inference algorithm starts from the node $(TG, 0, 0)$ with the initial fact $S_P S_{TG}(1, \infty)$ (*i.e.*, no restriction on the number of disjoint paths initially). The first step applies each inference rule to this initial fact. The algorithm uses rules l-mesh1 and l-local to reason about connectivity within a single pod for the TG and AG roles. It makes a call to νZ to minimize the expression $min(\infty, AG)$, which results in 2. Therefore, the algorithm learns a new invariant of the form $S_P A_{AG}(2, 1)$ for node $(AG, 0)$ to indicate that in some pod P, any group of 2 nodes is reachable. The algorithm will then eventually apply l-out2 to learn that any single spine node is reachable at $(S, 0)$. It will also apply l-striping to determine that there is some group of at least 2 spine routers reachable at $(S, 0)$ and that there is some group of at least 2 nodes reachable in the AL role in state $(AL, 0)$.

Note that, because each inference rule only applies to directed edges in the PG, the algorithm cannot make any inferences about connectivity from the AL role to the S role since there is no directed edge from AL to S. This restriction ensures that the analysis remains policy-sensitive.

The next step is to use l-mesh2 together with l-local to infer that any single node in the TL role for any pod Q is reachable via at least 2 disjoint paths. This process will continue until a fixed point is reached.

The algorithm could infer that there is at least 1 disjoint path to any spine router, and at least 2 disjoint paths to any TL router. In this case, the analysis is precise. There exists a concrete network, namely the data center from Figure 2, where a single failure can disconnect a global ToR from a spine router due to the valley-free constraint.

7. Template Generation

At a high level, the translation from the PG representation to per-device templates that run the distributed BGP protocol involves the following steps: (1) Every BGP message is tagged to record the state of the product graph as messages are passed between routers. The tags allow BGP to only search for valid paths by dropping messages that do not correspond to any edge in the PG. (2) To ensure that BGP always finds the *best* paths in the network, we must rank route advertisements for each router locally such that, under this ranking function, the network as a whole satisfies the Propane/AT policy's end-to-end network preferences.

We introduce a simple, vendor-independent BGP configuration language called mBGP and describe a compilation function that uses the inferred preference ordering to generate concrete configurations from a concrete topology, or templates from an abstract topology. Given a concrete network, templates can be instantiated by replacing instances of abstract neighbors with the union of all concrete neighbors under the inverse homomorphism, and by replacing prefix template variables with separate entries for each concrete prefix provided by a context. We call this process of transforming templates into configurations *concretization*. We prove that concretization and compilation commute. Moreover, by modifying compilation slightly we can further ensure that every configuration depends only on its immediate neighbors. This guarantees that any change made to the concrete topology will result in a minimal number of changes to the configurations. We now look at each of these steps in turn.

Tagging and Filtering The BGP routing protocol allows *community tags*—32-bit integer tags that can be arbitrarily attached to, or removed from, routing advertisements. Community tags serve as a simple form of *history*, and operators routinely use such tags to implement policy. For example, operators might tag advertisements at certain entry points and then block the export of tagged advertisements to prevent their network from becoming a transit point.

Template generation uses community tags to record the state of the automata from the Product Graph in every BGP announcement and updates these tags whenever messages are passed between routers. This mechanism ensures that BGP only considers paths that are allowed by the Propane/AT policy (*i.e.*, paths in the PG). Routers will allow advertisements that correspond to an edge in the PG and will block any advertisements that do not.

mBGP Syntax

d	$\in$	*Integers*	
c	$\in$	*Communities*	
l	$\in$	*Topology Locations*	
t	::=	$\$x \mid d.d.d.d/[d..d]$	*predicate*
ns	::=	$\{l_1, \ldots, l_k\}$	*peers*
ma	::=	$d : (ns_1, c_1) \to (ns_2, c_2)$	*match action*
pc	::=	$ma_1, \ldots, ma_k$	*predicate config*
rc	::=	$t_1 \to pc_1, \ldots, t_k \to pc_k$	*router config*
$mbgp$	::=	$l_1 \to rc_1, \ldots, l_k \to rc_k$	*mbgp policy*

Compilation to mBGP

$$\begin{aligned}
&\text{compile}_{\text{mBGP}}([(t_1, PG_1, pref_1), \ldots, (t_k, PG_k, pref_k)], G) = \\
&\quad [\, l \to rc \mid l \in \text{internal}(G.V),\ rc = \text{append}_i \\
&\qquad [\, t_i \to [\, ma \mid \\
&\qquad\quad m \leftarrow (l, q_m) \in PG_i, \\
&\qquad\quad pin \leftarrow \text{adjIn}(PG_i, m), \\
&\qquad\quad (in, q_n) \leftarrow \{(bs, q_n) \mid bs = \{b \mid (b, q_n) \in pin\}, bs \neq \emptyset\}, \\
&\qquad\quad out \leftarrow \{c \mid (c, _) \in \text{adjOut}(PG_i, m)\}, \\
&\qquad\quad ma = \text{pref}_i(m) : (in, q_n) \to (out, q_m)\,]\,]\,] \\
&\\
&\text{compile}(p_1, \ldots, p_k, G) = \\
&\quad \text{compile}_{\text{mBGP}}([\text{compile}_{\text{PG}}(p_1, G), \ldots, \text{compile}_{\text{PG}}(p_k, G)], G)
\end{aligned}$$

Figure 9: mBGP syntax (left), and compilation from product graphs (right).

For example, in the concrete PG in Figure 6, router S1 appears in two PG nodes: $(S1, 1, 0)$ and $(S1, 0, 1)$. These nodes have two peers that can send them messages: $(N1, 0, 0)$ and $(N2, 0, 0)$. Therefore, S1 will allow advertisements from both peer $N1$ and peer $N2$. If S1 uses the path advertised by neighbor N1, then it will add the tag $(1, 0)$ to the advertisement before sending this to its neighbors A1 and A2. If S1 uses a path advertised from neighbor N2, then it will add the tag $(0, 1)$ instead. Similarly, router A1 appears in two PG nodes. It will admit advertisements that have the $(1, 0)$ or $(0, 1)$ tag attached (and drop all other advertisements). In either case, it will not modify the tag before exporting the route to its neighbors.

Preference Search Tagging and filtering based on the PG restricts the possible paths to those that are allowed by the policy. However, finding *some* path to the destination is not enough. We must ensure that each router uses its *best* available path according to the policy. Further, each router must continue to use its best available path as elements of the network fail. In the example from Figure 6, the policy was `true => exit(N1>>N2)`, which indicates that paths through N1 should be used whenever possible, and paths through N2 should only be used as a backup. If router S1 allows advertisements from both N1 and N2, but does not prefer one advertisement over another, it might end up choosing to use a path through N2 even though a more preferred path through N1 exists and is available[4]. On the other hand, if the S1-N1 link fails, then the best available path is through N2.

To enforce correct path preferences, we use the BGP local-preference attribute, which allows routers to prefer certain routes (e.g., those from a particular neighbor or with a tag) over other routes. The challenge is to find a collection of device-local preferences that correctly enforce the policy's network-wide preferences in the face of any set of failures.

The idea is to search for such a device-local preference function for each router that totally orders advertisements from different neighbors (possibly with different tags). For example, to satisfy the policy that leaving through N1 is preferred to leaving through N2, S1 should prefer a message it hears from N1 over N2. Similarly, A1 should prefer routes tagged with $(1, 0)$ over those with $(0, 1)$. Intuitively, S1 should prefer a message from N1 because it results in an accepting state for automata 1, which indicates a better path. Further, it can result in downstream routers such as A1 using a path that is accepting for automata 1 as well.

In general, because BGP is distributed, each router does not have a view of the entire network when choosing which path to use, and finding a collection of preferences to ensure correct end-to-end behavior for all failures is a hard problem. Propane introduced a conservative search strategy for determining route preferences that works well in practice. To make it work for abstract topologies, we modify the search based on the following observation about the PG structure:[5]

Definition 7.1. *Let $m \geq_{rank} n$ be a relation over PG vertices that holds iff $\text{topo}(m) = \text{topo}(n)$ and either $\min(\text{rank}(m)) \geq \min(\text{rank}(n))$ or $\text{rank}(n) = \emptyset$.*

Intuitively $m \geq_{rank} n$ means that paths ending at node n have lower automata rank and are thus better than paths ending at m. The PG can be viewed as a labeled transition system by pushing the location from each directed edge's target node onto the edge. That is, $m \xrightarrow{l} n$ if there is an edge (m, n) in the PG and $\text{topo}(n) = l$. For example, in the concrete PG we have the transition $(S1, 1, 0) \xrightarrow{A2} (A2, 1, 0)$.

Definition 7.2. *We write $m \leq n$ if the subgraph reachable from m and n respectively form a simulation relation with respect to $\geq_{rank}$. More specifically, we say that $m \leq n$ if $n \geq_{rank} m$ and for every transition $n \xrightarrow{l} n'$ from PG node n there exists a transition $m \xrightarrow{l} m'$ from m and $m' \leq n'$.*

If $m \leq n$, then advertisements received for node m can safely be preferred over those received for node n after accounting for the network-wide impact of the choice. For ex-

[4] Ties between equally preferred paths can be broken nondeterministically.

[5] Recall that $\text{rank}(m)$ is a set of priorities—those of the automata whose final states include m. For instance, if $\text{rank}(m)$ is {1,3} then automata 1 and 3 have m as a final state. Moreover, recall that the policy expressed by automata i is preferred to the policy expressed by automata j if $i < j$.

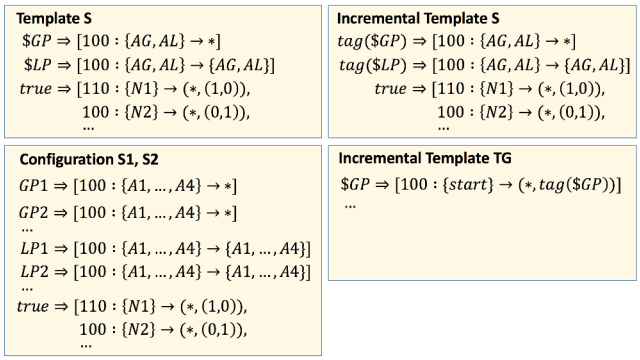


Figure 10: Spine template and concrete configurations (left), and evolution-friendly templates (right).

ample, in Figure 6, S1 can receive advertisements in two different contexts $(S1, 1, 0)$ and $(S1, 0, 1)$ and must choose between messages received in these different contexts. Advertisements are preferred in state $(S1, 1, 0)$ because they result in a better path for S1 ($(S1, 0, 1) \geq_{rank} (S1, 1, 0)$) and downstream routers such as A1 will also obtain paths no worse than if S1 had chosen $(S1, 1, 0)$. The policy is guaranteed to be safe from failures because if a link fails in the topology, then $m \leq n$ will still hold since any transition that becomes unusable for m also becomes unusable for n. That is $m \leq n$ before the failure implies $m \leq n$ after the failure.

For each router, its corresponding PG nodes are then sorted according to the $\leq$ operator. If the operator defines a total order on PG nodes, then the compiler can simply prefer advertisements from peers of node m over those of n whenever $m \leq n$. However, if $\leq$ does not form a total order, then the policy is rejected as being potentially unsafe under some failure conditions.

For example, in Figure 6 the inequality $(S1, 0, 1) \leq (S1, 1, 0)$ holds since nodes on the left side of the PG can always match transitions made on the right hand side with respect to the $\geq_{rank}$ relation. This relation does not hold the other way around since $(S1, 0, 1) \not\geq_{rank} (S1, 1, 0)$. Therefore, advertisements received at $S1$ with tag $(1, 0)$ must be preferred to those received at $S1$ with tag $(0, 1)$. Notice that in the abstract PG $(S, 0, 1) \leq (S, 1, 0)$ also holds. This leads to the following observation:

Lemma 7.1. *$m \leq n$ in the concrete PG iff $f_{pg}(m) \leq f_{pg}(n)$ in the abstract PG.*

Lemma 7.1 tells us that inferring preferences for the abstract PG before template instantiation is equivalent to inferring preferences for an already-instantiated concrete PG. A proof appears in the technical appendix.

mBGP To characterize compilation we first introduce a simple, vendor independent configuration language for the BGP protocol called mBGP. The syntax for mBGP is shown in Figure 9 (left). An mBGP policy consists of a sequence of router configurations (one for each internal topology location l). A router configuration is an ordered sequence of pairs, where is pair contains a predicate describing the traffic, and a predicate configuration. A predicate is either a template variable $x or a prefix. A predicate configuration is a collection of match action statements, where each match action indicates that the router will match advertisements from any of a set of peers ns_1 with a particular community tag c_1 with local preference d, before exporting the route to another set of peers ns_2 with a new community tag c_2.

Compilation Figure 9 (right) defines compilation from Propane/AT to mBGP. It proceeds by compiling constraints p_i in the original Propane/AT policy to a tuple of: the predicate t_i, the product graph PG_i, and the preference function $pref_i$. These tuples are passed to the $\text{compile}_{\text{mBGP}}$ function, along with the network topology G. For each internal router in the topology l, and each predicate t_i in the Propane/AT policy, compilation goes through each node m for l in PG_i, and groups the inbound neighbors of m by tag (q_n) into sets (in). It allows imports from these neighbors before exporting to the outbound neighbors of m. The local preference for these imports is given by $\text{pref}_i(m)$, which represents an integer based on the total ordering of $(\leq)$. We build configurations using list-comprehension notation. For instance, $[\, l \to rc \mid l \in V, p(l, rc)\,]$ denotes the mBGP policy $l_1 \to rc_1, ..., l_k \to rc_k$ where each rc_i satisfies $p(l_i, rc_i)$. We use append to denote sequence concatenation.

Figure 10 (left) shows part of the generated mBGP configuration for spine routers for both the concrete and abstract policies. For brevity, we using the symbol $(*)$ to denote the set of all neighbors and omit tags when irrelevant. For prefix `true`, the spine routers will match advertisements from peer N1 and N2. The match for N1 is preferred since it has a higher BGP local-preference attribute (110). If an advertisement from N1 is chosen, the spine attaches the community tag $(1, 0)$ before sending the route to all its peers $(*)$. If an advertisement is only available from the backup N2, then it attaches the tag $(0, 1)$ instead. The template configuration matches any global prefix $GP from any internal peer and re-advertises the route to all its peers. For any local prefix, it will allow an advertisement from any internal peer, and re-advertise the route to only other internal peers. The concrete configurations for S1 and S2 obtained from compilation for the concrete PG from Figure 6 have a similar structure for each local and global prefix where local routes are reflected downward, while global routes are advertised to all peers.

Concretization The similar structure between the spine template and concrete configurations is not a coincidence. We formalize this observation by defining two concretization functions (con), one for Propane/AT policies and another for mBGP policies. Concretization takes a context $\Gamma : Var \to 2^{Prefix \times V}$ that maps each template variable to a set of pairs of a concrete prefix and topology location where the prefix is owned. Both concretization functions traverse the policy and substitute instances of a topology location l in the template policy with the set of all concrete

locations that map to l, given by the inverse homomorphism $f^{-1}(l) = \{l' \mid f(l') = l\}$. Additionally, whenever a pair $(pfx, l) \in \Gamma(x)$, then a new entry is added to the concretized policy where pfx replaces $x and adds the constraint that traffic ends at l (`end(l)`). For example, the spine template in Figure 10, is obtained by substituting $\{A1, A2\}$ for AG and $\{A3, A4\}$ for AL and by also replacing the entry for $GP with entries for GP1 and GP2 given by the context.

Our main theoretical result is that the compilation and concretization functions commute:

Theorem 7.2. *For any context Γ, topologies G and G^A, homomorphism $f : G \to G^A$, and policy pol,*

$$con(compile(pol, G^A), \Gamma, f, G) = compile(con(pol, \Gamma, f), G)$$

Full definitions of concretization as well as the proof of Theorem 7.2 are included in the technical appendix.

Incrementality Suppose an operator wants to expand the concrete data center from Figure 1 by adding an additional ToR router to the TG role. Per the network routing policy, the new router will advertise any owned prefixes provided by looking up $GP in Γ. Because the new topology matches the abstraction, the compiled templates will remain the same. However, in the spine configurations, the match on the global prefix template variable $GP must be expanded when concretizing the template to include the new prefixes added by the ToR. Hence, this small change to the topology results in a change to every single spine configuration.

More generally, each configuration template depends on two things: the routing policy and the abstract topology. If the policy remains fixed and a change to the concrete topology preserves the topology abstraction, then the generated templates will not change. Further, each template has policy only in terms of its immediate neighbors. Because abstract neighbors are substituted for concrete neighbors during concretization, it would seem as though the generated configurations will also only depend on their concrete neighbors. However, prefix template variables allow for the possibility of introducing new prefixes in the context Γ after a change. For example, when adding a new ToR router with its own unique prefix, the spine configurations would need to know about this new prefix. In fact, the only way in which the concrete configurations can depend on anything non-local is when instantiating prefix template variables.

To prevent the non-local changes induced by template variables, we modify compilation in the following ways. First, we associate a new unique community tag for each template variable (*e.g.*, $GP), and add this tag where the route is originated (*e.g.*, role TG). Then, template variable tests elsewhere in the policy are replaced with a new test on this tag. Finally, during template instantiation the tags are left unmodified. Figure 10 (right) shows the spine and ToR templates after this transformation. Routers in the TG role will originate ($\{\text{start}\}$) global prefixes and tag them with a unique tag, while routers in the spine role match the tag.

	Fixed	**Reachability**		**K-paths**	
		Some Pairs	All Pairs	Some Pairs	All Pairs
Tree-based topologies, valley-free routing					
Fat tree [3]	–	✓	✓	✓	✓
Facebook [4]	–	✓	✓	✓	✓
F10 [25]	–	✓	✓	✓	✓
VL2 [16]	–	✓	✓	✓	✓
All topologies, shortest-path routing					
Fat tree [3]	–	✓	✓	✓	C
Facebook [4]	–	✓	✓	✓	✓
F10 [25]	–	✓	✓	✓	C
VL2 [16]	–	✓	✓	✓	C
BCube [18]	k	✓	✓	C	C
DCell [17]	k	✓	✓	C	C
Butterfly [23]	n	✓	✓	✓	✓
Hypercube	N	✓	✓	✓	✓
HyperX [2]	L	✓	✓	✓	✓

Figure 11: Expressiveness and precision of Propane/AT.

8. Implementation

The Propane/AT compiler generates configurations for Cisco and Quagga [30] routers. The fault-tolerance analysis uses νZ [6] to both test validity and minimize variables subject to the topology constraints. Since the analysis typically calls the SMT solver many times with relatively small optimization problems, we use a timeout of 200ms.

Although the disjoint path analysis takes place over the PG, each application of the inference rules from Figure 7 depends on the topology locations, but not the automata states, and can be reused across multiple PG nodes with the same topology location. Therefore, we lazily apply the rules and cache the satisfiability and minimization calls to νZ after their first use. Furthermore, the cached results are shared across different prefixes, each of which may have a unique PG representation.

9. Evaluation

9.1 Expressiveness and Precision

We evaluate the expressiveness of Propane/AT's topology abstractions and the precision of its fault-tolerance analysis on a range of network topologies found in production networks and in the networking literature. We characterize expressiveness by checking if the abstractions allow the topologies to evolve arbitrarily or certain aspects must be fixed (*i.e.*, cannot be symbolic). We measure precision by checking if we find a tight lower-bound on fault-tolerance (*i.e.*, there is a concrete network with that fault-tolerance).

The top part of Figure 11 shows the results for common data center networks: tree-based topologies coupled with valley-free routing. We consider four variants of tree topolo-

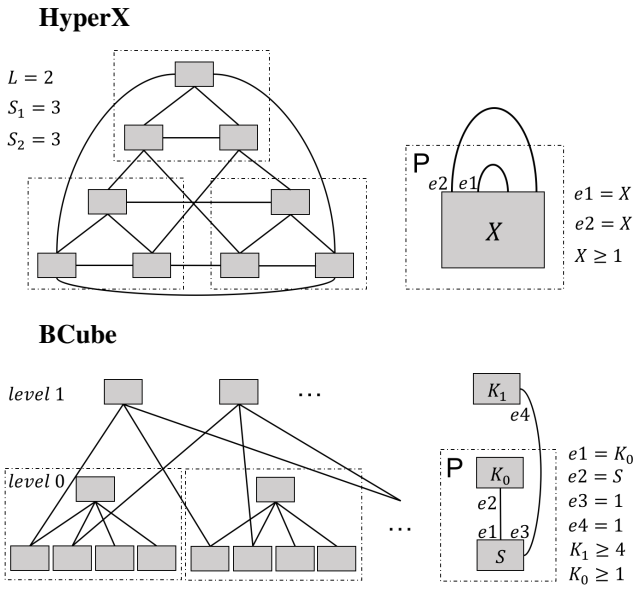


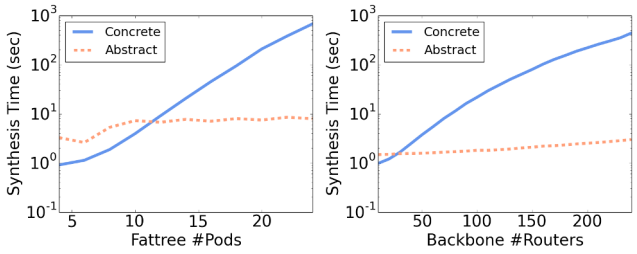
Figure 12: Example abstractions for HyperX and BCube.

gies: a standard fat tree [3], the Facebook fat tree [4], the F10 fault-tolerant fat tree [25], and VL2 [16]. These variants differ in the number of tiers and the connectivity pattern between roles. For each, we use a tiered abstraction similar to that in our example (§4) and parameterize over the number of pods, which can be scaled for expansion. We report precision of both analyzing reachability and disjoint paths, and we report if Propane/AT is precise for all pairs of abstract nodes or only some of them. We record a check when the analysis is precise and a C when the analysis is conservative.

Our results are encouraging for these settings. Our abstractions are perfectly expressive for tree-based topologies—we did not have to fix any aspect of their structure—and the analysis is precise in all cases.

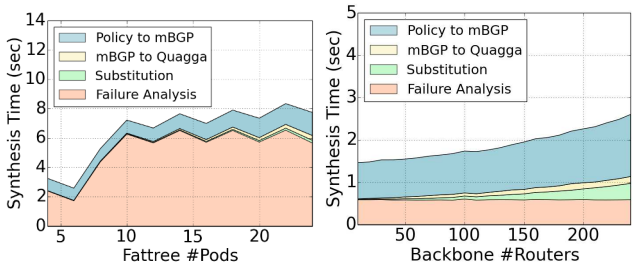
Recursive Topologies These topologies include BCube [18] and DCell [17]. Each topology includes a recursion depth parameter (k), which we fixed while abstracting them. For a recursive topology with depth k, we model it as an abstract topology consisting of a pod to represent all depth $k-1$ subcomponents. This allows for safe expansion within a subcomponent, but does not allow changing the recursion depth dynamically. For BCube, we model each tier of the data center as a separate role. Figure 12 shows an example of a BCube abstraction for $k = 1$.

Hypercube Topologies Hypercube variants can be used as an alternative to Clos-style topologies for networks with port density routers. The HyperX [2] topology generalizes the hypercube and butterfly topologies and includes parameters L for the lattice dimension of the network, and S_i for the node multiplicity of each dimension i. For a fixed number of dimensions L, we abstract each full mesh of S_L nodes into its own abstract node. Nodes in dimension S_{x-1} are



(a) Data center **(b)** Backbone

Figure 13: Concrete vs. Abstract Synthesis Time.



(a) Data center **(b)** Backbone

Figure 14: Abstract Synthesis Time by Phase.

abstracted using pods of abstract nodes from dimension S_x. Figure 12 shows an example for $L = 2$.

Results The bottom part of Figure 11 shows the results for all types of topologies with shortest-path routing. (Valley-free routing is not meaningful for non-tree-based topologies.) For all tree-based topologies, the analysis is precise for reachability, but for three of them, it does not compute a tight bound for disjoint paths for all router pairs. Specifically, it underestimates ToR-to-spine paths; it fails to account for some circuitous paths that traverse another spine because it could not disambiguate two concrete spines that map to the same abstract role. For instance, for the fat tree topology [3], it only finds 1 path between any ToR and any Spine when there should always be at least two. However, in this case the analysis computes the correct worst case connectivity between any source ToR and any other destination aggregation or ToR router. A similar pattern occurs with other tree-based topologies. For both recursive topologies, the analysis can only accurately determine reachability.

9.2 Synthesis time

We evaluate generation time in Propane/AT both with and without abstraction using routing policy for backbone and data center networks inspired by configurations obtained from a large cloud provider. For both types networks, we fix the routing policy and scale the size of the topology.

Topologies Routers in the data centers run BGP using unique AS numbers and connect to multiple external neighbors. The routers aggregate some prefix blocks when announcing them to external neighbors, and keep some pre-

fixes internal. The data center prefers that traffic leave through certain neighbors over others and should not transit traffic between neighbors. The policy also prevents routers from using external neighbors to reach "private" destinations (i.e., those in the IP address space reserved for private use). We use a fat tree [3] and scale it by increasing the number of pods. The abstract topology uses one abstract node for each tier with additional nodes for local and global ToRs.

The backbone policy classifies neighbors into several categories based on commercial relationship [13] and prefers paths through them in order. Like the data center, it blocks private destinations from neighbors, drops transit traffic between certain pairs of neighbors, and aggregates internal prefixes at the network border. We scale the backbone network from 10 to 240 routers. We split it into two parts: border routers that connect to external neighbors and an internal core. We use one abstract node for the border routers and one for the network core with mincut annotations both within the core and between the core and border roles. For neighbors, there is one abstract role per commercial category.

Results Figure 13 shows total configuration generation time for Propane/AT vs the concrete network synthesis tool Propane. All experiments were run on an 8 core, 2.4 GHz Intel i7 processor machine running Mac with 16GB of Ram.

For both networks, the abstract synthesis is slightly slower than concrete synthesis for small topologies due to the overhead of the fault-tolerance analysis. However, as the topology size increases, abstract synthesis becomes orders of magnitude faster. In all cases for both networks, it takes less than 10 seconds to complete.

Figure 14 shows the relative time taken by each phase of Propane/AT. The fault-tolerance analysis takes the most time, but that does not depend on the number of concrete nodes in the network, and thus is largely a fixed cost. In particular, the number of calls to νZ remains constant across topology size. The seesaw behavior for the data center networks results from differences in time taken by νZ to minimize similar constraints with different values.

9.3 Incrementality

Propane/AT's compilation strategy guarantees that network evolution requires configuration changes only for nodes that acquire or lose a neighbor. We experimentally confirmed that our implementation provides this guarantee. For the networks we studied above, we made a range of changes, including adding and removing routers and pods and changing prefixes that routers originate. In each case, we found the guarantee to hold. In contrast, all router configurations were modified with Propane because it heavily uses prefix lists which are sensitive to such changes. While Propane may be made friendlier to network evolution, its fundamental limitation will remain because it does not understand roles and the network's structure that Propane/AT leverages.

10. Related Work

Network Synthesis Many recent systems allow operators to specify their policies at a high level of abstraction. These can be classified into two classes. The first class targets networks based on SDN (software-defined networking) and generates dataplane rules [31, 32, 34]. The second class, to which our work belongs, targets conventional networks and generates control-plane router configurations [5, 8, 28].

While we borrow much from existing configuration synthesis work, especially Propane [5], our goal is to generate role templates from abstract topologies, rather than router configurations from concrete topologies. This goal aligns better with operators' mental models and tools and permits network evolution with minimal disruption. In the process, we develop new abstractions for network topologies, compilation algorithms that generate templates, and analysis techniques that operate over classes of networks.

Topology Design Network topology design, especially for data centers, is another active research area, with researchers exploring designs with different properties [2–4, 16–18, 23, 25]. We do not develop new designs but develop abstractions that can capture these designs (and their evolution).

Network Verification A complementary approach to reducing configuration errors is to analyze the forwarding rules or the configuration of a network to ensure the absence of (a class of) bugs [9–12, 14, 20, 22, 27, 36]. Our focus, in contrast, is on a correct-by-construction approach. It has the downside that operators must express policies in a new language, but it saves them from the challenging and time-consuming task of manual configuration generation.

11. Conclusions

To help configure large networks correctly, we develop Propane/AT, the first system that can generate role templates from high-level specifications of network topology and policy. Propane/AT is based on new abstractions for capturing network topologies, and their evolution, and algorithms to analyze the combined impact of topology and routing policy on the network's fault tolerance. Our analysis operates entirely in the abstract domain and guarantees correctness for all concrete instantiations of the topology. Experiments with many types of real-world topologies and policies show that our abstractions and analysis are effective and that, for large networks, configuration synthesis is two orders of magnitude faster than systems that operate over concrete topologies.

Acknowledgments

We thank George Chen and Lihua Yuan, who pointed out the importance of working with abstract topologies. We also thank the PLDI reviewers, and especially our shepherd Alexandra Silva for comments on the paper. This work is supported in part by the National Science Foundation award

CNS-1161595, Cisco award 677002, and an Okawa Foundation Research Grant.

References

[1] News and press — BGPMon. http://www.bgpmon.net/news-and-events/.

[2] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber. HyperX: Topology, routing, and packaging of efficient large-scale networks. In *SC*, November 2009.

[3] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *SIGCOMM*, August 2008.

[4] A. Andreyev. Introducing data center fabric, the next-generation facebook data center network. https://code.facebook.com/posts/360346274145943/, November 2014.

[5] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *SIGCOMM*, 2016.

[6] N. Bjørner, A.-D. Phan, and L. Fleckenstein. *νZ - An Optimizing SMT Solver*, pages 194–199. 2015.

[7] M. Casado, T. Koponen, R. Ramanathan, and S. Shenker. Virtualizing the network forwarding plane. In *PRESTO*, pages 8:1–8:6, 2010.

[8] A. El-Hassany, P. Tsankov, L. Vanbever, and M. Vechev. Network-wide configuration synthesis. https://arxiv.org/abs/1611.02537, November 2016.

[9] S. K. Fayaz, T. Sharma, A. Fogel, R. Mahajan, T. Millstein, V. Sekar, and G. Varghese. Efficient network reachability analysis using a succinct control plane representation. In *OSDI*, 2016.

[10] N. Feamster and H. Balakrishnan. Detecting BGP configuration faults with static analysis. In *NSDI*, May 2005.

[11] A. Fogel, S. Fung, L. Pedrosa, M. Walraed-Sullivan, R. Govindan, R. Mahajan, and T. Millstein. A general approach to network configuration analysis. In *NSDI*, March 2015.

[12] N. Foster, D. Kozen, M. Milano, A. Silva, and L. Thompson. A coalgebraic decision procedure for netkat. In *POPL*, pages 343–355, January 2015.

[13] L. Gao and J. Rexford. Stable internet routing without global coordination. In *SIGMETRICS*, pages 307–317, June 2000.

[14] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *SIGCOMM*, August 2016.

[15] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *SIGCOMM*, August 2011.

[16] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A scalable and flexible data center network. In *SIGCOMM*, pages 51–62, October 2009.

[17] C. Guo, H. Wu, K. Tan, L. Shi, Y. Zhang, and S. Lu. Dcell: A scalable and fault-tolerant network structure for data centers. In *SIGCOMM*, pages 75–86, October 2008.

[18] C. Guo, G. Lu, D. Li, H. Wu, X. Zhang, Y. Shi, C. Tian, Y. Zhang, and S. Lu. BCube: A high performance, server-centric network architecture for modular data centers. In *SIGCOMM*, pages 63–74, October 2009.

[19] hatch. Hatch – create and share configurations. http://www.hatchconfigs.com/.

[20] P. Kazemian, G. Varghese, and N. McKeown. Header space analysis: Static checking for networks. In *NSDI*, April 2012.

[21] Z. Kerravala. What is behind network downtime? proactive steps to reduce human error and improve availability of networks. https://www.cs.princeton.edu/courses/archive/fall10/cos561/papers/Yankee04.pdf, January 2004.

[22] A. Khurshid, X. Zou, W. Zhou, M. Caesar, and P. B. Godfrey. Veriflow: Verifying network-wide invariants in real time. In *NSDI*, pages 15–27, April 2013.

[23] J. Kim, W. J. Dally, and D. Abts. Flattened butterfly: A cost-efficient topology for high-radix networks. In *ISCA*, pages 126–137, May 2007.

[24] F. Le, G. G. Xie, and H. Zhang. On route aggregation. In *CoNEXT*, December 2011.

[25] V. Liu, D. Halperin, A. Krishnamurthy, and T. Anderson. F10: A fault-tolerant engineered network. In *NSDI*, pages 399–412, April 2013.

[26] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *SIGCOMM*, August 2002.

[27] H. Mai, A. Khurshid, R. Agarwal, M. Caesar, P. B. Godfrey, and S. T. King. Debugging the data plane with anteater. In *SIGCOMM*, pages 290–301, August 2011.

[28] S. Narain, G. Levin, S. Malik, and V. Kaul. Declarative infrastructure configuration synthesis and debugging. *Journal of Network Systems Management*, 16(3):235–258, October 2008.

[29] J. Networks. As the value of enterprise networks escalates, so does the need for configuration management. https://www-935.ibm.com/services/au/gts/pdf/200249.pdf, May 2008.

[30] Quagga. Quagga routing suite. http://www.nongnu.org/quagga/.

[31] M. Reitblatt, M. Canini, N. Foster, and A. Guha. FatTire: Declarative fault tolerance for software defined networks. In *HotSDN*, August 2013.

[32] S. Saha, S. Prabhu, and P. Madhusudan. Netgen: Synthesizing data-plane configurations for network policies. In *SOSR*, pages 17:1–17:6, June 2015.

[33] B. Schlinker, R. N. Mysore, S. Smith, J. C. Mogul, A. Vahdat, M. Yu, E. Katz-Bassett, and M. Rubin. Condor: Better topologies through declarative design. In *SIGCOMM*, pages 449–463, August 2015.

[34] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *CoNEXT*, December 2014.

[35] thwack. configuration templates — thwack. https://thwack.solarwinds.com/search.jspa?q=configuration+templates.

[36] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock. Formal semantics and automated verification for the border gateway protocol. In *NetPL*, March 2016.