Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations

Ryan Beckett, Princeton
Ratul Mahajan, Microsoft
Todd Millstein, UCLA
Jitu Padhye, Microsoft
David Walker, Princeton

Abstract— We develop Propane, a language and compiler to help network operators with a challenging, error-prone task—bridging the gap between network-wide routing objectives and low-level configurations of devices that run complex, distributed protocols. The language allows operators to specify their objectives naturally, using high-level constraints on both the shape and relative preference of traffic paths. The compiler automatically translates these specifications to router-level BGP configurations, using an effective intermediate representation that compactly encodes the flow of routing information along policy-compliant paths. It guarantees that the compiled configurations correctly implement the specified policy under all possible combinations of failures. We show that Propane can effectively express the policies of datacenter and backbone networks of a large cloud provider; and despite its strong guarantees, our compiler scales to networks with hundreds or thousands of routers.

CCS Concepts
• Networks → *Network control algorithms; Network reliability; Network management;* • Software and its engineering → *Automated static analysis; Domain specific languages*

Keywords
Propane; Domain-specific Language; BGP; Synthesis; Compilation; Fault Tolerance; Distributed Systems

1. INTRODUCTION

It is well known that configuring networks is error prone and that such errors can lead to disruptive downtimes [23, 11, 13, 17]. For instance, a recent misconfiguration led to an hour-long, nation-wide outage for Time Warner's backbone network [4]; and a major BGP-related incident makes international news every few months [6].

A fundamental reason for the prevalence of misconfigurations is the semantic mismatch between the intended high-level policies and the low-level configurations. Many policies involve network-wide properties—prefer a certain neighbor, never announce a particular destination externally, use a particular path only if another fails—but configurations describe the behavior of individual devices. Operators must manually decompose network-wide policy into device behaviors, such that policy-compliant behavior results from the distributed interactions of these devices. Policy-compliance must be ensured not only under normal circumstances but also during failures. The need to reason about all possible failures exacerbates the challenge for network operators. As a result, configurations that work correctly in failure-free environments have nonetheless been found to violate key network-wide properties when failures occur [13].

To reduce configuration errors, operators are increasingly adopting an approach in which common tasks are captured as parameterized templates [19, 34]. While templates help ensure certain kinds of consistency across devices, they do not provide fundamentally different abstractions from existing configuration languages or bridge the semantic divide between network-wide policies and device-level configuration. Thus, they still require operators to manually decompose policies into device behaviors.

As a complementary approach, configuration analysis tools can help reduce misconfigurations by checking if low-level configurations match high-level policy [13, 11]. However, such tools cannot help operators with the challenging task of generating configurations in the first place.

Software-defined networking (SDN) and its abstractions are, in part, the research community's response to the difficulty of maintaining policy compliance through distributed device interactions [8]. Instead of organizing networks around a distributed collection of devices that compute forwarding tables through mutual interactions, the devices are told how to forward packets by a centralized controller. The controller is responsible for ensuring that the paths taken are compliant with operator specifications.

The centralized control planes of SDN, however, are not a panacea. First, while many SDN programming systems [14] provide effective *intra*-domain routing abstractions, letting users specify paths within their network, they fail to provide a coherent means to specify *inter*-domain routes. Second, centralized control planes require careful design and engineering to be robust to failures—one must ensure that all devices can communicate with the controller at all times, even under arbitrary failure combinations. Even ignoring failures, it is necessary for the control system to scale to meet the demands of large or geographically-distributed networks, and to react quickly to environmental changes. For this challenge, researchers are exploring multi-controller systems with

SIGCOMM '16 August 22–26, 2016, Florianopolis, Brazil
© 2016 ACM. ISBN 978-1-4503-2138-9.
DOI: 10.1145/1235

interacting controllers, thus bringing back distributed control planes [25, 5] and their current programming difficulties.

Hence, in this paper, we have two central goals:

1. Design a new, high-level language with natural abstractions for expressing intra-domain routing, inter-domain routing and routing alternatives in case of failures.
2. Define algorithms for compiling these specifications into configurations for devices running standard distributed control plane algorithms, while ensuring correct behavior independent of the number of faults.

To achieve the first goal, we borrow the idea of using regular expressions to specify network paths from recent high-level SDN languages such as FatTire [32], Merlin [33], and NetKAT [3]. However, our design also contains several key departures from existing languages. The most important one is semantic: the paths specified can extend from outside the operator's network to inside the network, across several devices internally, and then out again. This design choice allows users to specify preferences about both external and internal routes in the exact same way. In addition, we augment the algebra of regular expressions to support a notion of *preferences* and provide a semantics in terms of sets of ranked paths. The preferences indicate fail-over behaviors: among all specified paths that are still available, the system guarantees that the distributed implementation will always use the highest-ranked ones. Although we target a distributed implementation, the language is more general and could potentially be used in an SDN context.

To achieve the second goal, we develop program analysis and compilation algorithms that translate the regular policies to a graph-based intermediate representation and from there to per-device BGP configurations, which include various filters and preferences that govern BGP behavior. We target BGP for pragmatic reasons: it is a highly flexible routing protocol, it is an industry standard, and many networks use it internally as well as externally. Despite the advent of SDN, many networks will continue to use BGP for the foreseeable future due to existing infrastructure investments, the difficulty of transitioning to SDN, and the scalability and fault-tolerance advantages of a distributed control plane.

The BGP configurations produced by our compiler are guaranteed to be policy-compliant in the face of *arbitrary* failures.[1] This guarantee does not mean that the implementation is always able to send traffic to its ultimate destination (*e.g.*, in the case of a network partition), but rather that it always respects the centralized policy, which may include dropping traffic when there is no route. In this way, we provide network operators with a strong guarantee that is otherwise impossible to achieve today. However, some policies simply cannot be implemented correctly in BGP in the presence of arbitrary failures. We develop new algorithms to detect such policies and report our findings to the operators, so they may fix the policy specification at compile time rather than experience undesirable behavior after the configurations are deployed.

[1]We assume that BGP is the only routing protocol running in the network or the other protocols are correctly configured and do not have adverse interactions with BGP [18, 12].

We have implemented our language and compiler in a system called Propane. To evaluate it, we use it to specify real policies for datacenter and backbone networks. We find that our language expresses such policies easily, and that the compiler scales to topologies with hundreds or thousands of routers, compiling in under 9 minutes in all cases.

2. BACKGROUND ON BGP

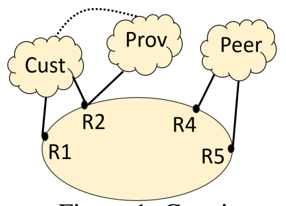
BGP is a path-vector routing protocol that connects autonomous systems (ASes). An AS has one or more routers managed by the same administrative entity. ASes exchange routing announcements with their neighbors. Each announcement has a destination IP prefix and some attributes (see below), and it indicates that the sending AS is willing to carry traffic destined to that prefix from the receiving AS. Traffic flows in the opposite direction, from announcement receivers to senders.

When a route announcement is received by an AS, it is processed by custom import filters that may drop the announcement or modify some attributes. If multiple announcements for the same prefix survive import filters, the router selects the best one based on local policy (see below). This route is then used to send traffic to the destination. It is also advertised to the neighbors, after processing through neighbor-specific export filters that may stop the announcement or modify some attributes.

All routing announcements are accompanied by an AS-path attribute that reflects the sequence of ASes that the announcement has traversed thus far. While the AS-path attribute has a global meaning, some attributes are meaningful only within an AS or between neighboring ASes. One such attribute is a list of community strings. ASes use such strings to associate network-specific information with particular routes (*e.g.*, "entered on West Coast") and then use the information later in the routing process. Communities are also used to signal to neighbors how they should handle an announcement (*e.g.*, do not export it further). Another non-global attribute is the multi-exit discriminator (MED). It is used when an AS has multiple links to a neighboring AS. Its (numeric) values signal to the neighbor how this AS prefers to receive traffic among those links.

The route selection process assigns a *local preference* to each route that survives the import filters. Routes with higher local preference are preferred. Among routes with the same local preference, other factors such as AS path length, MEDs, and internal routing cost, are considered in order. Because it is considered first during route selection, local preference is highly influential, and ASes may assign this preference based on any aspect of the route. A common practice is to assign it based on the commercial relationship with the neighbor. For instance, an AS may prefer in order customer ASes (which pay money), peer ASes (with free exchange of traffic), and provider ASes (which charge money for traffic).

The combination of arbitrary import and export filters and route selection policies at individual routers make BGP a



Policy

P1. Prefer Cust > Peer > Prov

P2. Disallow traffic between Prov and Peer

P3. For Cust, prefer R1 > R2

P4. Cust must be on path for its prefixes

P5. Cust must not be a transit to Prov

Figure 1: Creating router-level policies is difficult.

highly flexible routing protocol. That flexibility, however, comes at the cost of it being difficult to configure correctly. When configuring BGP, network operators assume that neighboring ASes correctly implement BGP and honor contracts for MEDs and communities. Propane makes the same assumption when deriving BGP configurations for a network.

3. MOTIVATION

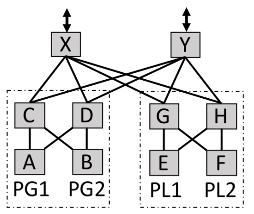
When generating BGP configurations, whether manually or aided by templates, the operators face the challenge of decomposing network-wide policies into correct device-level policies. This decomposition is not always straightforward and ensuring policy-compliance is tricky, especially in the face of failures. In this section, we illustrate this difficulty using two examples based on policies that we have seen in practice. The next section shows how Propane allows operators to express these policies naturally.

3.1 Example 1: The backbone

Consider the backbone network in Figure 1. It has three neighbors, a customer `Cust`, a peer `Peer`, and a provider `Prov`. The policy of this network is shown on the right. It prefers that traffic leave the network through neighbors in a certain order (P1) and does not want to act as a transit between `Peer` and `Prov` (P2). It prefers to exchange traffic with `Cust` over `R1` rather than `R2` because `R1` is cheaper (P3). To guard against another AS "hijacking" prefixes owned by `Cust`, the network only sends traffic to a neighbor if `Cust` is on the AS path (P4). Finally, to guard against `Cust` accidentally becoming a transit for `Prov`, it does not use `Cust` for traffic that will later traverse `Prov` (P5).

To implement policy P1, the operators must compute and assign local preferences such that preferences at `Cust`-facing interfaces $>$ `Peer`-facing interfaces $>$ `Prov`-facing interfaces. At the same time, to satisfy P3, the preference at `R2`'s `Cust`-facing interface should be lower than that at `R1`. Implementing P3 will also require MEDs to be appropriately configured on `R1` and `R2`. To implement P2, the operators can assign communities that indicate where a certain routing announcement entered the network. Then, `R4` must be configured to not announce to `Peer` routes that have communities that correspond to the `R2-Prov` link but to announce routes with communities for the `R2-Cust` and `R1-Cust` links. A similar type of policy must be configured for `R2` as well. Finally, to implement P4 and P5, the operators will have to compute and configure appropriate prefix- and AS-path-based import and export filters at each router.

Clearly, it is difficult to correctly configure even this small example network manually; correctly configuring real, larger networks can quickly become a nightmare. Such networks have hundreds of neighbors spanning multiple commercial-relationship classes, differing numbers of links to each neighbor, along with several neighbor- or prefix-based exceptions to the default behavior. A large AS with many peers in different geographic locations may be faced with complex challenges such as keeping traffic within national boundaries. Templates help to an extent by keeping preference and community values consistent across routers, but operators must still do much of the conceptually difficult work manually.



Policy

P1. Left cluster has global services with PG* prefixes, which should be announced externally as an aggregate PG

P2. Right cluster has local services with PL* prefixes, which should not be announced externally

Figure 2: Policy-compliance under failures is difficult.

3.2 Example 2: The datacenter

While configuring policies for a fully functional network is difficult, ensuring policy compliance in the face of failures can be almost impossible. Consider the datacenter network in Figure 2 with routers organized as a fat tree and running BGP.[2] The network has two clusters, one with services that should be reachable globally and one with services that should be accessible only internally. This policy is enabled by using non-overlapping address space in the two clusters and ensuring that only the address space for the global services is announced externally. Further, to reduce the number of prefixes that are announced externally, the global space is aggregated into a less-specific prefix `PG`. The semantics of aggregation is that the aggregate prefix is announced as long as the router has a path to at least one sub-prefix.

The operator may implement the policy by having X and Y: i) not export externally what they hear from G and H, routers that belong to the local services cluster; and ii) export externally what they hear from routers C and D and aggregate to `PG` if an announcement is subset of `PG`. This implementation is appealing because X and Y do not need to be made aware of which prefixes are global versus local and IP address assignment can occur independently, *e.g.*, local services can be assigned new prefixes without updating those routers' configurations.

However, this implementation has incorrect behavior in the face of failures. Suppose links X–G and X–H fail. Then, X will hear announcements for `PL*` from C and D, having traversed from G and H to Y to C and D. Per its policy implementation, X will start "leaking" these prefixes externally. Depending on the rationale for local services, this leak could impact security (*e.g.*, if the services are sensitive) or availability (*e.g.*, if the `PL*` prefixes are reused for other services outside of the datacenter). This problem does not manifest without failures because then X has and prefers paths to `PL*`

[2]For scale and policy flexibility, datacenter networks increasingly use BGP internally, with a private AS number per router [20].

through G and H since they are shorter. A similar problem will occur if links Y–G and Y–H fail. Link failures in datacenters are frequent and it is not uncommon to have many failed links at a given time [17].

To avoid this problem, the operator may disallow "valley" paths, *i.e.*, those that go up, down, and back up again. This guard can be implemented by X and Y rejecting paths through the other. But that creates a different problem in the face of failures—an aggregation-induced black hole [21]. If links D–A and X–C fail, X will hear an announcement for `PG2` from D and will thus announce `PG` externally. This announcement will bring traffic for `PG1` to X as well, but because valleys are disallowed, X does not have a valid route for `PG1` and will drop all traffic for it despite the fact that a valid path exists through Y.

Thus, we see that devising a configuration that ensures policy compliance in the face of failures is complex and error-prone. Propane lets operators implement their high-level policy specification in a way that guarantees compliance under all failures if possible—otherwise, it generates a compile-time error. For aggregation, it also provides a lower bound to operators on the number of failures under which aggregation will not result in black holes.

4. PROPANE OVERVIEW

Policies for (distributed) control planes differ from data-plane policies in a few important ways. First, they must account for all failures at compile time; there is no controller at runtime, so the routers must be configured in advance to handle failures in a compliant manner. In Propane, we enable such specifications through *path preferences*, with the semantics that a less-preferred path is taken only when a higher-preference path is unavailable in the network. Second, paths in a control-plane policy may be under-specified (*e.g.*, "prefer customer" does not indicate a concrete path). The Propane compiler treats such under-specifications as constraints on the set of allowed paths and automatically computes valid sets based on the topology.

This section introduces the Propane language using the examples from the previous section. The next section describes the complete syntax of the language as well as our strategy for compiling it to BGP.

4.1 Example 1: The backbone

Propane lets operators configure the network with the abstraction that they have centralized control over routing. Specifically, the operator simply provides a set of high-level constraints that describe the paths traffic should—or should not—take and their relative preferences. Propane specifications are written modularly via a series of declarations. For example, to begin specification of the backbone network from the previous section, we first express the idea that we prefer that traffic leave the network through `R1` over `R2` (to `Cust`) over `Peer` over `Prov` (policy P1 and P3 from Figure 1):

```
define Prefs = exit(R1 » R2 » Peer » Prov)
```

This statement defines a set of *ranked paths*, which includes all paths (and only those paths) for which traffic exits our network through either router `R1`, router `R2`, `Peer`, or `Prov`. The paths that exit through `R1` are preferred (») to those that exit through `R2`, which are preferred to those that leave through `Peer` and then `Prov`. As we describe in the next section, the **`exit`** expression, as well as other path expressions used later in this section, is simply a shorthand for a particular regular expression over paths that is expressible in our policy language. The preference operator (») is flexible and can be used between constraints as well as among individual routers. For example, the above constraint could have been written equivalently as **`exit`**`(R1)` »... » **`exit`**`(Prov)`

To associate ranked paths with one or more prefixes, we define a Propane *policy*. Within a policy, statements with the form t => p associate the prefixes defined by the predicate t with the set of ranked paths defined by the path expression p. In general, prefix predicates can be defined by arbitrary boolean combinations (and, or, not) of concrete prefixes and community tags. Here, we assume we have already defined the predicate `PCust` for the customer prefixes. In the following code, ranked paths are associated with customer prefixes, and all other prefixes (`true`). Policy statements are processed in order with earlier policy statements taking precedence over later policy statements. Hence, when the predicate `true` follows the statement involving `PCust`, it is interpreted as `true & !PCust`.

```
define Routing =
    {PCust => Prefs & end(Cust)
     true  => Prefs }
```

Line 2 of this policy restricts traffic destined to known customer prefixes (`PCust`) to only follow paths that end at the customer. In addition, it enforces the network-wide preference that traffic leaves through `R1` over `R2` over `Peer` over `Prov`. Line 3 applies to any other traffic not matching `PCust` and allows the traffic to leave through any direct neighbor with the usual preferences of `R1` over `R2` over `Peer` over `Prov`. To summarize our progress, the `Routing` policy implements P1, P3, and P4 from Figure 1.

Since, routing allows transit traffic by default (*e.g.*, traffic entering from `Peer` and leaving through `Prov`), we separately define a policy to enforce P2 and P5 from Figure 1, using conjunction (`&`), disjunction (`|`) and negation (`!`) of constraints. First, we create reusable abstractions for describing traffic that transits our network. In Propane, this is done by creating a new parameterized definition.

```
define transit(X,Y) = enter(X|Y) & exit(X|Y)
define cust-transit(X,Y) = later(X) & later(Y)
```

Here we define transit traffic between groups of neighbors X and Y as traffic that enters the network through some neighbor in X or Y and then also leaves the network through some neighbor in either X or Y. Similarly, we define customer transit for customer X and provider Y as traffic that later goes through both X and Y after leaving our network. Using these two new abstractions, we can now implement policies P2 and P5 with the following constraint.

```
define NoTrans =
  {true => !transit(Peer,Prov) &
           !cust-transit(Cust,Prov)}
```

The `NoTrans` constraint requires that all traffic not follow a path that transits the network between `Peer` and `Prov`. Additionally, it prevents traffic from ever following paths that leave our network and later go through both `Prov` and `Cust`. To implement both `Routing` and `NoTrans` simultaneously, we simply conjoin them: `Routing & NoTrans`.

Collectively, the constraints above capture the entire policy. From them, our compiler will generate per-device import and export filters, local preferences, MED attributes, and community tags to ensure that the policy is implemented correctly under all failures.

4.2 Example 2: The datacenter

Our datacenter example network has three main concerns: (1) traffic for the prefix allocated to each top-of-rack router must be able to reach that router, (2) local services must not leak outside the datacenter, and (3) aggregation must be performed on global prefixes to reduce churn in the network.

Propane allows modular specification of each of these constraints. The first constraint is about prefix ownership—we want traffic only for certain prefixes to end up at a particular location. The following definition captures this intent.

```
define Ownership =
    {PG1 => end(A)
     PG2 => end(B)
     PL1 => end(E)
     PL2 => end(F)
     true => end(out)}
```

This definition says that traffic for prefix `PG1` is allowed to follow only paths that end at router `A`; traffic for `PG2`, but not `PG1`, must end at router `B`; and so on. Any traffic destined for a prefix that is not a part of the datacenter should be allowed to leave the datacenter and end at some external location, which is otherwise unconstrained. The special keyword **`out`** matches any location outside the datacenter network, while the keyword **`in`** will match any location inside the network.

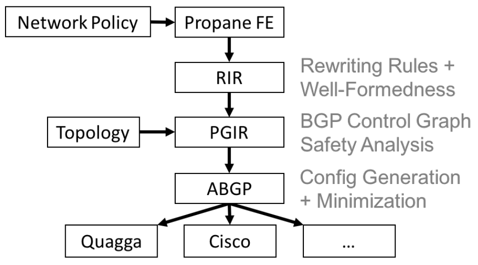
For the second constraint, we define another policy:

```
define Locality =
    {PL1 | PL2 => only(in)}
```

This definition says that traffic for local prefixes only follows paths that are internal to the network at each hop. This constraint guarantees that the services remain accessible only to locations inside the datacenter.

As in the backbone example, we can logically conjoin these constraints to specify the network-wide policy. However, in addition to constraints on the shape of paths, Propane allows the operator to specify constraints on the BGP control plane itself. For instance, a constraint on aggregation is included to ensure that aggregation for global prefixes is performed from locations inside (**`in`**) the network to locations outside (**`out`**). In this case, `PG1` and `PG2` will use the aggregate `PG` (which we assume is defined earlier) when advertised outside the datacenter.

```
Ownership & Locality & agg(PG, in -> out)
```


Network Policy
Propane FE
RIR
Rewriting Rules + Well-Formedness
Topology
PGIR
BGP Control Graph Safety Analysis
ABGP
Config Generation + Minimization
Quagga
Cisco
...
Figure 3: Compilation pipeline stages for Propane.

Once Propane compiles the policy, it is guaranteed to remain compliant under all possible failure scenarios, modulo any aggregation-induced black holes. In the presence of aggregation, the Propane compiler will also efficiently find a lower bound on the number of failures required to create an aggregation-induced black hole.

5. COMPILATION

The examples above use what we call the front end (FE) of Propane. It simplifies operators' task of describing preferred paths, but that simplicity comes at the cost of compilation complexity. The compiler must efficiently compute the sets of paths represented by the intersection of preferences and topology and ensure policy compliance under all failure scenarios.

To handle these challenges, we decompose compilation into multiple stages, shown in Figure 3, and develop efficient algorithms for the translation between stages. The first stage of the pipeline involves simple rewriting rules and substitutions from the FE to the core Regular Intermediate Representation (RIR). Policies in RIR are checked well-formedness (*e.g.*, never constraining traffic that does not enter the network), before being combined with the topology to obtain the Product Graph Intermediate Representation (PGIR). The PGIR is a data representation that compactly captures the flow of BGP announcements subject to the policy and topology restrictions. We develop efficient algorithms that operate over the PGIR to ensure policy compliance under failures, avoid BGP instability, and prevent aggregation-induced black holes. Once the compiler determines safety, it translates the PGIR to an abstract BGP (ABGP) representation. ABGP can then be translated into various vendor-specific device configurations as needed.

5.1 Regular IR (RIR)

The Propane FE is just a thin layer atop the RIR for describing preference-based path constraints. Figure 4 shows the RIR syntax. A policy has one or more constraints. The first kind of constraint is a test on the type of route and a corresponding set of preferred regular paths or a control constraint. Regular paths are regular expressions where the base characters are abstract locations representing either a router or an external AS. Special **`in`** and **`out`** symbols refer to any internal or external location respectively. In addition, Σ refers to any symbol. We also use the standard regular expression abbreviation r^+ for $r \cdot r^*$, a sequence of one or more occurrences of r. Predicates (t) consist of log-

Syntax

pol	$::=$	$p_1, \ldots, p_n$	*policies*
p	$::=$	t => r_1» … »r_m $\mid$ cc	*constraints*
x	$::=$	$d.d.d.d/d$	*prefix*
t	$::=$	`true`	*true*
	$\mid$	$!\,t$	*negation*
	$\mid$	$t_1 \mid t_2$	*disjunction*
	$\mid$	$t_1 \& t_2$	*conjunction*
	$\mid$	*prefix* $= x$	*prefix test*
	$\mid$	*comm* $= d$	*community test*
r	$::=$	l	*location*
	$\mid$	$\emptyset$	*empty set*
	$\mid$	**in**	*internal loc*
	$\mid$	**out**	*external loc*
	$\mid$	$r_1 \cup r_2$	*union*
	$\mid$	$r_1 \cap r_2$	*intersection*
	$\mid$	$r_1 \cdot r_2$	*concatenation*
	$\mid$	$!\,r$	*path negation*
	$\mid$	r^*	*iteration*
ln	$::=$	$r_1 \rightarrow r_2$	*links*
cc	$::=$	$agg(x, ln) \mid tag(d, t, ln)$	*control constraints*

Propane Expansions

any	$=$	$\textbf{out}^* \cdot \textbf{in}^+ \cdot \textbf{out}^*$
drop	$=$	$\emptyset$
internal	$=$	$\textbf{in}^+$
only(X)	$=$	$\textbf{any} \cap X^*$
never(X)	$=$	$\textbf{any} \cap (!X)^*$
through(X)	$=$	$\textbf{out}^* \cdot \textbf{in}^* \cdot X \cdot \textbf{in}^* \cdot \textbf{out}^*$
later(X)	$=$	$\textbf{out}^* \cdot (X \cap \textbf{out}) \cdot \textbf{out}^* \cdot \textbf{in}^+ \cdot \textbf{out}^*$
before(X)	$=$	$\textbf{out}^* \cdot \textbf{in}^+ \cdot \textbf{out}^* \cdot (X \cap \textbf{out}) \cdot \textbf{out}^*$
end(X)	$=$	$\textbf{any} \cap (\Sigma^* \cdot X)$
start(X)	$=$	$\textbf{any} \cap (X \cdot \Sigma^*)$
exit(X)	$=$	$(\textbf{out}^* \cdot \textbf{in}^* \cdot (X \cap \textbf{in}) \cdot \textbf{out} \cdot \textbf{out}^*) \cup$ $(\textbf{out}^* \cdot \textbf{in}^+ \cdot (X \cap \textbf{out}) \cdot \textbf{out}^*)$
enter(X)	$=$	$(\textbf{out}^* \cdot \textbf{out} \cdot (X \cap \textbf{in}) \cdot \textbf{in}^* \cdot \textbf{out}^*) \cup$ $(\textbf{out}^* \cdot (X \cap \textbf{out}) \cdot \textbf{in}^+ \cdot \textbf{out}^*)$
link(X, Y)	$=$	$\textbf{any} \cap (\Sigma^* \cdot X \cdot Y \cdot \Sigma^*)$
path$(\vec{X})$	$=$	$\textbf{any} \cap (\Sigma^* \cdot X_1 \ldots X_n \cdot \Sigma^*)$
novalley$(\vec{X})$	$=$	$\textbf{any} \cap$ $!\textbf{path}(X_2, X_1, X_2) \cap \cdots \cap$ $!\textbf{path}(X_n, X_{n-1}, X_n)$

Figure 4: Regular Intermediate Representation (RIR) syntax (left), and Propane language expansions (right).

ical boolean connectives (and, or, not) as well as tests that match a particular prefix (or group of prefixes) and tests for route advertisements with a particular community value (*i.e.*, an integer value associated with a path).

Propane also supports constraints on the control-plane behavior of BGP. For example, prefix aggregation is an important optimization to reduce routing table size. A constraint of the form $agg(x, ln)$ tells the compiler to perform aggregation for prefix x across all links described by ln. It is also often useful to be able to add community tags to exported routes in BGP (*e.g.*, to communicate non-standard information to peers). A constraint of the form $tag(d, t, ln)$ adds community tag d for any prefixes matching t across links ln. We list only the route aggregation and community tagging constraints in Figure 4, but Propane also supports other constraints such as limiting the maximum number of routes allowed between ASes, or enabling BGP multipath.

Semantics. We give a semantics to RIR programs using sets of ranked paths. Each path constraint r_1» … »r_j denotes a set of ranked network paths. A network path is a topologically valid string of abstract locations $l_1 l_2 \ldots l_k$. We use the notation $|p|$ to denote the length of the path p. A regular expression r matches path p, if $p \in \mathcal{L}(r)$, that is, the path is in the language of the regular expression. Paths are ranked lexicographically according to (1) the most preferred regular expression matched, and (2) as a tie breaker, the path length. Lower ranks indicate *more* preferred paths. More formally, a path p has rank:

$$(\min_i \{p \in \mathcal{L}(r_i)\}, |p|)$$

The set of ranked paths depends on which paths are valid in the topology, and thus when failures occur, the most preferred routes change. For any source s and destination d, Propane will send traffic along the highest ranked available path from s to d.

From FE to RIR. The first stage in Propane compilation reduces the FE to the simpler RIR from Figure 4. The main differences between the FE and RIR are: i) FE allows the programmer to specify constraints using a series of (modular) definitions, and combine them later, ii) FE provides high-level names that abstract sets of routes and groups of prefixes/neighbors, and iii) FE allows the preference operator to be used more flexibly.

A key constraint when translating FE to RIR is ensuring that all specified routes are well-formed. In particular, each regular path constraint r must satisfy $r \subseteq \textbf{out}^* \cdot \textbf{in}^+ \cdot \textbf{out}^*$. It ensures that users only control traffic that goes through their network at some point, and that such traffic does not loop back multiple times through their network.

The translation from FE to RIR is based on a set of rewriting rules. The first step merges separate constraints. It takes the cross product of per-prefix constraints, where logical conjunction (r_1 & r_2) is replaced by intersection on regular constraints ($r_1 \cap r_2$), logical disjunction is replaced by union, and logical negation ($!\,r$) is replaced by path negation ($\textbf{any} \cap\ !\,(r)$). The additional constraint **any** ensures the routes are well-formed by restricting the paths to only those that go through the user's network. For example, in the datacenter FE configuration from §4, combining the `Locality` and `Ownership` policies results in the following RIR:

```
PG1 => end(A)
PG2 => end(B)
PL1 => only(in) ∩ end(E)
PL2 => only(in) ∩ end(F)
true => exit(out)
```

The next step rewrites the high-level constraints such as **enter** according to the equivalences in Figure 4. Since preferences can only occur at the outermost level for an RIR expression, the final step is to "lift" occurrences of the preference operator in each regular expression. For example, the regular expression $r \cdot (s$»$t) \cdot u$ is lifted to $(r \cdot s \cdot u)$»$(r \cdot t \cdot u)$

by distributing the preference over the sequence operator. In general, we employ the following distributivity equivalences:

$$r \odot (s_1 \text{»} \ldots \text{»} s_n) = (r \odot s_1)\text{»} \ldots \text{»}(r \odot s_n)$$
$$(s_1 \text{»} \ldots \text{»} s_n) \odot r = (s_1 \odot r)\text{»} \ldots \text{»}(s_n \odot r)$$

where $\odot$ stands for an arbitrary regular binary operator, and r is a policy with a single preference. The compiler flags preferences nested under a unary operator (*i.e.*, *star* or *negation*) as invalid.

5.2 Product graph IR

Now that the user policy exists in a simplified form, we must consider the topology. In particular, we want a compact representation that describes all the possible ways BGP route announcements can flow through the network subject to the policy and topology constraints. The PGIR captures these constraints by "intersecting" each of the regular automata corresponding to the RIR path preferences with the topology. Paths through the PGIR correspond to real paths through the topology that satisfy the user constraints.

Formal definition. While paths in an RIR policy describe the direction traffic flows through the network, to implement the policy with BGP we are concerned about the way control-plane information is disseminated, *i.e.*, route announcements flowing in the opposite direction. To capture this idea, for each regular expression r_i in an RIR policy, we construct a deterministic finite state machine on the reversed regular expression. Each automaton for the reversed regular expression of r_i is a tuple $(\Sigma, Q_i, F_i, q_{0_i}, \sigma_i)$. The alphabet Σ consists of all abstract locations (*i.e.*, routers or ASes), 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 topology is represented as a graph (V, E), which consists of a set of vertices V and a set of directed edges $E\colon V \times V$. The combined PGIR is a tuple (V', E', s, e, P) with vertices $V'\colon V \times Q_1 \times \cdots \times Q_j$, edges $E'\colon V' \times V'$, a unique starting vertex s, a unique ending vertex e, and a preference function $P\colon V' \to 2^{\{1,\ldots,j\}}$, which maps nodes in the product graph to a set of path ranks.

For a PGIR vertex $n = (l, \ldots) \in V'$, we say that n is a *shadow* of topology location l, written as $\tilde{n} = l$, to indicate that the topology location for node n is l. When two PGIR nodes m and n shadow the same topology location (*i.e.*, $\tilde{m} = \tilde{n}$), we write $m \approx n$.

Throughout the remainder of the paper, we will use the convention that metavariables m and n stand for PGIR nodes and l stands for a topology location. Capital letters like X refer to concrete topology locations, while capital letters with subscripts such as X_1 and X_2 refer to concrete PGIR nodes that share a topology location (*i.e.*, $\tilde{X_1} = \tilde{X_2} = X$).

From RIR To PGIR. Let a_i and b_i denote states in the regular policy automata. The PGIR is constructed by adding an edge from $m = (l_m, a_1, \ldots, a_k)$ to $n = (l_n, b_1, \ldots, b_k)$ whenever $\sigma_i(a_i, l_n) = b_i$ for each i and $(l_m, l_n) \in E$ is a valid topology link. Additionally, we add edges from the start node s to any $n = (l, a_1, \ldots, a_k)$ when $\sigma_i(q_{0_i}, l) = a_i$ for each i. The preference function P is defined as $P(n) = \{i \mid a_i \in F_i\}$. That is, it records the path rank of each regular expression matched in the current node. Finally, there is an edge from each node in the PGIR such that $P(n) \neq \emptyset$ to the special end node e. We write $(m \leq_{rank} n)$ if either $P(m) = P(n) = \emptyset$ or $min\ P(m) \leq min\ P(n)$, which means that paths ending at PGIR node m are better (lower rank) than paths ending at n.

Intuitively, the PGIR tracks the policy states of each automaton as route announcements move between locations. Consider the topology in Figure 5. Suppose we want a primary route from neighbor W that allows traffic to enter the network at A and utilize the C–D link before leaving the network (through X or Y). As a backup, we also want to allow traffic to enter the network from B, in which case the traffic can also utilize the C–E link before leaving the network. For simplicity, we assume that the route ends in either X, Y, or Z. The RIR for the policy could be written as:

$$(\texttt{W} \cdot \texttt{A} \cdot \texttt{C} \cdot \texttt{D} \cdot \textbf{out})\text{»}(\texttt{W} \cdot \texttt{B} \cdot \textbf{in}^{+} \cdot \textbf{out})$$

Figure 5 shows the policy automata for each regular expression preference. Since we are interested in the flow of control messages, the automata match backwards. The figure also shows the PGIR after intersecting the topology and policy automata. Every path in the PGIR corresponds to a concrete path in the topology. In particular, every path through the PGIR that ends at a node n such that the preference function $P(n) = \{i_1, \ldots, i_k\}$ is non-empty, is a valid topological path that satisfies the policy constraints and results in a particular path with preferences i_1 through i_k. For example, the path $X \cdot D \cdot C \cdot A \cdot W$ is a valid path in the topology that BGP route announcements might take, which would lead to obtaining a path with the lowest (best) rank of 1. BGP control messages can start from peer X, which would match the `out` transition from both automata, leading to state 1 in the first automaton, and state 1 in the second automaton. This possibility is reflected in the product graph by the node with state $(X, 1, 1)$. From here, if X were to advertise this route to D, it would result in the path $D \cdot X$, which would lead to state 2 in the first automaton, and state 2 in the second automaton, and so on. The "–" state indicates the corresponding automaton cannot accept the current path or any extension of it. Since node $(W, 5, -)$ is in an accepting state for the first automaton, it indicates that this path has preference 1.

Minimization. After building the PGIR as described above, we minimize it in order to improve the precision of the subsequent analysis that checks if the policies captured by it are safe under failures. The minimization is based on the observation that, although every path in the PGIR is a valid path in the topology, we do not want to consider paths that form loops. In particular, BGP's loop prevention mechanism forces an AS to reject any route that is already in the AS path. For example, in Figure 5, the path $W \cdot A \cdot C \cdot B \cdot W$ is a valid topological path, leading to a path that satisfies the preference 2 policy, but which contains a loop.

We use graph dominators [22] as a relatively cheap ap-

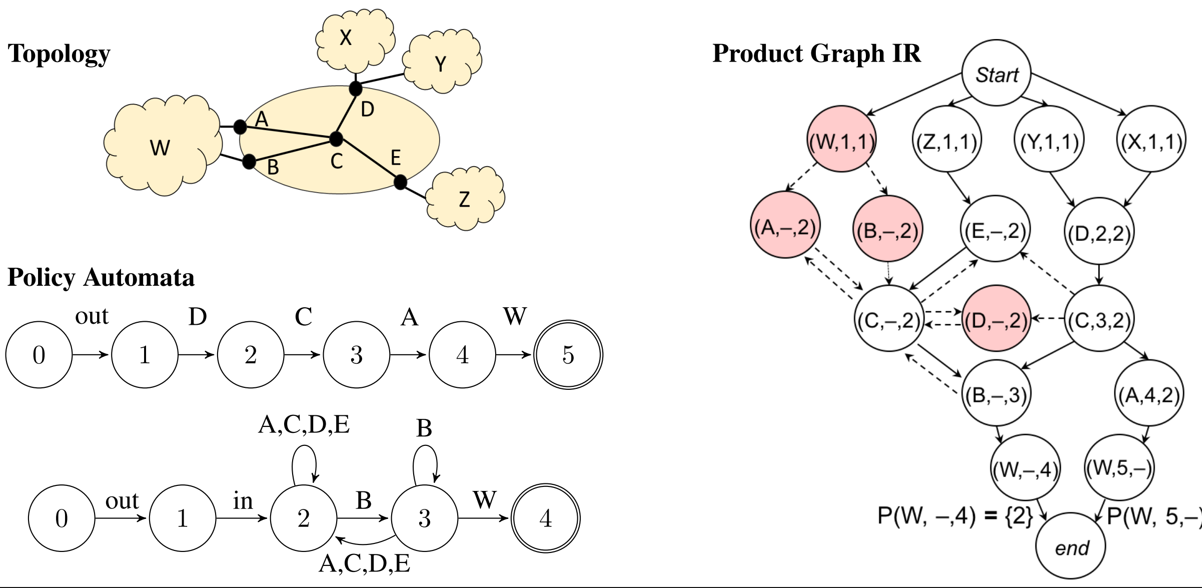


Figure 5: Product graph construction for policy $(\texttt{W} \cdot \texttt{A} \cdot \texttt{C} \cdot \texttt{D} \cdot \textbf{out})$»$(\texttt{W} \cdot \texttt{B} \cdot \textbf{in}^+ \cdot \textbf{out})$.

proximation for removing many nodes and edges in the PGIR that are never on any *simple* (loop free) path between the start and end nodes. In the PGIR, a node m dominates a node n if m appears on every path leading from the start node to n. Similarly, a node m post-dominates a node n in the PGIR if m appears on every path from n to the end node. We can safely remove nodes and edges in the PGIR when any of the following conditions hold, where we have m, m' and n, n' such that $m \approx m'$ and $n \approx n'$.

- Remove m if it is not reachable from the start node
- Remove m if it can not reach the end node
- Remove m if it is (post-)dominated by some m'
- Remove edge (m, n) if some m' post-dominates n
- Remove edge (m, n) if some n' dominates m

For example, node $(W, 1, 1)$ in Figure 5 is removed because every path to the end node must always go through node $(W, -, 4)$. That is, node $(W, 1, 1)$ is post-dominated by node $(W, -, 4)$ and both are shadows of topology location W.

We repeatedly apply the minimizations above until no further minimization is possible. In the example from Figure 5, colored nodes and dashed edges show edges and nodes removed after minimization.

5.3 Failure-safety analysis

To implement path preferences in routing, BGP uses local preferences on a per-device basis. However, the distributed nature of BGP makes setting preferences locally to achieve a network-wide routing policy difficult. This task becomes even more challenging in the presence of failures since routers running BGP lack a global view of the network.

An illustrative example. To demonstrate the difficulty of generating device-local preferences, consider the simple policy for the topology in Figure 6, which says to prefer the top path over the bottom path: $(\texttt{A} \cdot \texttt{B} \cdot \texttt{D} \cdot \texttt{E} \cdot \texttt{G})$»$(\texttt{A} \cdot \texttt{C} \cdot \texttt{D} \cdot \texttt{F} \cdot \texttt{G})$. How could such a policy be implemented in BGP? Suppose we set the local preferences to have D prefer E over F, and have A prefer B over C. This works as expected under normal conditions, however, if the A–B link fails, then suddenly D has made the wrong decision by preferring E. Traffic will now follow the $A \cdot C \cdot D \cdot E \cdot G$ path, even though this path was not allowed by the policy. Thus, the distributed implementation has used a route that is not allowed by the policy. To make matters worse, the second preference for the path $A \cdot C \cdot D \cdot F \cdot G$ is available in the network but not being used. Thus, a path for the best possible route available exists in the network but is not being used by the distributed implementation. The first problem could be fixed by tagging and filtering route advertisements appropriately so that C rejects routes that go through E, however the second problem cannot be fixed. In fact, this policy cannot be implemented in BGP in a way that is policy compliant under all failures since D cannot safely choose between E and F without knowing whether the A–B link is available.

Problem Formulation. The problem of determining local preferences for each router is reflected in the structure of the PGIR. Whenever a given router appears as multiple shadow nodes in the PGIR, the compiler must decide which shadow to prefer. In the example from Figure 5, the topology node C can receive an advertisement from E in shadow $(C, -, 2)$ or from D in shadow $(C, 3, 2)$. The compiler must determine a *total ordering* of shadow nodes for each router, which reflects the relative preference of advertisements received in each shadow and should be consistent with path ranks in routing policy. For example, if C's shadow $(C, 3, 2)$ can be preferred to $(C, -, 2)$, written as $(C, 3, 2) \leq_{lp} (C, -, 2)$, C can prefer advertisements from $(D, 2, 2)$ over $(E, -, 2)$. D and E tag their advertisements to let C know which shadow sent the advertisement. \$5.5 discusses tagging in detail.

Regret-free Preferences. To order PGIR shadows ($\leq_{lp}$) for each router in a way that is policy-compliant under all fail-

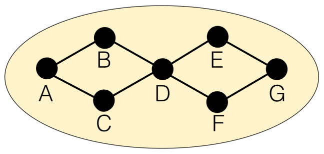


Figure 6: A network where the policy (A · B · D · E · G)»(A · C · D · F · G) is unimplementable in BGP under arbitrary failures.

ures, we introduce the notion of *regret-free* preferences, motivated by the observations from the example in Figure 6. A router (location) l has a regret-free preference for a set of advertisements A over B if, whenever l selects an advertisement to destination d from A over another from B, there is always some policy-compliant path to d that is at least as good ($\leq_{rank}$ for the final node along the path) as any possible path (not necessarily from l) to d if l had selected an advertisement from B instead. In other words, the preference of A over B at l is regret-free if l is never (under any failure) worse-off by choosing an advertisement from A when available. The notion of regret-free preferences can be lifted to PGIR nodes by considering the set of advertisements available to each node.

In the example of Figure 5, the choice for C to prefer shadow $(C, 3, 2)$ to $(C, -, 2)$ is regret-free, since there will always be at least as good a path to destination W regardless of any failures that might occur in the network. For example, if the C–A link fails, then there is still a backup path from $(C, 3, 2)$ to $(W, -, 4)$ that is just as good as any path from $(C, -, 2)$. Likewise, any combination of failures to disconnect $(C, 3, 2)$ from $(W, -, 4)$ would also disconnect $(C, -, 2)$ from $(W, -, 4)$.

A Preference Inference Algorithm. Searching for precise regret-free preferences in general is hard, and clearly enumerating all possible combinations of failures and preference orderings is intractable. We thus adopt a conservative analysis that we found to be effective and efficient in practice. The idea is to i) search for regret-free preferences by comparing the set of paths available after accepting advertisements in two different PGIR shadows N_1 and N_2 of topology node N, and ii) refine the comparison when necessary by considering where the announcements must have traversed before arriving at N_1 or N_2.

Algorithm 1 checks whether one shadow can be preferred to another ($N_1 \leq_{lp} N_2$). It walks from nodes N_1 and N_2 and ensures that for every *step* N_2 can take to some new topology location, N_1 can, at the very least, also take a step to an equivalent topology location ($\approx$). When there is no such equivalent step, the algorithm attempts to take into account where the advertisement must have already traversed. In particular, it checks if there is an equivalent dominator node and, if so, walks from this new node instead. The idea is that, since the advertisement must have already passed through the dominator, we can check to see if we are guaranteed to find paths that are at least as good from this new node instead. At each step, it requires that the current node reachable from N_1 has a path rank that is at least as good as that of the current node reachable from N_2 ($m \leq_{rank} n$). The intuition here is that if $m \not\leq_{rank} n$, then we can very likely fail every edge in the topology except for the path that leads to the current m and n, thereby generating a counterexample. Algorithm 1 terminates since the number of related states (m, n) that can be explored is finite.

Algorithm 1 Inferring regret-free preferences

```
1:  procedure REGRET-FREE($G$, $N_1$, $N_2$)
2:      if $N_1 \not\approx N_2$ then return false
3:      $q \leftarrow Queue()$
4:      $q.Enqueue(N_1, N_2)$
5:      while $!q.Empty()$ do
6:          $(m, n) \leftarrow q.Dequeue()$
7:          if $m \not\leq_{rank} n$ then return false
8:          for $n'$ in adj($G$, $n$) do
9:              if $(\exists m' \in$ adj$(G, m), m' \approx n')$ or
10:                 $(\exists m' \in G,$ dominates $m, m' \approx n')$ then
11:                 if $(m', n')$ not marked then
12:                     mark $(m', n')$ as seen
13:                     $q.Enqueue(m', n')$
14:             else return $false$
15:     return true
```

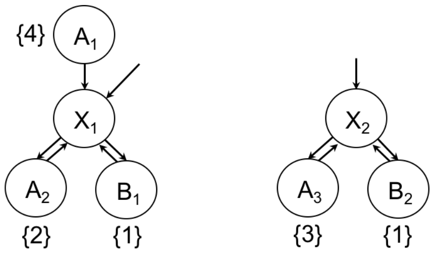


Figure 7: A product graph where preference inference is unsound before considering loops. Path ranks shown by nodes.

For each router in the topology, local preferences are now obtained by sorting the corresponding PGIR shadows according to the ($\leq_{lp}$) relation determined by Algorithm 1. If two nodes are incomparable, then the compiler rejects the policy as unimplementable.

Avoiding Loops. The checks for failure safety described above overlook one critical point: A better (lower rank) path might not be available due to loops rejected by BGP.

Consider the partial product graph shown in Figure 7. Our preference inference algorithm will determine that node X_1 should be locally preferred to node X_2 since this will result in a better (lower ranked) path to destination A. However, when applying Algorithm 1, we failed to take into account the possibility of loops. In particular, node A_2 may be unusable for advertisements that go through X_1 since the advertisement may have already gone through A_1 previously. In this case, X will have made the wrong choice since preferring X_2 would have resulted in a better path for the destination A (a path of rank 3 instead of 4).

On the other hand, this would not have been a problem

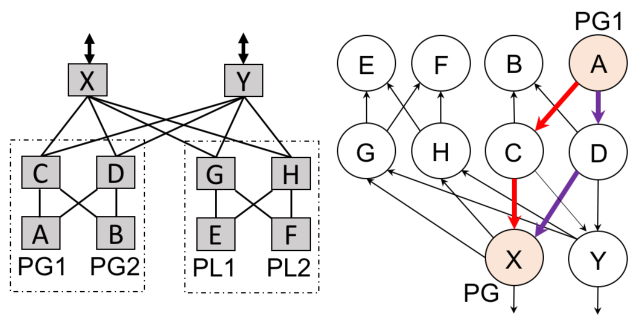


Figure 8: Aggregation safety for a datacenter.

if paths ending at A_1 had a lower rank than those ending at A_2 or A_3. For example, if paths ending at A_1 had rank 1, then any time A_2 is unusable due to a loop with A_1, it ultimately does not matter since A_1 is preferred anyway. In fact, checking if we are never worse off using A_1 instead of A_2 corresponds exactly with determining if A has a regret-free preference for A_1 over A_2. More specifically, the compiler checks that, any time there are two shadows N_1 and N_2 for the same topology location, where N_1 appears "above" (*i.e.*, can reach) N_2 in the PGIR, then N_1 must be strictly preferred to N_2 (*i.e.*, $N_1 <_{lp} N_2$).

5.4 Aggregation-safety analysis

As shown in §3, aggregation can lead to subtle blackholing of traffic when failures occur. Determining when this can happen requires knowledge, not only of the topology, but also of the policy. For instance, a policy might require all traffic for a particular prefix to go over a single link before being aggregated. If that one link fails, a black hole might be introduced. Because the PGIR encodes the complete user policy and topology, Propane can efficiently check that aggregates do not black hole traffic for up to k failures.

We view the aggregation problem as a variant of the min-cut problem in the PGIR. Specifically, for each prefix that falls under an aggregate, we are interested in finding a lower bound on the number of failures required to disconnect the prefix's origin from its aggregation point. The difficulty, however, is that each link in the topology might appear as multiple links in the PGIR, thus preventing the direct application of standard min-cut algorithms.

Instead, we adopt the following simple strategy: i) pick a random path in the PGIR between the prefix's origin and aggregation point, ii) remove all similar edges in the PGIR for each topology edge along the chosen path, and iii) repeat until no such path exists. Because each path chosen is both policy compliant and edge disjoint (due to ii), the number of paths that we are able to remove lower bounds the number of failures required to disconnect the prefix from its aggregate, subject to the policy constraints.

For example, recall the datacenter example from §3, with the policy `PG1 => end(A)`, where `PG1` falls under the `PG` aggregate. Figure 8 shows the simplified PGIR. Since the compiler knows aggregation will occur at X, and it knows that the `PG1` prefix will originate at A, we can compute the number of failures it would take to disconnect A from X. We could remove the A–D–X path first. We would then need to remove any other A–D or D–X links from the PGIR (in this case none). Next, we could remove the links along the A–C–X path, repeating the process. Because A is then disconnected from X, the compiler knows that 2 is a lower bound on the number of failures required to potentially introduce an aggregation black hole for prefix `PG1`. This process is repeated for other aggregation locations (*e.g.*, Y).

5.5 Abstract BGP

The final stage of our compiler translates policies from PGIR to a vendor-neutral abstraction of BGP (ABGP).

From PGIR to ABGP. Once we have the total ordering on node preferences in the PGIR from the failure safety analysis, the translation to ABGP is straightforward. The idea is to encode the state of the automata using BGP community values. Each router will match based on its peer and a community value corresponding to the state of the PGIR, and then update the state before exporting to the neighbors permitted by the PGIR. For example, router A in Figure 5 will allow an announcement from C with a community value for state $(3, 2)$ (and deny anything else). If it sees such an announcement, it will remove the old community value and add a new one for state $(4, 2)$ before exporting it to W.

To ensure preferred paths are always obtained, for each router r in the topology, the compiler sets a higher local preference for neighbors of a more-preferred node for r in the PGIR. For example, C will prefer an advertisement from D in state $(2, 2)$ over an advertisement from E in state $(-, 2)$.

Since the compiler can control community tagging only for routers under the control of the AS being programmed, it cannot match on communities for external ASes. Instead, it translates matches from external ASes into a BGP regular expression filter. For example, node D in Figure 5 would match the single hop external paths X or Y. In general, if routes are allowed from beyond X or Y, these will also be captured in the BGP regular expression filters. The unknown AS topology is modeled as a special node in the PGIR that generates a filter to match any sequence of ASes.

Finally, the external AS W should prefer our internal router A over B. In general, it is not possible to reliably control traffic entering the network beyond certain special cases. In this example, however, assuming our network and W have an agreement to honor MEDs, the MED attribute can influence W to prefer A over B. Additionally, the compiler can use the BGP no-export community to ensure that no other AS beyond W can send us traffic. The compiler can perform a simple analysis to determine when it can utilize BGP special attributes to ensure traffic enters the network in a particular way by looking at links in the product graph that cross from the internal topology to the external topology. Figure 9 shows the full configuration from the compilation example.

After configuration generation, the compiler further processes the ABGP policy, removing community tags when possible, combining filters, removing dead filters, and so on. In the compilation example all community tags can be removed, since there is never any ambiguity based on the

```
Router A:
  Match peer=C, comm=(3,2)
    Export comm ← (4,2),
           MED ← 80, peer ← W
Router B:
  Match peer = C, comm = (-,2)
    Export comm ← (-,3), comm ← noexport,
           MED ← 81, peer ← W
  Match peer = C, comm = (3,2)
    Export comm ← (-,3), comm ← noexport,
           MED ← 81, peer ← W
Router C:
  Match[LP=99] peer = E, comm = (-,2)
    Export comm ← (-,2), peer ← B
  Match peer = D, comm = (2,2)
    Export comm ← (3,2), peer ← A,B
Router D:
  Match regex(X + Y)
    Export comm ← (2,2), peer ← C
Router E:
  Match regex(Z)
    Export comm ← (-,2), peer ← C
```

Figure 9: Abstract BGP router configurations.

router importing the route and the neighbor the route is being imported from. Similarly, after removing the communities, the two rules at router B are merged into a single rule.

6. IMPLEMENTATION

Our Propane compiler is implemented in roughly 6700 lines of F# code. It includes command-line flags for enabling or disabling the use of the BGP MED attribute, AS path prepending, the no-export community, as well as for ensuring at least k-failure safety for aggregate prefixes. Since each prefix has a separate routing policy, we compile each routing policy in parallel. Currently, Propane supports generating Quagga router configurations out of the box. Users can add new vendor-specific adapters to translate from ABGP to other router configuration languages, or incorporate the compiler into an existing template-based system, *e.g.*, by mixing the Propane-generated BGP configuration with other, non-BGP configuration elements.

Our compiler includes the following features that improve its performance and usability.

Efficient PGIR construction. Constructing automata for extended regular expressions (*i.e.*, regular expressions with negation and intersection operations) is known to have high complexity [15]. The Propane compiler uses regular expression derivatives [31] with character classes to construct deterministic automata for extended regular expressions efficiently. Since regular expressions are defined over a finite alphabet, and since much of the AS topology is unknown, we set the alphabet to include all uniquely referenced external ASes in the policy. To model the unknown external AS topology beyond immediate peers, we include a special topology node to represent any unknown location. Rather than construct the product graph in full, our implementation prevents exploring parts of the graph during construction when no automata has a reachable accepting state.

Fast failure-safety analysis. When computing local preferences and ensuring failure safety, as described in §5, the compiler performs memoization of the Regret-Free function. That is, whenever for two states N_1 and N_2 we compute Regret-Free(G, N_1, N_2) and the function evaluates to $true$, then each of the intermediate related states m and n must also satisfy Regret-Free(G, m, n). Memoizing these states dramatically reduces the amount of work performed to find preferences in the common case.

Efficient configuration generation. The naive code generation algorithm described in §5.5 is extremely memory inefficient since it generates a separate match-export pair for every unique in-edge/out-edge pair for every node in the product graph before minimization. Our implementation performs partial minimization during generation by recognizing common cases such as when there is no restriction on exporting to or importing from neighbors.

Checking policy correctness. Even when programming the network centrally, it is possible for operators to make mistakes and write incorrect policies. Propane includes many analyses to identify common mistakes at compile time. A subset includes: (i) a preference analysis to determine when backup paths/preferences will never be used, (ii) a reachability analysis to check if locations that should be reachable according to the policy are not reachable after combining both the topology and policy, (iii) an anycast analysis to find instances where the operator might accidentally anycast a prefix (*i.e.*, originates the prefix from multiple locations), (iv) an aggregate analysis to find aggregates that do not summarize any specific prefix.

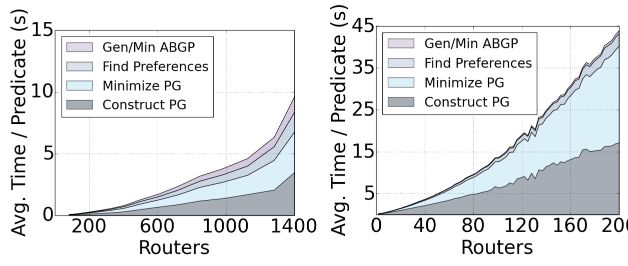
7. EVALUATION

We apply Propane on real policies for backbone and datacenter networks. Our main goals are to evaluate if its language is expressive enough for real-world policies, the time the compiler takes to generate router configurations, and the size of the resulting configurations.

7.1 Networks studied

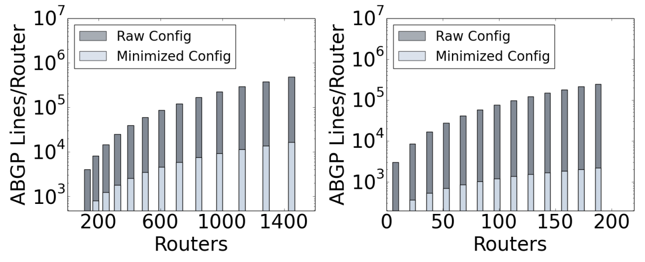
We obtained routing policy for the backbone network and datacenters of a large cloud provider. Multiple datacenters share this policy. The backbone network connects to the datacenters and also has many external BGP neighbors. The high-level policies of these networks are captured in an English document which guides operators when writing configuration templates for datacenter routers or actual configurations for the backbone network (where templates are not used because the network has a less regular structure).

The networks have the types of policies that we outlined earlier (§3). The backbone network classifies external neighbors into several different categories and prefers paths through them in order. It does not want to provide transit among certain types of neighbors. For some neighbors, it prefers some links over the others. It supports communities based on which it will not announce certain routes externally or



(a) Datacenter (b) Backbone

Figure 10: Compilation time.



(a) Datacenter (b) Backbone

Figure 11: Configuration minimization.

announce them only within a geographic region (*e.g.*, West Coast of the USA). Finally, it has many filters, *e.g.*, to prevent bogons (private address space) from external neighbors, prevent customers from providing transit to other large networks, prevent traversing providers through peers, *etc.*

Routers in the datacenter network run BGP using private AS numbers and peer with each other and with the backbone network over eBGP. The routers aggregate some prefix blocks when announcing them to the backbone network, they keep some prefixes internal, and attach communities for some other prefixes that should not traverse beyond the geographic region. The datacenter networks also have policies by which some prefixes should not be announced beyond a certain tier in the datacenter hierarchy.

7.2 Expressiveness

We found that we could translate all network policies to Propane. We verified with the operators that our translation preserved intended semantics.[3] We further found that the datacenter policies were correctly translated. For the backbone network, the operator mentioned an additional policy not present in the English document, which we added later. For both the datacenter and backbone networks, the Propane compiler was able to guarantee policy-compliance under all possible failure scenarios.

Not counting the lines for various definitions like prefix and customer groups or for prefix ownership constraints, which we cannot reveal because of confidentiality concerns, the routing policies for Propane were 43 lines for the backbone network and 31 lines for the datacenter networks.

7.3 Compilation time

We study the compilation of time for both policies as a function of network size. Even though the networks we study have a fixed topology and size, we can explore the impact of size because the policies are network-wide and the compiler takes the topology itself as an input. For the datacenter network, we build and provide as input fat tree [1] topologies of different sizes, assign a /24 prefix to each ToR switch, and randomly map prefixes to each type of prefix group with a distinct routing policy. For the backbone network, the internal topology does not matter since all routers connect to each other through iBGP. We explore different (full iBGP) mesh sizes and randomly map neighboring networks to routers. Even though each border router connects to many external peers, we count only the mesh size.

All experiments are run on an 8 core, 3.6 GHz Intel Xeon processor running Windows 7. Figure 10 shows the compilation times for datacenter and backbone networks of different sizes. For both policies, we measure the mean compilation time per prefix predicate since the compiler operates on predicates in parallel. A single predicate can describe many prefixes, for example by matching on a disjunction of prefixes. At their largest sizes, the per-predicate compilation time is roughly 10 seconds for the datacenter network and 45 seconds for the backbone network.

Total compilation for the largest datacenter is less than 9 minutes total. Unlike the datacenter policy, the number of prefixes for the backbone policy remains relatively fixed as the topology size increases. Compilation for the largest backbone network, takes less than 3 minutes total. The inclusion of both more preferences and more neighboring ASes in the backbone policy increases the size of the resulting PGIR, which in turn leads to PGIR construction and minimization taking proportionally more time.

In both examples, we observe that Algorithm 1 for inferring Regret-Free preferences is efficient, taking only a small fraction of the total running time. PGIR minimization is the most expensive compilation phase. If needed, minimization can be limited to a fixed number of iterations for large networks. Both the backbone and datacenter policies could be successfully compiled without performing minimization.

7.4 Configuration size

Figure 11 shows the size of the compiled ABGP policies as a function of the topology size. The naive translation of PGIR to ABGP outlined in §5 generates extremely large ABGP policies by default. To offset this, the compiler performs ABGP configuration minimization both during and after the PGIR to ABGP translation phase. Minimization is highly effective for both the datacenter and backbone policies. In all cases, minimized policies are a small fraction of the size of their non-minimized counterparts.

However, even minimized configurations are hundreds or

[3]Not intended as a scientific test, but we also asked the two operators if they would find it easy to express their policies in Propane. The datacenter operator said that he found the language intuitive. The backbone operator said that formalizing the policy in Propane seemed equally easy or difficult as formalizing in RPSL [2], but he appreciated that he would have to do it only once for the whole network (not per-router) and did not have to manually compute various local preferences, import-export filters, and MEDs.

thousands of lines per router. For the backbone network, the size of Propane configurations is roughly similar to the BGP components of actual router configurations, though qualitative differences exist (see below). We did not have actual configurations for the datacenter network; they are dynamically generated from templates.

7.5 Propane vs. operator configurations

We comment briefly on how Propane-generated configurations differ from configurations or templates written by operators. In some ways, Propane configurations are similar. For example, preferences among neighboring ASes are implemented with a community value to tag incoming routes according to preference, which is then used at other border routers to influence decisions.

In other ways, the Propane configurations are different, relying on a different BGP mechanism to achieve the same result. Some key differences that we observed were:

i) operators used the no-export community to prevent routes from leaking beyond a certain tier of the datacenter, while Propane selectively imported the route only below the tier;

ii) operators prevented unneeded propagation of more-specific route announcements from a less-preferred neighboring AS based on their out-of-band knowledge about the topology, whereas Propane propagated these advertisements;

iii) operators used a layer of indirection for community values, using community groups and re-writing values, to implement certain policies in a more maintainable manner, where Propane uses flat communities; and

iv) operators used BGP regular expression filters to enforce certain invariants that are independent of any particular prefix, whereas Propane enforced these invariants per prefix.

We are investigating if such differences matter to operators, *e.g.*, if they want to read Propane configurations, and, if necessary, how to reduce them.

8. RELATED WORK

Our work draws on four threads of prior work.

SDN languages. Propane was heavily influenced by SDN programming languages such as NetKAT [3], Merlin [33], FatTire [32], as well as path queries [29]. Each of these languages is oriented around regular expressions, which describe paths through a network, and predicates, which classify packets. In particular, FatTire allows programmers to define sets of paths together with a fault tolerance level (*i.e.*, tolerate 1 or 2 faults) and the compiler generates appropriate OpenFlow rules. Propane is more expressive as it allows users to specify preferences among paths, and it generates distributed implementations that tolerate any number of faults. Because FatTire generates data plane rules up front, specifying higher levels of fault tolerance comes at the cost of generating additional rules that tax switch memory. In contrast, Propane relies on distributed control plane mechanisms to react to faults, which do not have additional memory cost. Because of the differences in the underlying technology, the analyses and compilation algorithms used in Propane are quite different from previous work on SDN. Finally, in addition to using path-based abstractions for intra-domain routing, Propane uses them for inter-domain routing as well, unlike existing SDN languages.

Configuration automation. Many practitioners use configuration templates [19, 34], to ensure certain kinds of consistency across similar devices. In addition, configuration languages such as RPSL [2], Yang [7], and Netconf [9] allow operators to express routing policy in a vendor-neutral way. However, all of these solutions remain low-level, for example, requiring operators to specify exact local preferences. Unlike Propane, there is no guarantee that these low-level configurations satisfy the original, high-level intent.

Configuration analysis. The notion that configuring network devices is difficult and error-prone is not new. In the past, researchers have tried to tackle this problem by analyzing existing firewall configurations [24, 36, 30] and router configurations [11, 10, 28, 13, 35, 16] and reporting errors or inconsistencies when they are detected. Our research is complementary to these analysis efforts. We hope to eliminate bugs by using higher-level languages and a "correct-by-construction" methodology. By proposing that network administrators write configurations at a high-level of abstraction, a whole host of low-level errors can be avoided and policy implementation can be simplified.

Configuration synthesis. ConfigAssure [26, 27] is another system designed to help users define and debug low-level router configurations. Inputs to ConfigAssure include a *configuration database*, which contains a collection of tuples over constants and configuration variables, and a *requirement*, which is a set of constraints. The authors use a combination of logic programming and SAT solving to find concrete values for configuration variables. ConfigAssure handles configuration for a wide range of protocols and many different concerns. In contrast, the scope of Propane is much narrower. In return, Propane offers compact, higher-level abstractions customized for our domain, such as regular paths, as well as domain-specific analyses customized to those abstractions, such as our failure safety analysis. The implementation technology is also entirely different, as we define algorithms over automata and graphs as opposed to using logic programming and SAT-based model-finding.

9. CONCLUSIONS

We introduced Propane, a language and compiler for implementing network-wide policies using a distributed set of devices running BGP. Propane allows operators to describe their policy naturally through high-level constraints on both the shape and relative preferences of paths for different types of traffic. When Propane compiles a policy, the resulting BGP configurations are guaranteed to faithfully implement the centralized policy in a purely distributed fashion, regardless of any number of network failures. Applying Propane to real-world networks showed that its language is expressive and its compiler is scalable.

Acknowledgements. We thank R. Aditya, George Chen, and Lihua Yuan for feedback on the work and the SIGCOMM reviewers for comments on the paper. This work is supported in part by the National Science Foundation award CNS-1161595.

10. REFERENCES

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

[2] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra. Routing policy specification language (rpsl). RFC 2622, RFC Editor, June 1999.

[3] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. NetKAT: Semantic foundations for networks. In *POPL*, January 2014.

[4] M. Anderson. Time warner cable says outages largely resolved, August 2014.

[5] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, et al. ONOS: towards an open, distributed SDN OS. In *Proceedings of the third workshop on Hot topics in software defined networking*, pages 1–6. ACM, 2014.

[6] News and press | BGPmon. http://www.bgpmon.net/news-and-events/. Retrieved 2016-01-26.

[7] M. Bjorklund. Yang - a data modeling language for the network configuration protocol (netconf). RFC 6020, RFC Editor, October 2010. http://www.rfc-editor.org/rfc/rfc6020.txt.

[8] M. Casado, M. J. Freedman, J. Pettit, J. Luo, N. McKeown, and S. Shenker. Ethane: Taking control of the enterprise. In *SIGCOMM*, 2007.

[9] R. Enns, M. Bjorklund, J. Schoenwaelder, and A. Bierman. Network configuration protocol (netconf). RFC 6241, RFC Editor, June 2011. http://www.rfc-editor.org/rfc/rfc6241.txt.

[10] N. Feamster. *Proactive Techniques for Correct and Predictable Internet Routing*. PhD thesis, Massachusetts Institute of Technology, 2005.

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

[12] N. Feamster, J. Winick, and J. Rexford. A model of bgp routing for network engineering. In *in Proc. ACM SIGMETRICS*, 2004.

[13] 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*, 2015.

[14] N. Foster, M. J. Freedman, A. Guha, R. Harrison, N. P. Katta, C. Monsanto, J. Reich, M. Reitblatt, J. Rexford, C. Schlesinger, A. Story, and D. Walker. Languages for software-defined networks. *IEEE Communications Magazine*, 51(2):128–134, 2013.

[15] W. Gelade and F. Neven. Succinctness of the complement and intersection of regular expressions. 2008.

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

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

[18] T. G. Griffin and G. Wilfong. On the correctness of ibgp configuration. In *SIGCOMM*, 2002.

[19] Hatch – create and share configurations. http://www.hatchconfigs.com/. Retrieved 2016-01-26.

[20] P. Lapukhov, A. Premji, and J. Mitchell. Use of BGP for routing in large-scale data centers. Internet draft, 2015.

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

[22] T. Lengauer and R. Tarjan. A fast algorithm for finding dominators in a flowgraph. In *ACM Trans. Program. Lang. Syst.*, January 1979.

[23] R. Mahajan, D. Wetherall, and T. Anderson. Understanding BGP misconfiguration. In *SIGCOMM*, pages 3–16, 2002.

[24] A. Mayer, A. Wool, and E. Ziskind. Fang: A firewall analysis engine. In *IEEE Symposium on Security and Privacy*, pages 177–187, 2000.

[25] J. McCauley, A. Panda, M. Casado, T. Koponen, and S. Shenker. Extending sdn to large-scale networks. In *Open Networking Summit*, 2013.

[26] S. Narain. Network configuration management via model finding. In *Proceedings of the 19th Conference on Systems Administration*, pages 155–168, 2005.

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

[28] S. Narain, R. Talpade, and G. Levin. *Guide to Reliable Internet Services and Applications*, chapter Chapter on Network Configuration Validation. Springer, 2010.

[29] S. Narayana, J. Rexford, and D. Walker. Compiling path queries in software-defined networks. In *HotSDN*, pages 181–186, 2014.

[30] T. Nelson, C. Barratt, D. J. Dougherty, K. Fisler, and S. Krishnamurthi. The margrave tool for firewall analysis. In *USENIX Large Installation System Administration Conference*, 2010.

[31] S. Owens, J. Reppy, and A. Turon. Regular-expression derivatives re-examined. In *J. Funct. Program.*, March 2009.

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

[33] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. *CoRR*, abs/1407.1199, 2014.

[34] configuration templates | thwack. https://thwack.solarwinds.com/search.jspa?q=configuration+templates. Retrieved 2016-01-26.

[35] K. Weitz, D. Woos, E. Torlak, M. D. Ernst, A. Krishnamurthy, and Z. Tatlock. Getting started with Bagpipe. http://www.konne.me/bagpipe/started.html, 2015.

[36] A. Wool. Architecting the lumeta firewall analyzer. In *USENIX Security Symposium*, 2001.