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

Ryan Beckett
Princeton University
rbeckett@princeton.edu

Ratul Mahajan
Microsoft Research
ratul@microsoft.com

Todd Millstein
University of California, Los Angeles
todd@cs.ucla.edu

Jitu Padhye
Microsoft Research
padhye@microsoft.com

David Walker
Princeton University
dpw@princeton.edu

**Abstract—** We develop a system called Propane to help network operators with a challenging and error prone task: bridging the gap between network-wide routing objectives and low-level configurations of devices that run complex distributed protocols. Propane allows operators to specify their objectives naturally, describing high-level constraints and preferences on the types of paths that traffic should take through their network or other networks. Our compiler automatically translates these specifications to a collection of BGP router configurations. We introduce data structures and algorithms that guarantee that the compiled, distributed configurations correctly implement the routing policy under all possible combinations of failures. We use Propane to specify and compile the configurations of real data center and backbone networks. We show that, despite its strong guarantees, it scales to topologies with hundreds to thousands of routers.

## 1. INTRODUCTION

It is well known that configuring networks is error prone and such errors can lead to disruptive network downtimes [22, 11, 13, 16]. 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 therefore 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 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, many practitioners have adopted a template-based approach [18, 33], in which common tasks are captured as parameterized templates. 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.

Configuration analysis tools [13, 11] have been developed as a complementary approach for reducing misconfigurations by checking that the low-level configurations match high-level operator intent. However, such tools cannot help operators come up with the configuration in the first place. Further, today's tools can only check correctness under concrete failure scenarios, rather than under all possible failures.

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 interacting controllers, thus bringing back distributed control planes [24, 5] and their current programming difficulties.

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

1. Design a new, high-level language with natural abstrac-

tions 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 [31], Merlin [32], 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 directly 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 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.

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

[1]In this paper we assume that BGP is the only control plane protocol running in the network or the other protocols are correctly configured and do not have adverse interactions with BGP [17, 12].

## 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 color routes on different criteria (e.g., "entered on West Coast") and then use the color 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).

In implementing their policy, each network operator assumes that neighboring ASes correctly implement BGP and honor contracts related to MEDs and communities. Propane makes the same assumption when deriving BGP configura-

tions for a network.

The combination of arbitrary import and export filters and route selection policies at individual routers make BGP a highly flexible routing protocol. That flexibility, however, comes at the cost of it being difficult to configure correctly.

## 3. MOTIVATION

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

### 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 the 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 P1 of the policy, 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 properly will 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. 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 properly configure even this small example network manually; devising correct configurations for real networks can quickly become a nightmare. Such networks have many 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.

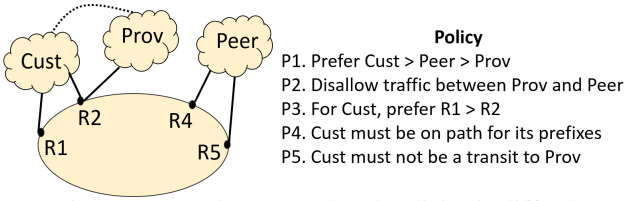



Figure 1: Creating router-level policies is difficult.

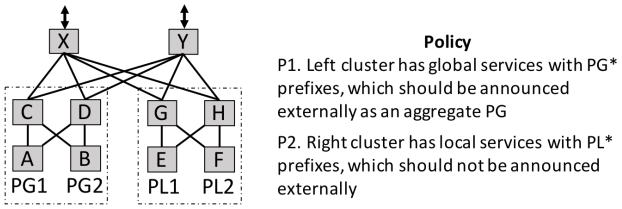



Figure 2: Policy-compliance under failures is difficult.

### 3.2 Example 2: The Data Center

While configuring policies for a fully functional network is difficult, ensuring policy compliance in the face of failures can be almost impossible. Consider the data center 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 decide that a simple way to implement the policy is to have X and Y: $i$) not export externally what they hear from G and H, routers that belong to the local services cluster; and $ii$) announce 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 be done independently (e.g., new prefixes can be assigned to local services without updating router configurations).

However, this implementation does not have the right 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 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 used for other services outside of the data center). This problem does

[2]For scale and policy flexibility, data center networks increasingly use BGP internally, with a private AS number per router [19].

not manifest without failures because then X has and prefers paths to PL* through G and H since they are shorter. A similar problem will happen if links Y–G and Y–H fail. Link failures in data centers are frequent and it is not uncommon to have multiple failed links at a given time [16].

To avoid this problem, the operator may decide to 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 blackhole [20]. If links D–A and X–C fail, X will hear announcements for PG2 from D and will thus announce PG externally. This announcement will bring to X traffic for PG1 as well, but because of valley-free filtering, X does not have a valid route for PG1 and will thus drop all traffic to it.

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 that is 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 blackholes.

# 4. PROPANE OVERVIEW

Policies for (distributed) control planes differ from data-plane policies in a few important ways. First, control-plane policies 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 a notion of *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.

In the rest of this section, we define Propane policies for the two examples from the previous section, introducing key aspects of the Propane language along the way. The complete syntax of the language as well as our strategy for compiling Propane policies to BGP is defined in §5.

## 4.1 Example 1: The Backbone

Propane allows operators to 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 different routes traffic may (or may 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 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** predicate, as well as other path predicates 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 predicates 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 by default routing allows transit traffic (*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 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 above 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`. These constraints capture the entire policy. From them, Propane 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 Data Center

In our data center, there are three main concerns: (1) traffic for the prefix block allocated to each top-of-rack router must be able to reach that router, (2) local services must not leak outside the data center, and (3) aggregation must be performed on global prefixes to reduce churn in the network.

Propane allows us to specify each of these constraints modularly. The first constraint is about prefix ownership—namely, we only want traffic 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)}
```

In English: traffic for prefix `PG1` is only allowed to follow paths that end at router `A`; traffic matching `PG2`, but not `PG1`, must end at router `B`; and so on. Any traffic destined for a prefix that not a part of the data center should be allowed to leave the data center and end at some external location, which is otherwise unconstrained. The special keyword **`out`** here matches any location outside the user's network, while the keyword **`in`** will match any location inside the network. For the second constraint, we can define another policy:

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

The locality constraint ensures that traffic for local prefixes only ever follows paths that are internal to the network at each hop. This constraint guarantees that the services remain reachable only to locations inside the data center.

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 using an appropriate prefix such as `74.3.28.0/24`) when advertised outside the datacenter.

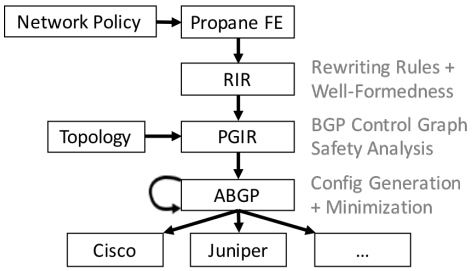


Figure 3: Compilation pipeline stages for Propane.

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

Once Propane compiles the policy, it is guaranteed to be compliant under all possible failure scenarios, modulo aggregation black holes (which are unavoidable in general). 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 front-end (FE) of Propane simplifies operators' task of describing preferred paths. That simplicity, however, comes at the cost of compilation complexity. The compiler must efficiently compute the sets of paths represented by the intersection of preferences and topology, determine which ones can be honored under failures, and ensure policy compliance under all possible failure cases.

We handle these challenges by breaking compilation into multiple stages and developing efficient algorithms for each (See Figure 3). 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 for well-formedness (*e.g.*, never constraining traffic that does not enter the network), before being combined with the network 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 that the PGIR is safe, it translates it to an abstract BGP (ABGP) representation. Vendor-specific adaptors to translate from ABGP to concrete device configurations or to integrate with an existing template-based system are easily added. The Propane compiler currently generates Quagga router configurations by default.

## 5.1 Regular IR (RIR)

## Syntax

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

## Propane Expansions

| | | |
|---|---|---|
| **any** | = | $\texttt{out}^* \cdot \texttt{in}^+ \cdot \texttt{out}^*$ |
| **none** | = | $\emptyset$ |
| **internal** | = | $\texttt{in}^+$ |
| **only**$(X)$ | = | $\texttt{any} \cap X^*$ |
| **never**$(X)$ | = | $\texttt{any} \cap (!X)^*$ |
| **through**$(X)$ | = | $\texttt{out}^* \cdot \texttt{in}^* \cdot X \cdot \texttt{in}^* \cdot \texttt{out}^*$ |
| **later**$(X)$ | = | $\texttt{out}^* \cdot (X \cap \texttt{out}) \cdot \texttt{out}^* \cdot \texttt{in}^+ \cdot \texttt{out}^*$ |
| **before**$(X)$ | = | $\texttt{out}^* \cdot \texttt{in}^+ \cdot \texttt{out}^* \cdot (X \cap \texttt{out}) \cdot \texttt{out}^*$ |
| **end**$(X)$ | = | $\texttt{any} \cap (\Sigma^* \cdot X)$ |
| **start**$(X)$ | = | $\texttt{any} \cap (X \cdot \Sigma^*)$ |
| **exit**$(X)$ | = | $(\texttt{out}^* \cdot \texttt{in}^* \cdot (X \cap \texttt{in}) \cdot \texttt{out} \cdot \texttt{out}^*) \cup$ $(\texttt{out}^* \cdot \texttt{in}^+ \cdot (X \cap \texttt{out}) \cdot \texttt{out}^*)$ |
| **enter**$(X)$ | = | $(\texttt{out}^* \cdot \texttt{out} \cdot (X \cap \texttt{in}) \cdot \texttt{in}^* \cdot \texttt{out}^*) \cup$ $(\texttt{out}^* \cdot (X \cap \texttt{out}) \cdot \texttt{in}^+ \cdot \texttt{out}^*)$ |
| **link**$(X, Y)$ | = | $\texttt{any} \cap (\Sigma^* \cdot X \cdot Y \cdot \Sigma^*)$ |
| **path**$(\vec{X})$ | = | $\texttt{any} \cap (\Sigma^* \cdot X_1 \ldots X_n \cdot \Sigma^*)$ |
| **novalley**$(\vec{X})$ | = | $\texttt{any} \cap$ $!\texttt{path}(X_2, X_1, X_2) \cap \cdots \cap$ $!\texttt{path}(X_n, X_{n-1}, X_n)$ |

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

**Syntax**. The Propane FE is just a thin layer atop a core, regular-expression language (RIR) for describing preference-based path constraints. Figure 4 shows the RIR syntax. A policy consists of one or more constraints, each of which have a test on the type of route, and a corresponding set of preferred regular paths. Regular paths are regular expressions where the base characters are abstract locations—either a router or an AS. Special **in** and **out** symbols refer to any internal or external location respectively. In addition, $\Sigma$ refers to any symbol. We also use the standard regular expressions abbreviation $r^+$ to stand for $r \cdot r^*$, a sequence of one or more occurrences of $r$. Predicates ($t$) consist of logical boolean connectives (and, or, not) as well as tests that match a particular prefix (or group of prefixes) and tests for route advertisments with a particular community value (i.e., an integer value associated with a path).

Propane also supports constraints purely on the control-plane behavior of BGP. For example, prefix aggregation is an important optimization to reduce routing table size and churn in practice. A constraint of the form $agg(x, l)$ tells the compiler to perform aggregation for prefix $x$ across all links described by $l$. 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 $tag(c, t, l)$ adds community tag $c$ for any prefixes matching $t$ across links $l$. We list only the route aggregation and community tagging constraints in Figure 4, but we also support other constraints such as limiting the maximum number of routes allowed between ASes, or enabling BGP multipath.

**Semantics**. The semantics of RIR is in terms of ranked paths. Each preference-based regular path constraint (of the form $r_1$»…»$r_j$) maps to a set of concrete paths in the network that match one of $r_i$. We denote a network path as a string of abstract locations (routers or external ASes) of the form: $n_1 n_2 \ldots n_k$. 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, and $p$ is a topologically-valid path. We denote the length of a path $p$ as $|p|$. A path $p$ will have a rank:

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

where the rank is lexicographically ordered according to (1) the most preferred regular expression matched, and (2) as a tie breaker, the path length. Lower ranks indicate *more* preferred paths. Traffic may be sent on any of the most preferred paths for each pair of starting and ending locations that appear in some specified path. There is an implicit lowest preference to drop traffic: $\emptyset$ when no other route is possible.

The set of ranked paths depends on which paths are valid in the topology, and thus when failures occur, the most preferred routes change. The Propane compiler ensures that generated configurations for a policy always achieve the most preferred path possible given the failures in the topology, using only distributed mechanisms.

**Propane FE to RIR**. 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 to ensure that all specified routes are well-formed. In particular, each regular path constraint $r$ must satisfy $r \subseteq \texttt{out}^* \cdot \texttt{in}^+ \cdot \texttt{out}^*$. This ensures that the user only control traffic that goes through the user's network at some point, and that such traffic does not loop back multiple times to the user network.

The translation from Propane 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 ($a$`&`$b$) is replaced by intersection on regular constraints ($a \cap b$), logical disjunction is replaced by union,

and logical negation ($!\,a$) is replaced by ($\texttt{any} \cap !\,(a)$), where `any` ensures the routes are well-formed. For example, in the data center 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 $a \cdot (b » c) \cdot d$ is lifted to $(a \cdot b \cdot d) » (a \cdot c \cdot d)$ by distributing the preference over the sequence operator. In general, we employ the follwing distributivity equivalences:

$$x \odot (y_1 » \ldots » y_n) = (x \odot y_1) » \ldots » (x \odot y_n)$$
$$(y_1 » \ldots » y_n) \odot x = (y_1 \odot x) » \ldots » (y_n \odot x)$$

where $\odot$ stands for an arbitrary regular binary operator. Preferences nested under a unary operator, *star* or *negation*, are flagged by the compiler as invalid policies.

## 5.2 Product Graph IR

Now that the user policy exists in a simplified form, we must consider 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. Our PGIR captures these dependencies by "intersecting" each of the regular automata corresponding to the RIR path preferences, and the topology. Paths through the PGIR correspond to real paths through the topology that satisfy the user constraints.

**Formal definition**. The RIR policy is an ordered sequence of regular expressions $r_1 » \ldots » r_j$. While paths talk about 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 $r_i$, we construct a deterministic finite state machine on the reversed regular expression. Each automaton is a tuple: $(\Sigma, Q_i, F_i, q_{0_i}, \sigma_i)$. The alphabet $\Sigma$ consists of all all abstraction locations (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 $E \colon 2^{V \times V}$. 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 $E' \colon 2^{V' \times V'}$, 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 preference ranks. For a vertex $v' = (v, \ldots) \in V'$, we say $loc(v') = v$. Two nodes $x$ and $y$ *shadow* each other in the product graph when $x \in V'$ and $y \in V'$ and $loc(x) = loc(y)$ but $x \neq y$. That is, $x$ and $y$ shadow each other when they are different nodes in the PGIR that represent the same topology location.

**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 $v_1 = (x, a_1, \ldots, a_m)$ to $v_2 = (y, b_1, \ldots, b_m)$ whenever $\sigma_i(a_i, y) = b_i$ for each $i$ and $(x, y) \in E$ is a valid topology link. Additionally, we add edges from the start node $s$ to any $v = (x, a_1, \ldots, a_m)$ when $\sigma_i(q_{0_i}, x) = a_i$ for each $i$. The preference function $P$ for node $v = (x, a_1, \ldots, a_m)$ is defined as $P(v) = \{i \mid a_i \in F_i\}$. That is, it records which preferences are achieved from each policy automaton for every node in the product construction. Finally, there is an edge from each node in the PGIR such that $P(v) \neq \emptyset$ to the special end node $e$.

Intuitively, PGIR tracks which states of each automaton the policy is in as route announcements move between routers. Consider the topology in Figure 5. Suppose we want a primary route from W that enters the network from $A$, and utilizes the A–C and C–D links. We also want a backup route that enters from $B$, and utilizes the B–C link, but is otherwise unconstrained. For simplicity, we assume that the route can end in either $X$, $Y$, or $Z$. The RIR for the policy is:

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

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 it corresponds to a concrete path in the topology. In particular, every path through the PGIR that ends at a node $v$ such that the preference function $P(v) = \{i_1, \ldots, i_m\}$ is non-empty, is a valid topological path that satisfies the policy constraints and results in a particular path with preferences $i_1$ through $i_m$. 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 the most prefered preference 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 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 can not accept the current path or any extension of it. Since node $(W, 5, -)$ is in an accepting state for the first automata, it indicates this path has preference 1.

**Minimization**. Before checking if policies captured by the PGIR are safe, we minimize the PGIR to improve our safety analysis. The minimization is based on the observation that,

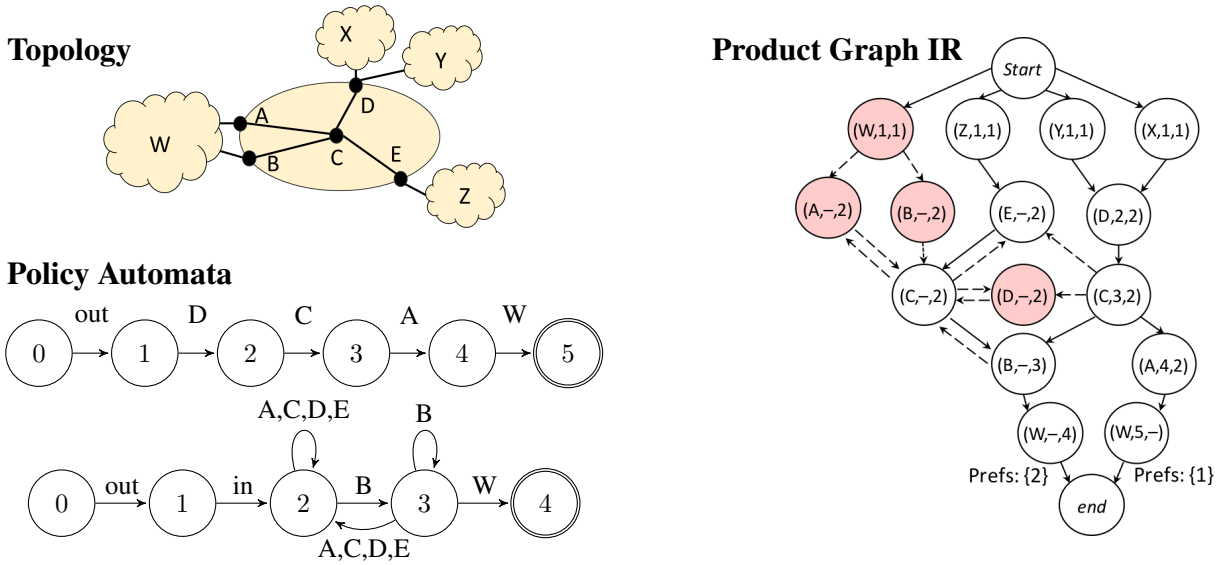



Figure 5: Example Product Graph IR construction for policy: $(\texttt{W} \cdot \texttt{A} \cdot \texttt{C} \cdot \texttt{D} \cdot \textbf{out}) \texttt{»} (\texttt{W} \cdot \texttt{B} \cdot \textbf{in}^{+} \cdot \textbf{out})$.

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 for which it 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 1 policy, but which contains a loop. It often improves the precision of the failure-safety analysis to eliminate cases that are not possible due to loops. We use graph dominators [21] as a cheap approximation 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 $x$ dominates a node $y$ if $x$ appears on every path leading to $y$ from the start node. For example, node $(W, 1, 1)$ in Figure 5 is never on a simple path to the end node since it must go through another $W$ in either case. Colored nodes and dashed edges are removed after minimization since they are irrelvant to the BGP decision process.

**Failure Safety**. 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-route routing policy difficult, particularly in the presence of failures. For example, imagine an extremely simple policy for the topology in Figure 5, which says to prefer one route over another: $(\texttt{W} \cdot \texttt{A} \cdot \texttt{C} \cdot \texttt{D}) \texttt{»} (\texttt{W} \cdot \texttt{B} \cdot \texttt{C} \cdot \texttt{E})$.

How could such a policy be implemented in BGP? Suppose we set the local preference at router $C$ to prefer $D$ over $E$, and use the BGP MED attribute to influence $W$ to prefer $A$ over $B$. This might work as expected under normal conditions, however, if the A – C link fails, then suddenly $C$ has made the wrong decision. Traffic will now follow the $W \cdot B \cdot C \cdot D$ path, even though this path was not specified by the policy. This is a violation of *soundness*: The distributed implementation has used a route not specified by the policy. To make matters worse, the second preference $W \cdot B \cdot C \cdot E$ is available in the network but not being used. This is a violation of *completeness*: A path for the best possible route available exists in the network but is not being used by the distributed implementation. We could solve the first problem by filtering routes that go through $D$ at $B$, however, we can not fix the second problem—$C$ has made the wrong choice. In fact, this policy can not be implemented in BGP in a way that is policy compliant under all failures.

At a first cut, any time a router must make a decision locally between several route options, there is the possibility it might choose incorrectly. The PGIR captures this notion of choice precisely. For example, router $D$ in Figure 5 appears only once in the PGIR, possibly receiving announcements from $X$ or $Y$. However, regardless of whether $D$ chooses a route from $X$ or $Y$, the set of paths allowed by the policy after going through $D$ will be the same in either case. This remains true despite any failures that might have occurred in the network. Thus $D$ can safely prefer $X$ and $Y$ equally.

The more challenging case is when a topology location occurs in mutliple contexts in the PGIR. For example, in the compilation example from Figure 5, the topology node $C$ can receive an announcement from $E$ and later achieve the backup path, or it can receive an announcement from its neighbor $D$ and later achieve either the primary or backup path. Is it safe for $C$ to prefer its neighbor $D$ over its other neighbor $E$? The important observation is that, if $C$ prefers $D$, then it is never worse off—it will always achieve at least as good a path as if it had choosen $E$. For example, suppose $C$ chooses a route from $D$, but cannot achieve its primary path because the $A$ – $W$ link has failed. In this case, the advertisement from $C$ will still be sent along towards the ultimate backup location $W$. Since the $(C, 3, 2)$ node has the same one-step and two-step next hops as node $(C, -, 2)$ in the product graph, no possible failure will prevent a route advertisment from reaching $(W, -, 4)$ that wouldn't have otherwise prevented it if $C$ had choosen $E$.

**Algorithm 1** Failure Protection

```
1: procedure PROTECT(G1, N1, G2, N2)
2:     if loc(N1) ≠ loc(N2) then return false
3:     q ← Queue()
4:     q.Enqueue(N1, N2)
5:     while !q.Empty() do
6:         (n1, n2) ← q.Dequeue()
7:         for x in adj(G2, n2) do
8:             if (∃y ∈ adj(G1,n1), shadows x) or
9:                (∃y ∈ G1, dominates n1, shadows x) then
10:                if (x, y) not marked then
11:                    mark (x, y) as seen
12:                    q.Enqueue(x, y)
13:            else return false
       return true
```

In general, ensuring that an individual router's preferences respect the policy under all possible failures is hard, and enumerating all failures is intractable. We thus adopt a conservative analysis based on the observations above. The high-level idea is to order each PGIR node according to a *can prefer* relation. For example, node $(C, 3, 2)$ can be preferred to node $(C, -, 2)$, but not the other way around. Intuitively, a node $N_1$ can be preferred to another node $N_2$ when, for each preference $N_2$ might achieve, there is a better preference that $N_1$ will achieve regardless of failures. Formally this means that $N_1$ can be preferred to $N_2$ when:

$$\forall i, \exists j, j \leq i \wedge protect(G_j, N_1, G_i, N_2)$$

where *protect* means that, from $N_1$ on the product graph restricted to nodes that can potentially achieve preference $j$ or better, there is always a path to each ending location whenever this is a path from $N_2$ restricted to $G_i$.

Algorithm 1 defines what it means for one node to *protect* against the failures of another. It walks from nodes $N_1$ and $N_2$ in $G_j$ and $G_i$ respectively, and ensures that for every *step* $N_2$ can take to some new topology location, $N_1$ can, at the very least, take an equivalent step. When there is no equivalent step, the algorithm checks to see if there is an equivalent dominator (a topology location the advertisement must have passed through earlier) that can protect against failures instead. Thus any time $N_2$ can get to a final location for preference $i$, $N_1$ can get to at least the same final locations for a better preference $j$. The algorithm terminates since the number of related states $(x, y)$ that can be expolored is finite.

For each router in the topology, local preferences are now obtained by sorting the corresponding PGIR nodes according to the *can prefer* relation. If two nodes are incomparable, the compiler rejects the policy as unimplementable.

**Avoiding Loops**. The checks for failure safety described above overlook one critical point: Paths for a higher preference might not exist due to loops. To avoid this situation, the compiler checks that, when node $x$ protects against failures for node $y$, its does so without using any nodes that shadow another node that appears "above" $x$ in the PGIR.

**Aggregation-Induced Black Holes**. Because the Propane compiler has the complete user policy and topology, it is also able to efficiently check that aggregates do not black hole traffic for up to $k$ failures. The algorithm for aggregation failure safety is detailed in Section A of the appendix.

## 5.3 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 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 automaton 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 is only able to 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 modelled 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 6 shows the full configuration from the compilation example.

After configuration generation, the compiler further processes the ABGP policy, removing community tags when

```
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
```

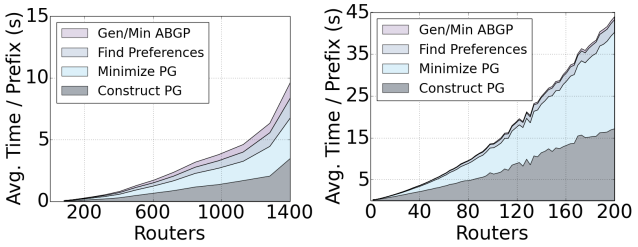
Figure 6: Abstract BGP router configurations.

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 importing the route, and the neigbor 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

We have written a prototype compiler for Propane that is implemented in roughly 5500 lines of F# code. The compiler 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 user-specified aggregates. Since each prefix has a separate routing policy, the Propane compiler is able to compile each routing policy in parallel. The current compiler produces only ABGP policies, which can be incorporated into an existing template-based system, for example by mixing the Propane configuration with other configuration components, or by generating vendor-specific configurations.

**PG construction**. Constructing automata for extended regular expressions (regular expressions including negation and intersection) are known to have high complexity [15]. The Propane compiler uses regular expressions derivatives [30] with character classes to construct deterministic automata for extended regular expressions over large alphabets 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. Similarly, 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 when no accepting state is reachable in any of the corresponding regular automata during construction.




(a) Data center (b) Backbone network

Figure 7: Compilation time.

**Failure Safety**. When computing local preferences and failure safety, as described in Section 5, the compiler performs memoization of the *protect* function. That is, whenever for two states $n_1$ and $n_2$ we compute $protect(G_j, n_1, G_i, n_2)$, if the function evaluates to $true$, then each of the intermediate related states $x$ and $y$ must also satisfy $protect(G_j, x, G_i, y)$. Memoizing these states significantly cuts down the amount of work that needs to be in the common case.

**Configuration generation**. The naive code generation algorithm described in Section 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 where there is no restriction on exporting to or importing from neighbors.

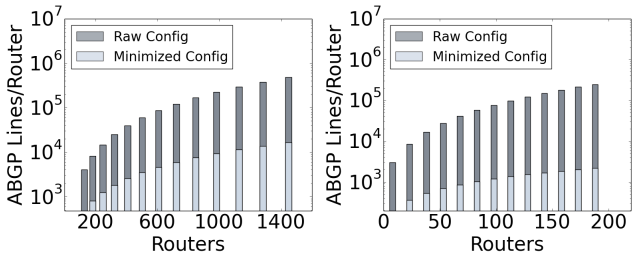
# 7. EVALUATION

We apply Propane on real policies for backbone and data center networks. Our mains goals are to evaluate if its front-end is expressive enough for real-world policies and the time the compiler takes to generate router configurations.

## 7.1 Networks studied

We obtained routing policy for the backbone network and for the data centers of a large cloud provider. Multiple data centers share this policy. The backbone network connects to the data centers 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 data center routers or actual configurations for the backbone network (where templates are not used because the network has less regular structure).

The networks have the type of policies that we outline 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 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,




(a) Data center (b) Backbone network

Figure 8: Configuration minimization.

prevent customers from providing transit to other large networks, prevent traversing providers through peers, etc.

All routers in the datacenter network run BGP using a private AS number and peer with each other and with the backbone network over eBGP. The routers aggregate prefixes 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. They 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 found that the data center policies were correctly translated. For the backbone network, the operator mentioned an additional policy that was not present in the English document, which we added later.

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 Propane policies were 43 lines for the backbone network and 31 lines for the data center 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 our converted policies are network-wide and the compiler takes topology itself as an input. For the data center 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. We take this approach to smoothly explore different sizes.

For the backbone network, the internal topology does not matter since all routers connect in a full iBGP mesh. We explore different 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 7 shows the compilation times for data centers (a), and backbone networks (b) of different sizes. For both policies, we measure the mean compilation time per prefix since the compiler operates on each prefix in parallel. At their largest sizes, the per-prefix compilation time is roughly 10 seconds for the data center network and 45 seconds for the backbone network.

Total compilation for the largest data center is less than 9 minutes total. Unlike the data center 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 more preferences in the backbone policy increases the size of the PGIR, which leads to PGIR construction and minimization taking proportionally more time.

## 7.4 Configuration size

Figure 8 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 data center 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 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 data center network; they are dynamically generated from templates.

## 7.5 Propane vs. operator configurations

Finally, we comment briefly on how Propane-generated configurations differ from configurations or templates generated by operators. In some cases, 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 cases, 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 neighboring AS based

[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 data center 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.

on their out-of-band knowledge about the topology, whereas Propane propagated these advertisements; and

$iii$) 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.

We are currently 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

*SDN Languages.* Propane was heavily influenced by recent SDN programming languages such as NetKAT [3], Merlin [32], FatTire [31], as well as path queries [28]. 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 will generate 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 uses traditional distributed control plane mechanisms to react to faults, which do not impose 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 any existing SDN language.

*Configuration Automation.* Many practitioners use configuration templates [18, 33], 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. Past researchers have tried to tackle this problem by analyzing existing firewall configurations [23, 35, 29] and BGP configurations [11, 10, 27, 13, 34] 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 network administrators write configurations at a high-level of abstraction, a whole host of low-level errors can be avoided.

*Configuration Synthesis.* ConfigAssure [25, 26] 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. FUTURE WORK

There are a number of possible directions for future work in programming distributed control planes. One option would be to integrate a model of the environment into the compiler. For example, many ASes make use of informal peering agreements (e.g., tagging routes with certain communities) to enable a wider range of policies. A compiler with this information, could automatically derive routes conforming to these agreements. Another direction would be to perform policy verification at the level of centralized language. There has been great deal of work on verification of the data plane, but much less so on control plane verification. The Propane language provides a high-level abstraction of the control plane, which could be amenable to verification or checking equivalence of policies. For example, an operator could check if adding aggregation at various points in the network as an optimization ever changes routing behavior. The automata-based representation of the product graph may lend itself well to this kinds of analysis. While we target BGP in this paper as a distributed backend for Propane due to its expressiveness, scalability, and uniformity, it should be possible to use other protocols to achieve different types of routing policies (e.g., OSPF). In particular, one could potentially combine different distributed routing protocols through route redistribution to achieve a larger variety of policies. Another possible future direction is automate AS-wide load balancing for BGP. Load-balancing with BGP across external ASes is difficult since there are few mechanisms at the operators disposal. However BGP policies can artificially prepend to the AS path to increases its length, and influence peer decisions. The product graph representation describes, not only which neighbors routers should prefer, but also which neighbors it is indifferent towards. Thus the compiler knowns when it can safely perform load-balancing across different neighbors.

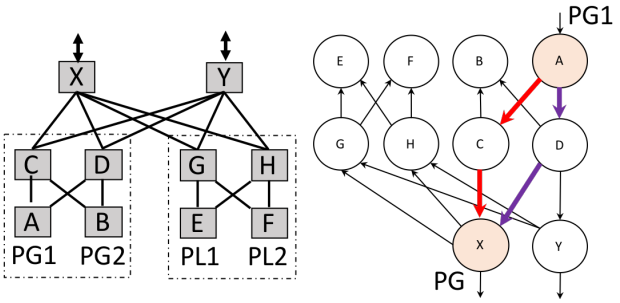



Figure 9: Aggregation safety.

## 10. CONCLUSIONS

In this paper, we introduced Propane, a language and compiler for implementing centralized network policies using a distributed set of devices running BGP. Propane allows operators to describe network-wide intent directly through high-level constraints on the shape and relative preferences of paths for different types of traffic. When Propane compiles a policy, then the resulting BGP configurations are correct-by-construction, faithfully implementing the centralized policy in a distributed fashion, regardless of failures.

## Appendix

## A Aggregation-Induced Black Holes

As demonstrated with Example 2 in §3, BGP aggregation can lead to subtle black-holing of traffic when failures occur. Deciding when this can happen requires knowledge of the routing policy and not just the topology. For instance, a policy might require all traffic for a particular prefix to go over a single link before being aggregated, even if there are several available in the toplogy. If that link fails, a black hole might be introduced. Fortunately, the PGIR captures the information about both the policy and topology precisely.

We frame the aggregation problem as a problem of connectivity in the PGIR. Specifically, for each prefix that falls under an aggregate, we find a lower bound on the number of failures that would disconnect where the prefix originates from where its more specific aggregate is located. The difficulty is that the same links in the topology can appear in multiple places in the PGIR. We adopt the following simple strategy to lower bound the number of failures: $i$) Pick a random start-to-end path in PGIR, $ii$) remove all edges in the PGIR for the set of topological edges chosen, and $iii$) repeat until no such path exists. The number of PGIR paths that we are able to remove is a lower bound on the number of failures required to disconnect the prefix from its aggregate.

For example, imagine we use the data center example from §3, with the policy: `PG1 => end(A)`, where `PG1` falls under the `PG` aggregate. Figure 9 shows the network topology and a simplified PGIR. Since the compiler knows an aggregate will be placed at $X$, and it knows that, the route for `PG1` 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, though in this case there are 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 knowns that 2 is a lower bound on the number of failures for aggregation safety for prefix `PG1`. This process is repeated for other aggregation locations (e.g., $Y$).

## B Propane Properties

**Proposition 10.1.** *BGP configurations for internal routers produced by* Propane *will be stable.*

**Proposition 10.2.** *BGP configurations produced by* Propane *are Sound with respect to the policy: That is, any route traffic takes in the network, is a valid route specified by the policy.*

**Proposition 10.3.** *BGP configurations produced by* Propane *are Complete: That is, the distributed implementation always obtains a most preferred route between two nodes when the corresponding path exists in the network.*

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

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

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

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

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

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

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

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

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

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

[26] 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.

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

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

[29] 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.

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

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

[32] 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.

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

[34] 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.

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