Policy Enforcement using Distributed Control Plane Synthesis

Paper #131
6 pages + 1 page references

Abstract

Operators of modern networks require support for diverse and complex end-to-end policies, such as, middlebox traversals, isolation, and traffic engineering. While Software-defined Networking (SDN) provides centralized custom routing functionality in networks to realize these policies, many networks deploy "traditional" control planes running distributed routing protocols like OSPF and BGP because they are scalable and robust to failures. However, realization of policies by distributed control plane configurations is manual and error-prone, and as failure handling is distributed, ensuring policy-compliance under failures can be challenging. We propose a three-phase architecture to automatically realize fault-tolerant control planes from policy specifications. First, we construct the data plane from policies, and use the data plane to synthesize a control plane whose paths follow the data plane paths. We then transform the control plane to ensure policy-compliance under failures to produce a fault-tolerant control plane. We use a weighted graph-based abstract representation of the control plane called ARC to simplify the synthesis of control planes and compile device configurations based on the underlying network infrastructure requirements. Our paper presents a preliminary treatment of the architecture and the challenges in realizing our vision.

1. INTRODUCTION

Programming networks to correctly forward flows according to user- and application-induced policies is notoriously difficult and error-prone [27, 20]. At least three common characteristics of network policies are to blame: (1) a network may need to satisfy several types of policies, including reachability, isolation, service chaining, resilience, and traffic engineering; (2) many of these policies must also hold under failures; and (3) most policies are global—i.e., they concern end-to-end paths, not individual devices and links.

The global nature of network policies is one motivation for software-defined networking (SDN). SDN allows paths to be centrally computed over a global view of the network. However, it is difficult to ensure forwarding paths are correctly computed and installed in the presence of failures, even if the SDN controller is distributed [4]. Traditional control planes that rely on distributed protocols to compute paths are preferable from a failure-tolerance perspective, but determining the appropriate distributed realization of policies is hard.

Our goal is to *automate the process of creating a correct distributed realization of policies in a traditional control plane*. We seek to handle a wide range of policies (e.g., reachability, service chaining, traffic engineering) and leverage the many protocols (e.g., OSPF, BGP) and mechanisms (e.g., route filters, route redistribution) available in traditional control planes. We also want to ensure the generated control plane computes policy-compliant paths under a bounded number of failures. This contrasts with prior works [26, 7], which generate SDN- or BGP-based control planes for a limited range of policies (e.g., inter-domain reachability).

Although a distributed control plane is composed of a collection of routing processes running on individual switches, we can view the control plane as a single program. The input is a set of endpoints plus the state of network links; the output of this program is a set of paths that conform to the policies encoded in the program. This view inspires us to explore *program synthesis* as a mechanism for generating a policy-compliance control plane.

The problem of synthesizing programs that meet a given specification is computationally hard. As we show in §2, synthesizing a control plane directly from policies is complicated by the fact that, to infer the concrete set of paths induced by a given control plane, one has to incorporate into the synthesis procedure complex concepts such as shortest path algorithms using theories like propositional logic (SAT) and linear rational arithmetic (LRA). Even with recent advances in Satisfiability Modulo Theories (SMT) solvers (e.g., Z3 [14]), this approach does not scale to even moderately-sized networks or sets of policies.

Instead, we propose a three-phased approach (§3; Figure 1). First, we synthesize a concrete set of paths—i.e., a data plane—meeting all specified policies. Second, we generate a simple control plane that, under no failures, induces the set of synthesized paths. This makes the synthesis tractable, as the second phase uses only the theory of linear rational arithmetic (LRA) to generate a control plane. Finally, we apply a series of transformations to obtain a control plane that computes policy-compliant paths under a bounded set of failures (e.g., all single- and double-link failures). If the procedure fails, we produce a different set of paths using the approach in the first phase and try again.

Making this approach scalable and practical requires overcoming several challenges. First, control plane configurations are low-level (e.g., on which interfaces should a routing protocol operate and what costs should be assigned to the incident links) while policies are high-level. Second, we must ensure the transformations we apply to the synthesized control plane cause the program to no longer conform to the input paths. We discuss these challenges in more depth in §3 and present preliminary solutions to address them.

Automatically synthesizing distributed realizations of network policies is an important step towards simplifying network management and reducing the frequency and impact of network failures. Our approach is an important contribution towards the vision of intent-driven networking [19].

2. VISION AND CHALLENGES

One of the foremost tasks in network management is programming the network to forward traffic in a manner consistent with user- and application-induced policies. Common types of policies include: reachability (i.e., which end-hosts can communicate), isolation (i.e., which flows cannot share links), service chaining (i.e., which middleboxes must be traversed), resilience (e.g., how many backup paths are available), and traffic engineering. Every set of forwarding paths (i.e., data plane) installed in the network should conform to these operator-provided policies, otherwise performance, security, or availability problems may arise.

To enforce these policies, the most widely adopted approach is to dynamically compute forwarding paths using distributed routing protocols running on "traditional" network switches. This avoids the scalability and failure tolerance challenges of centralized approaches like SDN [9, 4, 7]. However, programming (i.e., configuring) a traditional control plane to compute policy-compliant paths remains challenging for several reasons:

- The available path-computation algorithms available are predefined—e.g., OSPF uses Dijkstra's shortest path—and their behavior can only be influenced through a limited set of parameters—e.g., link weights.
- Most policies are global—i.e., they concern end-to-end paths, not individual devices and links—making it difficult to map policies to individual devices and control plane parameters.
- A single set of parameters must result in policy-compliant paths under all reasonable network states—e.g., a bounded number of link failures.

These issues motivate us to design a system that automatically synthesizes (configurations for) a control plane that computes policy-compliant paths under a bounded number of failures. Unfortunately, this task is inherently challenging as it requires solving in one shot multiple computationally hard problems. For example, even when considering static routing and no failures, synthesizing a control plane that satisfies policies such as isolation and waypoints is an NP-hard problem. Although the progress in SMT solving has made many NP-hard problems more practical, attempts of incorporating concepts such as shortest path algorithms into these solvers have resulted into huge performance losses [6]. Finally, adding failure handling to the problem results in yet another layer of combinatorial explosion.

Our proposed approach is to tackle these three problems in separate phases (Figure 1). First we synthesize a data plane that meets the input policies and then, by using the data plane as input to the second phase, we express the problem of finding a policy-compliant control plane using only the theory of linear rational arithmetic (LRA), for which we can use efficient off-the-shelf LP-solvers. Finally, we modify the obtained control plane to obtain failure tolerance. The drawback of this approach is that one might need to consider multiple possible data planes before finding one that can successfully complete all the phases of the synthesis algorithm

The solution space of output network configurations to enforce a set of policies is large; configurations can use different network protocols (e.g., only BGP, only OSPF or a hybrid) and mechanisms (e.g., route filters, access control lists or protocol-specific variables). The large solution space of configurations complicates the synthesis of network configurations from input policies, or ties the synthesis to a particular type of configuration. Instead, we borrow a page from programming languages research and synthesize an Abstract Representation for Control planes (ARC) [16] from the data plane; the ARC can then be translated to actual device configurations. The ARC uses the notion that most routing protocols in use today employ a cost-based path selection algorithm; thus a weighted graph can be used to abstractly represent the control plane such that the path between two points taken in the network would be the lowest weight path in the graph. The ARC effectively decouples the policy component with the actual network infrastructure.

3. CONTROL PLANE SYNTHESIS VIA ARCS

In this section, we describe preliminary ideas on how each phase of Figure 1 can be implemented. First, we briefly discuss how existing tools can be used to generate policy-compliant paths (§3.1). Second, we show how to generate ARCs that induce the set of paths synthesized in the first phase (§3.2). Third, we show how ARCs can be modified to deal with resilience (§3.3). We then show how to restart the process whenever one of the three phases fails (§3.4). Finally, we show how we can transform ARCs to actual device configurations (§3.5).

3.1 From policies to paths

The first phase of our approach produces a set of paths adhering to different policies like reachability, service chaining, traffic engineering and isolation. Synthesizing sets of paths—i.e., data planes—that adhere to a given set of policies is already a hard problem, but a few practical approaches

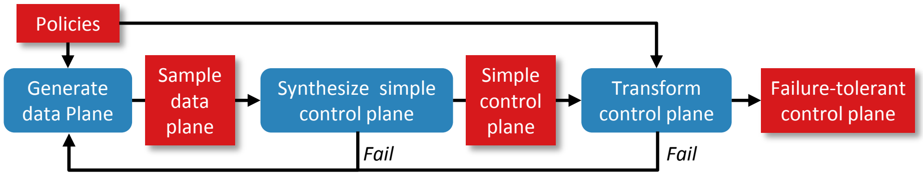

Figure 1: Three-phase process for generating a control plane that generates policy-compliant paths under bounded failures

have been proposed. The approach that is most similar to ours is the one used in Merlin [22]. Given a set of waypoints, reachability, and bandwidth guarantee policies, Merlin generates a mixed integer linear programming instance. A solution to this instance is a set of paths meeting the input policies. Our prototype implementation uses a similar idea and, since this part of the problem is not novel, we briefly provide the intuition behind our solution. Given a set of policies we generate a set of constraints over the propositional theory of rationals and the solutions to these constraints are paths satisfying all the policies. Our rich policy language requires complex Boolean reasoning that forces us to adopt propositional theories in place of simple conjunctions of constraints. In particular, policies like isolation and waypoints require disjunctive formulas.

3.2 From paths to ARCs

The second phase of our approach takes as input the set of paths produced by the first phase and produces an Abstract Representation of the Control Plane (ARC) that realizes the given set of paths. First, we consider the problem of generating ARCs that forward packets based on shortest-paths—e.g., OSPF routing. Then, we consider more complex forms of ARCs in which additional mechanisms like route-filtering are allowed.

3.2.1 From paths to simplified ARCs

The *Simplified Abstract Representation of the Control Plane* (sARC) is a directed graph comprising switches as vertices and weighted edges corresponding to all links in the topology. This is an ideal control plane supporting classic shortest path routing in which no links can be disabled.

The problem of synthesizing a sARC that realizes an input set of paths reduces to a variation of the so-called *inverse shortest path* problem [10]. Assume we are given the following inputs: (1) a directed graph $T = (S, L)$ (the network topology), (2) a set of endpoints $\Gamma \subseteq S \times S$ describing the sources and destinations of the input paths, and (3) a function $P : \Gamma \to 2^{L^*}$ that assigns to each pair of endpoints $(s,t) \in \Gamma$ a set of *acyclic* paths, such that for every path $l_0 \cdots l_n \in P(s,t)$, $l_0 = (s, s')$, for some $s' \in S$, and $l_n = (s'', t)$, for some $s'' \in S$.[1] The *sARC synthesis* problem is to find rational weights for the edges in L such that for each pair of endpoints $(s,t) \in \Gamma$, the paths in $P(s,t)$ are *the* shortest paths from s to t in the graph. Notice, that there can be multiple shortest paths of equal cost for multi-path support (e.g., for traffic engineering).

In sARC, packet forwarding is based on destination. For a destination d, we call ξ_d the directed subgraph of T obtained by only keeping the nodes and edges that are traversed by paths with destination d. Since paths are acyclic, ξ_d is a directed acyclic graph. We use $\Omega = \{d \mid (\_, d) \in \Gamma\}$ to denote the set of all destinations and $\Delta = \{\xi_d \mid d \in \Omega\}$ to denote the set of all destination DAGs.

We use $sw_1 \to sw_2$ to denote $(sw_1, sw_2) \in L$ and $sw_1 \to^* sw_2$ (resp. $sw_1 \to^+ sw_2$) to denote that sw_2 is reachable from sw_1 by crossing zero (resp. one) or more links in T. Similarly, we use $\to_{\xi_d}$ to denote the same relations in the destination DAGs.

Distance equations. To solve the sARC synthesis problem, we generate a set of linear equations to find the required edge weights. We use $E(sw_1, sw_2)$ to denote the weight of the edge $(sw_1, sw_2) \in L$ and $D(sw_1, sw_2)$ to denote the shortest distance from sw_1 to sw_2. We add the equation $D(s,s) = 0$ for every $s \in S$ to denote that the distance from a node to itself is 0. The following equation guarantees that $D(s,t)$ is not greater than the actual shortest distance from s to t.

$$\forall s,t,sw.(s \to sw \to^* t).$$
$$D(s,t) \leq E(s,sw) + D(sw,t) \qquad (1)$$

For each destination DAG $\xi_d \in \Delta$, we add equations to ensure that the input paths with destination d are indeed the shortest ones. If a path is the shortest path between its endpoints, then every subpath of the path has to be the shortest between its endpoints as well (otherwise the complete path would not be the shortest).

Consider a DAG ξ_d for destination d. We define two neighbour functions: $N_T(s)$ denotes the set of neighbours of switch s in the input graph T, and $N_{\xi_d}(s)$ denote the set of neighbours of switch s in the destination DAG ξ_d. Given a destination $d \in \Omega$, we use the following equations to ensure that, given two nodes s and t in ξ_d, the set of paths from s to t in ξ_d are exactly *the* shortest paths from s to t in T. Let s and t be two nodes in ξ_d and let $Paths_{\xi_d}(s,t)$ be the set of paths from s to t in ξ_d.

$$\forall l_0 \cdots l_n \in Paths_{\xi_d}(s,t). \forall n' \in N(s) \setminus N_{\xi_d}(s).$$
$$E(s,n') + D(n',t) > \sum_{l_i=(s_i,t_i)} E(s_i,t_i) \qquad (2)$$

[1] We use L^* the denote the set of all finite sequences over L.

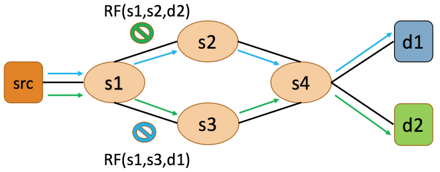

Figure 2: Example of paths that requires route-filtering

$$\forall l_0 \cdots l_n \in Paths_{\xi_d}(s,t).\forall n' \in N_{\xi_d}(s).n' \not\to^{+}_{\xi_d} t.$$
$$E(s,n') + D(n',t) > \sum_{l_i=(s_i,t_i)} E(s_i,t_i) \quad (3)$$

$$\forall l_0 \cdots l_n, l'_0 \cdots l'_n \in Paths_{\xi_d}(s,t).$$
$$\sum_{l_i=(s_i,t_i)} E(s_i,t_i) = \sum_{l'_i=(s'_i,t'_i)} E(s'_i,t'_i) \quad (4)$$

Equation 2 guarantees that the sum of the weights belonging to a path from s to t in ξ_d is smaller than any path that goes to t via a node n' that is a neighbour of s in T but not in ξ_d. Equation 3 guarantees that the sum of the weights belonging to a path from s to t in ξ_d is smaller than any path that goes to t via a node n' that is a neighbour of s in ξ_d but such that t is not reachable from n' in ξ_d. Finally, Equation 3 guarantees that all the paths from s to t in ξ_d have the same weight.

3.2.2 *Synthesis of ARC with route-filters*

The sARC synthesis problem does not always admit a solution. For example, consider the set of paths illustrated in Figure 2. In this example, both the blue and green paths are required to be the unique shortest paths and, clearly, this is cannot be enforced for any choice of the edge weights.

In this section, we consider ARCs that use more refined mechanisms to cope with this problem. One way to synthesize an ARC in the scenario given in Figure 2 is to "disable" the edge $(s1, s3)$ for destination $d1$. Using this technique, there is only one possible path from src to $d1$ and the edge weights become irrelevant for destination $d1$.

This blocking mechanism supported by (non-simplified) ARC is called a route-filter. A route-filter can selectively disable an edge for a given destination by blocking advertisements to a particular destination along a link. Formally, a route-filter is a pair $(l, d) \in L \times S$ disabling a link l for destination d and path to a destination d is *unfiltered* if it does not contain edges that are filtered for destination d. The *ARC synthesis* problem is to find rational weights for the edges in L and a set of route-filters $F \subseteq L \times S$ such that for each pair of endpoints $(s,t) \in \Gamma$, the paths in $P(s,t)$ are the shortest *unfiltered* paths from s to t in the graph.

The ARC synthesis problem admits a trivial solution in which route-filters are used to enforce the exact set of input paths by blocking all other possible paths. This can be done by creating a route-filter (l, d) for every link l not in ξ_d. However, this solution may overly restrict the structure of the network. In particular, if two nodes were connected by a single path, a link failure would immediately result in the network becoming disconnected! Moreover, installing so many route-filters might be costly as not all switches might support this mechanism.

Ideally, we would like to impose further restrictions to the ARC synthesis problem to make sure that the obtained solution satisfy desirable connectivity or resilience properties. In the following we informally use the word objective to identify a property we desire for the synthesized ARC. Examples of objectives are minimizing the number of route-filters or maximizing the number of edge-disjoint paths in the ARC for each endpoint. Given an objective O, the *augmented ARC synthesis* problem is to find rational weights for the edges in L and a set of route-filters $F \subseteq L \times S$ such that 1) for each pair of endpoints $(s,t) \in \Gamma$, the paths in $P(s,t)$ are the shortest *unfiltered* paths from s to t in the graph, 2) the resulting ARC satisfies the objective O. In the following we only allow route-filters for a destination d to be applied to edges of T that are directly connected to nodes in ξ_d.

We propose an iterative approach to this problem. Our algorithm starts by trying to synthesize a solution that does not use route-filters using the equations proposed in §3.2.1. In the case of a failure, the algorithm uses the "proof of unsatisfiability" generated by the constraint solver to greedily add a small set of route-filters. New equations are then generated and approach is repeated until a solution is found. We first describe the modified linear equations generated when a set of route-filters are enabled, and then describe two techniques used to choose route-filters.

Equations with route-filters. We show how the technique used to solve the simplified synthesis problem can be modified to handle filtered and unfiltered paths. We assume we are given a set of route-filters F and use $s \to^{*}_{d} t$ to denote that s can reach t in T without using any edge l, such that $(l, d) \in F$. We use $D_d(sw_1, sw_2)$ to denote the shortest distance from sw_1 to sw_2 using only edges that are not filtered for destination d. We can revise the equation in (1) to correctly restrict the values of D_d by simply ignoring all the filtered edges. In the same way, we can modify equations (2) and (3), while equation (4) remains unchanged.

Unfortunately, if the encoding without route-filters produces n equations, this encoding produces $|\Omega|n$ due to the multiple different distances D_d. Notice that, the shortest distance $D_d(s,t)$ between two nodes s and t without using edges filtered for d cannot be smaller than the shortest distance $D(s,t)$ obtained without considering route-filters. We use this property to simplify the encoding by only computing $D(s,t)$ and by replacing each instance of $D_d(s,t)$ with $D(s,t)$ in equations (2) and (3). It is easy to see that every solution of this simplified set of constraints is also a solution to the original solution (because $D(s,t) \leq D_d(s,t)$). However, the reverse is not true and the set of simplified equations can be unsatisfiable in cases in which the original set is satisfiable. We use the simplified set of equations in our preliminary implementation and show how it yields encouraging results in practice.

If the set of linear equations does not admit a solution, we can add new route-filters so that the equations resulting from the added route-filters admit a solution. We discuss two schemes used to add route-filters: the first scheme uses unsatisfiable cores generated by the solver and the second scheme finds geometrical structures called diamonds that cannot be handled without route-filters.

Adding filters using unsatisfiable cores. Modern LP-solvers have efficient procedures to return an unsatisfiable core, also called IIS (Irreducible Inconsistent Subsystem) [13]. Formally, an IIS is a subset of constraints such that, if all constraints except those in the IIS are removed, the resulting set of linear equations is still inconsistent (unsatisfiable). Moreover, the set is irreducible—i.e., removing any one constraint of the IIS produces a consistent set of constraints.

Suppose that, upon producing our set of linear equations, the solver returns unsatisfiable and produces a concrete unsatisfiable core. Some of the linear inequalities from Equations (2) and (3) will appear in the unsat-core (an unsat-core cannot consist of only constraints from Equation (1) and (4), as all distances and edges set to zero would trivially be consistent with these constraints). In particular, the unsat-core contains some constraint

$$E(s,n') + D(n',t) > \sum_{l_i=(s_i,t_i)} E(s_i,t_i) \tag{5}$$

that was added to reason about some DAG ξ_d.

By adding the route-filter $((s,n'),d)$ to F, this inequality is removed from the set of constraints and the combination of the other constraints appearing in the other unsat-core is now satisfiable. The complete set of equations may still be inconsistent as other unsat-cores might exist. The procedure can be repeated until the resulting set of constraints becomes satisfiable and we have therefore reached a solution to the ARC synthesis problem.

Diamond elimination. Finding an IIS is an NP-hard problem [12] and can result in slow synthesis times. We identify a topological property of the set of input paths that is guaranteed to require route-filters and use it to produce an initial set of necessary route-filters. This technique allows us to reduce the number of times we are required to compute unsat-cores.

We define the structure shown in Figure 2 as a *diamond*. Formally, a problem instance contains a diamond iff there exists two different destinations d and d' such that, in their corresponding DAGs ξ_d and $\xi_{d'}$, there exists two nodes s and t, such that $s \rightarrow_{\xi_d} t$, $s \rightarrow_{\xi_{d'}} t$, and there exists a path $l_0 \cdots l_n$ from s to t in ξ_d that is not a path from s to t in $\xi_{d'}$. As we mentioned at the beginning, synthesizing an ARC for diamond structures requires the addition of a route-filter. In fact, each such a diamond can be "removed" by introducing a route-filter (l_0, d') that hides the path $l_0 \cdots l_n$ for the destination d' (see Figure 2). Diamonds can detected and removed in polynomial time.

Evaluation. Figure 3 shows the preliminary evaluation of our second phase, synthesis of the ARC with route-filters from the input data plane for varying number of paths in the data plane and size of a fat-tree topology [5]. We generate the input paths randomly (the maximum path is 7) and report the average synthesis time. Our implementation uses the Gurobi LP-solver [2].

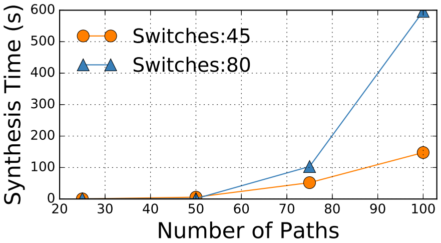

Figure 3: Preliminary evaluation of synthesis of ARC with route-filters.

Policy	**Graph characteristic**
P1: Flow blocked	After removing the edges with filters for the flow, there is no path from s to t
P2: Flow traverses waypoint	After removing the vertices corresponding to waypoints, there is no path from s to t
P3: Destination reachable under $\leq k$ link failures	Max-flow from s to t is $\geq k+1$

Table 1: Graph characteristics an ARC must satisfy to ensure backup paths are policy compliant under up to k link failures

3.3 From ARCs to resilient ARCs

The ARC synthesized using the aforementioned approach represents a control plane that will compute policy-compliant paths in the absence of failures. However, it does not guarantee policy compliance when failures occur. For example, if a link along the shortest path fails, the next shortest path (if one exists) will become the new path to reach the destination. However, the new path may not conform to the same policies as the path in the original failure-free data plane from which the ARC was synthesized—e.g., the new path may no longer traverse a waypoint or have the same bandwidth capacity. Given the high frequency of failures in data center [18], campus, and wide-area [24] networks, it is desirable to generate a control plane that computes *policy-compliant backup paths*. In particular, we want a network to be *k-resilient* [23]—i.e., the control plane will compute a policy-compliant path when there are k or fewer link failures in the network.[2]

Our proposed approach is to attempt to *transform* the ARC synthesized in phase two into a k-resilient ARC. The transformations are based on simple graph characteristics that an ARC must satisfy to be policy-compliant. Table 1 lists several policies and the requisite graph characteristics of a k-resilient control plane.

We illustrate how this can be done to satisfy the policy that a flow from switch s to switch t is blocked on all backup paths (*P1*), it must be the case that all paths between s and t in the ARC contain at least one edge on which the flow is

[2] The value of k depends on operator objectives and the level of redundancy available in the physical topology.

filtered. If we remove all edges with such a filter from the graph, and there remains a path from s to t, then there is some backup path for which the policy does not hold; (one of) the remaining path(s) is the backup path for a scenario in which all of the links with filters have failed. If the aforementioned graph characteristics are not satisfied, then we need to add filters or waypoints to the graph until the characteristic is satisfied. We can do this using a simple iterative process: 1) remove all edges with a filter (or vertices corresponding to waypoints); 2) find a path from s to t; if none exists, then the control plane is k-resilient w.r.t. the current policy and we are done; 3) otherwise, add a filter (waypoint) to some edge along the path, and repeat. Note that adding filters does not impact the selection of paths for a flow, as is the case in §3.2.2, because we are adding filters to ensure there is *no* path for the flow. Although the above steps will produce a k-resilient control plane, the resulting control plane may contain more filters or waypoints than necessary. We can find the minimal number of filters or waypoints to add by computing a min-cut after the first step and adding a filter or waypoint to each edge in the cut.

Compliance with a "flow traverses waypoint" policy (*P2*) requires the flow's graph to have a similar characteristic. Other policies require more complex transformations. For example, to ensure two endpoints can communicate even in the presence of up to k link failures (*P3*), the ARC must contain at least $k + 1$ edge-disjoint, filter-free paths; this is equivalent to a unit-weight, filter-edges-removed version of the ARC having a max-flow of at least $k + 1$ [16]. By removing filters, we may be able to increase the number of available edge-disjoint, filter-free paths. However, this may counteract the addition of filters that occurred during the ARC synthesis phase (§3.2). In this case we need to restart the synthesis procedure with a different set of paths or try considering different route-filters assignments.

3.4 Restarting the synthesis process

One of the drawbacks of using multiple phases is that early phases can produce intermediate solutions that will cause the synthesis to fail in later phases. For example, there might no be a way to produce a resilient ARC for the set of paths produce by the first phase. When we detect such an instance, we produce a different set of paths that satisfies the input policies and restart the synthesis procedure. One could imagine ways to detect failures earlier and reuse part of the computation performed by earlier synthesis attempts.

3.5 Resilient ARCs to device configurations

To deploy the resulting control plane, the ARC must be transformed to actual device configurations expressed in a generic [3] or vendor-specific [1] language. ARC's semantics are equivalent to the semantics of OSPF and layer-3 access control lists (ACLs). Thus, we can trivially compile the ARC into device configurations by: (1) setting the OSPF link weights to the edge weights of the ARC, and (2) adding a layer-3 ACL for each route-filter in the ARC. However, the resulting configurations may be overly complex [8] or use features that make the network more prone to failures [17]. Consequently, part of our future work is to explore how to generate more optimal configurations from ARCs.

4. RELATED WORK

Centralized control. Programming routers for centralized control has been an active area of research in recent times. RCP [11] uses BGP as an substitute to OpenFlow to install rules at the switches, but must respond to failures and update the rules at switches. Fibbing [25] provides centralized control over distributed routing by creating fake nodes and fake links to steer the traffic in the network through paths that are not the shortest. However, these fake advertisements can create forwarding loops in the network during failures, and the centralized controller has to respond to failures (response to failures can precomputed), thus making the controller a central point of failure. In contrast, our approach to distributed control plane synthesis can provide the same expressive power as Fibbing but also avoids any centralized component by engineering the control plane parameters to match the input specifications.

Configuration synthesis. Propane [7] tackles synthesis of BGP configurations to satisfy network-wide objectives, however they tackle a narrower space of BGP policies using algorithms over automata and graphs. ConfigAssure [21] uses a combination of logic programming and SAT solving to synthesize network configurations. Fortz et. al [15] tackle the problem of optimizing OSPF weights for performing traffic engineering, however their work is tailor-made to the specific problem of TE and is inextensible to apply to custom forwarding behaviors for other kinds of policies.

5. CONCLUSION

Programming a legacy network control plane to satisfy a variety of connectivity, security, and performance policies is a complex and error-prone task. We have shown that program synthesis is a promising approach to automate this process and produce a control plane that is correct by construction. In particular, we presented an architecture where a network operator provides a set of policies and the system automatically provides a set of device configurations that leverage the control plane features available on each device to compute policy-compliant paths, even in the presence of failures. We show how the synthesis problem can be made tractable by modeling the control plane using a graph-based abstract representation tied to a traditional shortest path algorithm and by dividing the synthesis procedures into multiple phases. However, we have only explored how to address a subset of important policies and leverage a few control plane features available on traditional networking hardware. Our future work will focus on satisfying a wider range of policies using more features and into making our system practical for use with real networks.

6. REFERENCES

[1] Cisco IOS configuration fundamentals command reference. http://cisco.com/c/en/us/td/docs/ios/fundamentals/command/reference/cf_book.html.

[2] Gurobi Optimization. http://http://www.gurobi.com/.

[3] OpenConfig: Vendor-neutral, model-driven network management designed by users. http://openconfig.net.

[4] A. Akella and A. Krishnamurthy. A highly available software defined fabric. In *HotNets*, 2014.

[5] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *ACM SIGCOMM Computer Communication Review*, volume 38, pages 63–74. ACM, 2008.

[6] S. Bayless, N. Bayless, H. H. Hoos, and A. J. Hu. Sat modulo monotonic theories. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[7] R. Beckett, R. Mahajan, T. Millstein, J. Padhye, and D. Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the ACM SIGCOMM 2016 Conference on SIGCOMM*, SIGCOMM '16.

[8] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *NSDI*, 2009.

[9] P. Berde, M. Gerola, J. Hart, Y. Higuchi, M. Kobayashi, T. Koide, B. Lantz, B. O'Connor, P. Radoslavov, W. Snow, and G. M. Parulkar. ONOS: towards an open, distributed SDN OS. In *HotSDN*, 2014.

[10] P. Broström and K. Holmberg. Compatible weights and valid cycles in non-spanning ospf routing patterns. *Algorithmic Operations Research*, 4(1):19–35, 2009.

[11] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a routing control platform. In *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*, NSDI'05, pages 15–28, Berkeley, CA, USA, 2005. USENIX Association.

[12] N. Chakravarti. Some results concerning post-infeasibility analysis. *European Journal of Operational Research*, 73(1):139–143, 1994.

[13] J. W. Chinneck. *Feasibility and Infeasibility in Optimization:: Algorithms and Computational Methods*, volume 118. Springer Science & Business Media, 2007.

[14] L. M. de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS), 14th International Conference*, pages 337–340, 2008.

[15] B. Fortz and M. Thorup. Internet traffic engineering by optimizing ospf weights. In *INFOCOM 2000. Nineteenth annual joint conference of the IEEE computer and communications societies. Proceedings. IEEE*, volume 2, pages 519–528. IEEE, 2000.

[16] A. Gember-Jacobson, R. Viswanathan, A. Akella, and R. Mahajan. Fast control plane analysis using an abstract representation. In *Proceedings of the ACM SIGCOMM 2016 Conference on SIGCOMM*, SIGCOMM '16.

[17] A. Gember-Jacobson, W. Wu, X. Li, A. Akella, and R. Mahajan. Management plane analytics. In *IMC*, 2015.

[18] P. Gill, N. Jain, and N. Nagappan. Understanding network failures in data centers: Measurement, analysis, and implications. In *Proceedings of the ACM SIGCOMM 2011 Conference*, SIGCOMM '11, pages 350–361, New York, NY, USA, 2011. ACM.

[19] D. Lenrow. Intent: What. not how. http://opennetworking.org/?p=1633&option=com_wordpress&Itemid=471.

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

[21] S. Narain, G. Levin, S. Malik, and V. Kaul. Declarative infrastructure configuration synthesis and debugging. *J. Netw. Syst. Manage.*, 16(3):235–258, Sept. 2008.

[22] R. Soulé, S. Basu, P. J. Marandi, F. Pedone, R. Kleinberg, E. G. Sirer, and N. Foster. Merlin: A language for provisioning network resources. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 213–226, New York, NY, USA, 2014. ACM.

[23] B. Stephens, A. L. Cox, and S. Rixner. Plinko: Building provably resilient forwarding tables. In *Proceedings of the Twelfth ACM Workshop on Hot Topics in Networks*, HotNets-XII, pages 26:1–26:7, New York, NY, USA, 2013. ACM.

[24] D. Turner, K. Levchenko, A. C. Snoeren, and S. Savage. California fault lines: Understanding the causes and impact of network failures. In *SIGCOMM*, 2010.

[25] S. Vissicchio, O. Tilmans, L. Vanbever, and J. Rexford. Central control over distributed routing. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 43–56, New York, NY, USA, 2015. ACM.

[26] Y. Yuan, R. Alur, and B. T. Loo. Netegg: Programming network policies by examples. In *HotNets*, 2014.

[27] H. Zeng, P. Kazemian, G. Varghese, and N. McKeown. A survey on network troubleshooting. Technical Report TR12-HPNG-061012, Stanford University, June 2012.