Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks

Kausik Subramanian Loris D'Antoni Aditya Akella

University of Wisconsin-Madison, USA

{sskausik08, loris, akella}@cs.wisc.edu

Abstract

Operators in multi-tenant cloud datacenters require support for diverse and complex end-to-end policies, such as, reachability, middlebox traversals, isolation, traffic engineering, and network resource management. We present GENESIS, a datacenter network management system which allows policies to be specified in a declarative manner without explicitly programming the network data plane. GENESIS tackles the problem of enforcing policies by synthesizing switch forwarding tables. It uses the formal foundations of constraint solving in combination with fast off-the-shelf SMT solvers. To improve synthesis performance, GENESIS incorporates a novel search strategy that uses regular expressions to specify properties that leverage the structure of datacenter networks, and a divide-and-conquer synthesis procedure which exploits the structure of policy relationships. We have prototyped GENESIS, and conducted experiments with a variety of workloads on real-world topologies to demonstrate its performance.

Categories and Subject Descriptors C.2.3 [*Network Operations*]: Network management; I.2.2 [*Automatic Programming*]: Program synthesis; D.2.4 [*Software/Program Verification*]: Formal methods

Keywords Network management, Software-defined networks, SMT

1. Introduction

Many enterprises are increasingly migrating their on-premise IT infrastructure to cloud datacenters. In such environments, the different enterprises (tenants) share different resources, such as, the compute machines that run their applications and network infrastructure used for communication among these applications. Operators of such multi-tenant datacenters thus have to deal with a multitude of machines communicating with each other (flows) over a network that is composed of many tens to hundreds of routers or switches (devices) [14]. With growing diversity of enterprise applications and the need for security and compliance, these pathways of communication through the datacenter network are subject to increasingly complex network-based policies.

Consider a tenant in such a datacenter. She may desire basic communication among her applications (reachability) along shortest paths based on certain metrics. In addition, she may wish that traffic attempting to reach some of her applications is examined by a set of "middleboxes" (traversal) for auditing and access control. For strong security or Quality-of-Service considerations, a tenant may additionally desire that a subset of her flows does not share any infrastructure with others' flows (isolation). In parallel, cloud operators must meet key operational policies. For instance, they often need to optimize network performance objectives (traffic engineering), e.g., minimizing the maximum load imposed by all tenants on network links, and deal with resource constraints such as link capacity bounds and switch table sizes. Also, since datacenter networks are highly prone to link/switch failures [15], operators need to gracefully transition the old (pre-failure) data plane to a policy-compliant one (post-failure) in a rapid and/or efficient manner.

Today, configuring network devices to enforce these complex policies in aggregate is manual, ad-hoc, and error-prone. This can lead to misconfigurations and violations of tenant service-level agreements, which can have severe performance and security impacts.

With software-defined networking (SDN), operators can program networks in a more intuitive manner. In SDN, a general-purpose centralized controller machine (control plane) controls end-to-end communication pathways by managing network forwarding rules on a collection of programmable switches (data plane). Using a global view of the current network topology, the controller can program forwarding rules on switches based on application requirements. Unfortunately, many existing SDN programming languages [13, 23] present too narrow a view: operators would ideally want to specify and realize policies network-wide, whereas these languages focus on programming *individual* switch behaviors. Other recent works on network-wide policy enforcement [5, 18, 27, 28, 31, 32] go beyond the single-switch model, but they target specific types of policies and are not easily extensible to different kinds of policies. Notably, NetKAT is among the most expressive and can be used to encode certain network-wide policies like regular paths and programs on virtual topologies, however, it cannot be used to express policies based on hyperproperties [10] (where one class's path is dependent on the other) like isolated paths or traffic engineering. Furthermore, for many types of policies, generating a data plane that enforces them is a computationally hard problem, requiring the design of efficient custom heuristics per policy type.

In this paper, we seek a general approach that allows a variety of rich policies to be specified as the input, with the output being the corresponding set of switch forwarding rules such that the complexities of correctly realizing the policies in the data plane are hidden from operators. This is an important step toward *intent-based networking* [2], where operators specify *what* they want the network to do instead of worrying about *how* the network must be configured. We argue that data plane synthesis can help realize this vision in the multi-tenant datacenter context.

POPL '17, January 18 - 20, 2017, Paris, France

ACM 978-1-4503-4660-3/17/01…$15.00
DOI: http://dx.doi.org/10.1145/3009837.3009845

We present GENESIS, a framework for declaratively specifying and enforcing complex policies such as, isolation, middlebox traversals, network optimization objectives, and failure resilience. To tackle the high complexity of enforcing some of these policies (e.g., enforcing isolation is NP-complete), GENESIS encodes the problem of enforcing policies as a constraint solving problem and leverages recent advances in fast Satisfiability Modulo Theories (SMT) solvers to efficiently search for a solution to the constraints. The solution is then translated into switch forwarding rules. GENESIS uses two intuitive relations that concisely capture the semantics of custom network forwarding behaviors. These help express a variety of both path-based and global policies desired in a datacenter. Interestingly, complex global policies (specifically, policy-compliant failure resilience) can be realized within this framework without requiring additional encoding (of specific failure scenarios) by just cleverly transforming path-based policies. By leveraging the formal guarantees of constraint solving, GENESIS eliminates the room for error in the enforcement of complex policies.

Further, we present two novel techniques that leverage domain-specific properties to speed up GENESIS's synthesis. First, GENESIS allows the network operator to write restricted forms of regular expressions, called *tactics*, that blacklist paths based on certain patterns that are not desired in a datacenter network (e.g., paths that alternate between topology tiers). These tactics are used to discard several constraints, acting as a search strategy for the solver. Tactics can speed up the synthesis procedure by $1.5-400\times$ (median speedup: $1.6\times$, average speedup: $22\times$). Second, we develop a *divide-and-conquer* synthesis procedure that opportunistically leverages the dependency relationships among isolation policies to improve synthesis performance. The procedure partitions the input policies into components such that GENESIS can synthesize these components separately and faster than the complete problem. Divide-and-conquer synthesis can halve the synthesis time for 40% of synthetic isolation workloads.

Contributions. Our contributions are the following.

- An extensible declarative framework for describing complex policies like isolation, waypoints (§4), traffic engineering (§5), and failure resiliency (§5.3.1) and a modular SMT-based algorithm for enforcing such policies;
- A tactic-based synthesis algorithm, which leverages datacenter network structure to blacklist undesirable path patterns (§6);
- A divide-and-conquer procedure for speeding up synthesis by leveraging the structure of policy interactions (§7);
- An implementation of GENESIS and an evaluation on different policy workloads, topologies and multi-tenancy settings (§8).

2. Preliminaries and Policies Supported

We describe the type of policies desired in multi-tenant data centers that GENESIS supports. We use Figure 1 as a running example. In our setting, an "operator" manages a multi-tenant datacenter like an enterprise network or a private datacenter. The operator specifies "operator policies" which reflect important global objectives pertaining to how she wishes to manage her overall infrastructure. A tenant is an entity (e.g., an enterprise or a department thereof) that has offloaded its IT infrastructure to the datacenter. Each tenant controls a number of host machines in the datacenter running some of its applications, and specifies path-based policies (as opposed to operator's global policies). Tenant policies define whether paths can exist among its hosts, and if so, what additional properties the paths must satisfy for security, performance or access control reasons. Given the policies, the cloud operator solves the VM placement problem separately. The resulting tenant VM locations, along with the policies to apply on paths between locations are then provided as input to Genesis. Tenants are *unaware* of the physical topology and cannot program physical switches directly, maintaining the virtual network abstraction.

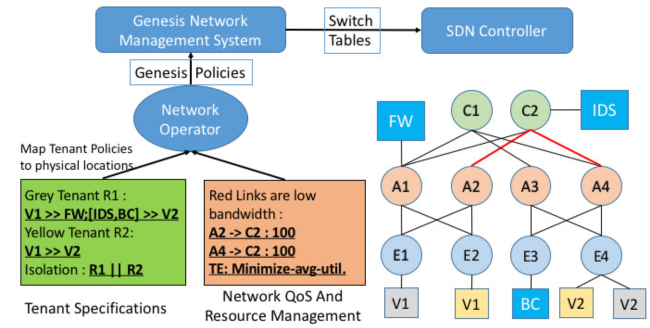


Figure 1: GENESIS in a multi-tenant datacenter setting of a network containing several VMs and middleboxes. The network operator translates the tenant specifications and network resource policies to GENESIS policies and GENESIS synthesizes switch forwarding rules which are installed by the SDN controller.

Figure 1 shows several tenants who differ in the nature of policies they wish to realize; we also show the operator policies. The policies supported by GENESIS are described below. Notice that these reflect and, in some cases, extend policies that today's enterprises and datacenter operators realize in their networks [14].

Tenant Policy: Reachability. This enables network communication between specific pairs of tenant's virtual instances (VMs), applications, or hosts. In our example, one tenant has defined a reachability policy (R2) for its two VMs: `V1 » V2`, which is translated after VM placement to the source and destination switches `E2 » E4`. A pair of VMs, applications or hosts that are allowed to communicate by means of a reachability policy defines a flow or a "packet class"; we use these terms interchangeably. Any communication that is not defined by a reachability policy is implicitly blocked (i.e., all communication is "default off").

Tenant Policy: Middlebox Traversals. A tenant may wish that the flow between two of her end hosts, or from another tenant, must traverse specific middleboxes, which we also refer to as "waypoints" in this paper. Middleboxes are custom processing appliances often used for security, access control, or performance reasons (e.g., firewalls, intrusion prevention systems, monitoring/accounting gateways, proxies, and load balancers). Specifically, for particular flows of interest, a tenant can provide a sequence of unordered sets of middleboxes to traverse [26]. The flows must traverse these sets in order, while in a set, all middleboxes must be traversed and the order is irrelevant.[1] For example, one of the tenants defines a traversal policy (R1): `V1 » FW; [IDS,BC] » V2` which specifies that traffic must first pass through the firewall (FW), and then through the Intrusion detection system (IDS) and the byte counter (BC) in any order.

Tenant Policy: Isolation. Tenants may require various Quality-of-Service (QoS) or security guarantees that stipulate varying degrees of isolation for their traffic. In the extreme, a tenant could require that her flows are not affected in any manner by any other tenant by strictly isolating the path of the tenant's flows from others' flows. In Figure 1, we have two tenants whose traffic will be isolated from one another (`R1||R2`), i.e., the network paths used by the tenants will not share any links in the topology. A tenant could also specify isolation for a subset of her (performance-sensitive) flows from other

[1] The unordered set abstraction leverages the fact that middleboxes without dependencies in their traffic processing behavior can be placed in any order relative to each other [26].

Type	Policy	GPL Syntax	Description
Tenant	Reachability	`predicate : src » dst`	Forwarding Rules for path from switch src to switch dst for packets matching $predicate$
	Reachability with Ordered Waypoints	`predicate : src » W`$_1$`; ...;W`$_n$` » dst`	Forwarding Rules for path from switch src to switch dst for packets matching $predicate$ such that the path traverses $w \in W_1$ in any order, then $w \in W_2$ in any order after all waypoints in W_1 are traversed and so on.
	Traffic Isolation	`R1 || R2`	Paths of two reachability policies $R1$ and $R2$ do not share a link in the same direction
	Link Isolation	`R1 <> R2`	Paths of two reachability policies $R1$ and $R2$ do not share a link in any direction (edge-disjoint)
Operator	Link Capacity	`sw`$_1$ $\rightarrow$ `sw`$_2$ `: capacity`	The weights of flows traversing the link $sw_1 \rightarrow sw_2$ do not exceed $capacity$
	Switch Table Size	`sw : size`	The number of flows traversing through sw do not exceed $size$ as each flow would require a forwarding rule at sw
	Traffic Engineering	`minimize-tot-te, minimize-max-te`	TE objectives: minimize total/max link utilization

Table 1: Genesis Policy Support with Genesis Policy Language (GPL) syntax

flows of the same tenant or those belonging to other tenants; the rest of the tenant's flows may require no guarantees.

Operator Policy: Managing Capacity Constraints. While support for the above policies can be used to satisfy tenant requirements, network operators often wish to carefully manage constrained resources. Common examples that GENESIS supports include enforcing strict constraints on aggregate number of flows traversing a switch (due to all tenants) so as to adhere to switch memory constraints, and ensuring that the total load on certain links is within predefined thresholds (we assume here that each flow has a predefined load that it imposes on the path it uses). In our example in Figure 1, A2-C2 and A4-C2 links are of low bandwidth, and the operator wants to ensure that total load on these links does not exceed 100 (`A2` $\rightarrow$ `C2: 100, A4` $\rightarrow$ `C2:100`). These policies could also be used to provide QoS guarantees to tenants like minimum bandwidth guarantees.

Operator Policy: Traffic Engineering. As an alternative to managing strict (link) capacity constraints, operates may also want to balance load on their network infrastructure. This is often done by optimizing a network-wide objective such as total or maximum utilization of network links due to traffic induced by all tenants.

Operator Policy: Handling failure gracefully. Modern networks experience link and switch failures frequently [15]. When a failure occurs, we must reconfigure the forwarding rules so that the policies are satisfied. Naively recomputing forwarding rules incurs an unduly large overhead because old forwarding rules have to be torn down at all switches, and new rules must be installed. Switch rules deletions/insertions take a non-trivial amount of time [17], potentially leading to disruptions. It is therefore desirable to have graceful approaches that either minimize the potential for disruption by minimizing the number of forwarding rules or switches modified in transitioning from an old data-plane, or eliminate the possibility of disruption altogether by precomputing backup policy-compliant paths for a fixed number of failures (at the cost of storing extra rules at switches).

Realizing these policies is challenging today. In particular, state-of-the-art SDN frameworks, e.g., Pyretic [23] and Frenetic [13], are insufficient to program networks to realize them. This is because the above policies are global and cannot be enforced (at least not in an intuitive manner) by programming individual behavior of switches. While some existing SDN-based network management systems [20, 27, 32] overcome these limitations by taking a network-wide view, they are tailored to support specific policies such as middlebox placement or link capacity constraints. As such they cannot enforce several of the policies above.

3. Data Plane Synthesis

Our contribution is GENESIS, a new general network management system that supports the above policies, and can be extended to support others. The architecture of GENESIS is shown in Figure 1. GENESIS performs *synthesis* of switch forwarding rules to enforce policies. The policies are specified using Genesis Policy Language, or GPL, as shown in Table 1.

Unlike previous efforts in the network synthesis space [29, 32], GENESIS is not tailored to specific formalisms such as regular expressions; this aspect makes it *modular* and *easy to extend*. To draw an analogy with SMT solvers, GENESIS can be seen as a constraint solver that allows the addition of different types of policies (respectively, theories in SMT) and the design of optimizations based on properties desired by network operators.

Our work is motivated by recent advances in program synthesis, i.e., the task of discovering an executable program from user intent expressed in the form of some constraints. There are three key dimensions to a synthesis problem: the type of constraints that it accepts as expression of user intent, the space of programs over which it searches, and the search technique it employs. GENESIS leverages synthesis as follows: given a set of policies which describe tenant and operator intent, the search space is the space of all data planes (i.e., the set of forwarding rules) and the search technique involved is SAT/SMT solving.

GENESIS's approach has the following salient features:

(1) Enforcement of the different policies can be translated to the following problem: Given a set of node pairs (derived from the reachability policies) in the graph (topology), find paths in the graph for each of the node pairs satisfying certain properties (derived from the rest of the policies). Thus, the different policies can be enforced by a correct set of forwarding rules at the switches. No extra functionality is required from the controller; its only role is to install the forwarding rules on switches.

(2) Correct enforcement is challenging due to different goals for each of the policies — ensuring isolation between paths may lead to overshooting link utilizations and vice-versa — and is a common cause of incorrect configurations in networks. Our approach removes the need for a verification step in which the operator has to "check" whether the forwarding rules satisfy the desired policies. By using a formal reasoning technique, we are able to consider the space of all data planes and find a solution which is *correct by construction*, eliminating room for operator errors.

(3) Automatically enforcing policies is a task with *high theoretical complexity*. For example, enforcing isolation policies is as hard as solving graph-coloring, an NP-complete problem. Specialized techniques can be used to find the forwarding rules when handling a particular class of policy, but devising good search techniques

becomes challenging when multiple types of policies are combined,– e.g., isolation, waypoints, and traffic engineering. Thanks to the many engineering efforts, SMT solvers abstract away most of this complexity and allow us to unify search objectives for every policy into a generalized search technique. Crucially, GENESIS can be extended with ease to support new policies without requiring changes to the underlying search techniques.

In the next section, we describe the GENESIS synthesis algorithm for tenant policies. We then describe how to accommodate operator policies pertaining to capacity constraints, traffic engineering and failure resiliency (§5). Finally, we describe two novel techniques aimed at speeding up GENESIS's synthesis: tactics (§6) and divide-and-conquer synthesis (§7).

4. Synthesis of Tenant Policies

The problem statement here is as follows: Given the network topology graph and the set of tenant policies written in GPL, generate paths in the network for every source-destination pair (derived from reachability policies) satisfying all policies. To achieve this, GENESIS creates constraints that encode the forwarding and reachability rules pertaining to the paths such that they satisfy the input policies. The synthesized solution of paths obtained from the constraints are then translated to switch forwarding rules.

4.1 Network Forwarding Model

We start by describing the basic forwarding model we use in GENESIS. We define the physical switch topology as an undirected graph $T = (S, L)$, where S is the set of switches and L is the set of links. We use the neighbour function $N(s) = \{s' \mid (s, s') \in L\}$ to denote the set of neighbour switches of s. We assume a set of packet classes $PC : [0, \lambda]$ and map each reachability policy to a unique integer in PC. In the rest of the paper, we often use the term packet class to identify the corresponding reachability policy. Other policies are not mapped to packet classes as they do not produce a path, but specify restrictions on paths of packet classes. We use R to denote the set of reachability policies; each policy $r \in R$ is a pair $(predicate{:}src >> W_1; W_2; \ldots; W_n >> dst, pc)$ where:

- $predicate$ is the packet header identifier pertaining to r;
- $src, dst \in S$ are the source and destination switches;
- $W_1, W_2, \ldots, W_n \subseteq S$ are the (potentially empty) ordered sets of waypoints;
- $pc \in PC$ is the packet class and is a unique integer used to identify the variables associated to r

We fix a constant μ and assume all paths to have length at most μ. $K = [0, \mu]$ is the set of all permissible path lengths. The network forwarding model abstracts the actual forwarding rules at each node and encodes the reachability of each packet class.

We use the relation $Fwd \subseteq S \times S \times PC$ to capture the network forwarding behavior,—i.e. $(sw_1, sw_2, pc) \in Fwd$ if sw_1 forwards packets of class pc to switch sw_2. We use the relation $Reach \subseteq S \times PC \times K$ to capture the path reachability,—i.e. $(sw, pc, k) \in Reach$ if the switch sw is reachable in the path from the source switch of packet class pc in exactly k steps. For brevity, we write $Fwd(sw_1, sw_2, pc)$ for $(sw_1, sw_2, pc) \in Fwd$ and similarly for the $Reach$ relation. Since Fwd depends on the topology, for all sw_1, sw_2 that are not connected by a link, we have that $\forall pc$, $(sw_1, sw_2, pc) \notin Fwd$.

Given a set of policies, GENESIS generates a set of constraints denoted by Ψ over the Fwd and $Reach$ relations. $(Fwd, Reach) \models \Psi$ denotes that Fwd and $Reach$ is a model of Ψ.

Definition 1. *Given two concrete relations Fwd and Reach, the set of induced paths* $\Pi = \mathtt{paths}(Fwd, Reach)$ *is defined as follows: given a class pc,* $(sw_0 \ldots sw_k, pc) \in \Pi$ *iff :*

1. $\forall i \in [0, k].(sw_i, pc, i) \in Reach$

2. $\forall i \in [0, k-1].(sw_i, sw_{i+1}, pc) \in Fwd$

Figure 2 illustrates these definitions.

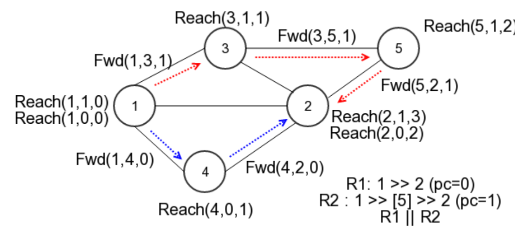


Figure 2: Values of the Fwd and $Reach$ relations of the network forwarding model for the policies specified in the figure. The blue and red arrows indicate the paths of packet classes 0 and 1 respectively according to the model.

Definition 2. *Given the set of constraints* Ψ *corresponding to the input policies, a set of paths* Π *is a solution to* Ψ, $\Pi \models \Psi$, *if there exists* $Fwd, Reach$ *such that* $(Fwd, Reach) \models \Psi$ *and* $\Pi = \mathtt{paths}(Fwd, Reach)$.

In practice, we model the forwarding and reachability relations using propositions and reduce enforcement of tenant policies like reachability, waypoints and isolation to a Boolean Satisfiability Problem (SAT) problem. Using these relations, operators can write custom policies in a concise and intuitive manner.

4.2 Reachability

We first discuss the constraints added to Ψ for reachability policies without waypoints. For a reachability policy $s >> d$ and packet class pc, the added constraints must ensure that the solution model represents a path from source to destination. The base constraint states that $(s, pc, 0) \in Reach$ meaning that s can be reached in 0 steps. The following constraint states that there must be a forwarding rule from s to one of the neighbors of s[2].

$$\exists n \in N(s).\ Fwd(s, n, pc) \wedge Reach(n, pc, 1). \qquad (1)$$

Next, we add the following constraints that state that d can be reached in some number of steps and, since d is the last switch in the path, there are no forwarding rules from it.

$$\exists k.\ Reach(d, pc, k)\ \wedge\ \forall n \in N(d).\ \neg Fwd(d, n, pc). \qquad (2)$$

Finally, we add implication constraints that propagate reachability backward from destination to source. If a node n_1 is reachable in k steps, there must be a node n_2 reachable in $k-1$ steps and a forwarding rule $n_2 \to n_1$.

$$\forall n_1.\forall k \geq 1.\ Reach(n_1, pc, k) \implies \exists n_2.n_2 \in N(n_1) \wedge \\ Reach(n_2, pc, k-1) \wedge Fwd(n_2, n_1, pc). \qquad (3)$$

When combined together, these constraints are sufficient to ensure the existence of a path from s to d for packet class pc. However, since there is no restriction on the number of Fwd values that can be true at a switch, we can get multiple forwarding rules at switches, and also multiple paths to the destination. These can also create forwarding loops. Concretely, this is not a problem: as long as there is at least one path from s to d we can recover it from the solution of the constraints. Moreover, this representation is quite efficient, as forcing a single path would require adding further constraints (§4.3) and increase the synthesis time.

To extract a concrete s-to-d path we perform a breadth-first search on the reachability graph induced by the solution to the con-

[2] We unroll the existential quantifier $\exists n \in N(s)$ using disjunction of clauses $\vee_{n \in N(s)}$ and the universal quantifier $\forall n \in N(dst)$ using conjunction of clauses $\wedge_{n \in N(dst)}$ and stay in propositional logic.

straints. A directed edge $n_1 \rightarrow n_2$ appears in the reachability-graph if there is a forwarding rule indicated by the relation $(n_1, n_2, pc) \in Fwd$. We extract the rules relevant to the shortest path from source to destination from the model, and the additional rules obtained in the solution (extra paths, forwarding loops) are ignored.

4.3 Waypoint Traversal

For a reachability policy with a sequence of waypoint sets $s >> W_1; \ldots; W_n >> d$ and packet class pc, we add all the constraints specified in §4.2 to ensure the existence of a path from s to d. We then add constraints so that all waypoints w are traversed.

$$\forall w \in W_1, \ldots, W_n.\ \exists k.\ Reach(w, pc, k). \tag{4}$$

For each set W_i for $i > 1$, we add constraints to ensure that all waypoints in W_i are reached after all waypoints in W_{i-1} :

$$\begin{aligned}\forall w_i \in W_i, \forall k_i.\ Reach(w_i, pc, k_i) \implies \forall w_{i-1} \in W_{i-1}.\\ \exists k_{i-1}.\ k_{i-1} < k_i \wedge Reach(w_{i-1}, pc, k_{i-1}).\end{aligned} \tag{5}$$

Previously, we imposed no restriction on the number of paths from s to d. In the case of waypoints, this can result in a solution with multiple paths, with each individual path traversing some of the waypoints, which is not the correct enforcement for a waypoint policy. Thus, we need to ensure the solver returns a single path traversing all the waypoints. To achieve this, we limit the number of forwarding rules for pc at a switch to 0 or 1. We define the forwarding set as:

$$FwdSet(sw, pc) = \{k \mid Fwd(sw, k, pc)\}. \tag{6}$$

We then add constraints stating that the size of the forwarding set must not exceed 1:

$$\forall sw, pc.\ |FwdSet(sw, pc)| \leq 1. \tag{7}$$

Here $|A|$ denotes the size of set A. The above constraints are expressed in SAT as follows:

$$\bigvee_{k_1 \in N(sw)} Fwd(sw, k_1, pc) \wedge (\bigwedge_{k_2 \in N(sw), k_2 \neq k_1} \neg Fwd(sw, k_2, pc)) \tag{8}$$

Since, there cannot exist multiple rules at a switch, the model will contain a single path from source to destination for pc traversing the waypoints in the right order.

4.4 Isolation

A traffic isolation policy $pc_1 || pc_2$ states that the paths for pc_1 and pc_2 do not share any link in the same direction. We enforce this policy by adding to Ψ, constraints stating that at every switch, pc_1 and pc_2 must not forward to the same switch:

$$\forall n_1.\ \neg(\exists n_2. Fwd(n_1, n_2, pc_1) \wedge Fwd(n_1, n_2, pc_2)). \tag{9}$$

For a link isolation policy $pc_1 <> pc_2$ which prevents sharing a link in both directions, we add the constraints:

$$\begin{aligned}\forall n_1.\ \neg(\exists n_2. Fwd(n_1, n_2, pc_1)\ \wedge\\ (Fwd(n_1, n_2, pc_2) \vee Fwd(n_2, n_1, pc_2))).\end{aligned} \tag{10}$$

With single paths for pc_1 and pc_2 (when combined with Equation (7)), the above constraints ensure those paths are isolated. Interestingly, for a reachability policy without waypoints, the constraints in Equation (7) are not required to enforce isolation. Even though the solver could produce multiple forwarding rules which induce multiple paths, the constraints in Equation (9) or Equation (10) guarantee isolation as the solver would discard the rules conflicting with another packet class.

5. Synthesis of Operator Policies

We now describe how to extend GENESIS's synthesis to support various operator policies. We describe GENESIS can support hard capacity constraints and optimization objectives pertaining to traffic engineering using linear rational arithmetic (LRA) and linear optimization objectives in SMT. We conclude by describing how GENESIS can be extended to allow operators to handle datacenter network failures in a graceful policy-compliant manner.

5.1 Link and Switch Table Capacity

For a link capacity policy on the link $sw_1 \rightarrow sw_2 : \omega$, GENESIS must ensure that the sum of traffic rates of packet classes using link $sw_1 \rightarrow sw_2$ does not exceed ω. As input, we have the traffic rates $\sigma(pc)$ of each of the packet classes. The constraints added to Ψ are:

$$\sum_{\forall pc} \mathtt{ite}(Fwd(sw_1, sw_2, pc), \sigma(pc), 0) \leq \omega. \tag{11}$$

If a class pc uses link $sw_1 \rightarrow sw_2$, then $(sw_1, sw_2, pc) \in Fwd$ and $\sigma(pc)$ is added in the utilization of the link.
A switch table policy $sw : \gamma$ specifies that the number of forwarding rules on sw must not exceed γ. Similar to the link capacity policy, the constraints ensure the count of all packet classes which traverse sw (each will require a forwarding rule) is $\leq \gamma$:

$$\sum_{\forall pc} \mathtt{ite}(\ \exists k. Reach(sw, pc, k), 1, 0) \leq \gamma. \tag{12}$$

5.2 Traffic Engineering

While the above capacity policies can be used to perform a strict form of traffic engineering (TE) in terms of adhering to link bandwidths, it is often more useful to balance traffic across links because a link failure will affect fewer flows when the flows are spread evenly across the network. To this end, network operators often impose traffic engineering objectives such as minimizing the total link utilization or the maximum link utilization. GENESIS performs coarse-grained TE, e.g., given information about diurnal traffic patterns, expected load (such as background systems workloads), plan how to route flows according to a global objective.

Min-tot TE. To perform traffic engineering, link capacities of the network $C(sw_1, sw_2)$ and traffic rates of the packet classes $\sigma(pc)$ are specified as input to GENESIS (we assume a single path for a packet class). The utilization of a link $U(sw_1, sw_2)$ is defined as the ratio of total traffic flowing through the link to the link capacity, and encoded using the theory of linear rational arithmetic as:

$$U(sw_1, sw_2) = \frac{\sum_{\forall pc} \mathtt{ite}(Fwd(sw_1, sw_2, pc), \sigma(pc), 0)}{C(sw_1, sw_2)} \tag{13}$$

The following objective minimizes the total link utilization:

$$\mathtt{minimize} \sum_{\forall sw_1, sw_2} U(sw_1, sw_2) \tag{14}$$

Min-max TE. To encode the TE objective of minimizing the maximum link utilization, we define a variable $maxU$ which represents the maximum link utilization. The constraints added to ensure that $maxU$ is greater than or equal to all individual link utilizations:

$$\forall sw_1, sw_2.\ \ maxU \geq U(sw_1, sw_2) \tag{15}$$

We then impose the following objective:

$$\mathtt{minimize}\ maxU \tag{16}$$

Multipath TE. GENESIS can support multipath-TE: for a packet class pc, we can create k subclasses and split traffic of pc among the

k paths adhering to the global objective. However, this requires a static split of traffic among the subclasses. By adding isolation policies to the k subclasses, we can split traffic of pc along edge-disjoint paths. Furthermore, GENESIS can be used for other quantitative objectives like minimizing total latency and load balancing traffic across middleboxes by using an encoding similar to the one presented in this section.

5.3 Handling Failures Gracefully

Another network management consideration for operators is the occurrence of failures (switches, links etc.), which are all too frequent in datacenter networks [15]. Failures require recomputation of paths compliant to the input policies for the modified topology. A naive approach is to use GENESIS to resynthesize the modified instance; However, the new solution may be drastically different from the original data plane, incurring a large overhead of removing old rules and installing new ones [17, 19].

In what follows, we describe two techniques to handle failures more gracefully. The first technique is data-plane resiliency (§5.3.1), which synthesizes and pre-installs resilient data planes, which even in the event of a bounded number of link failures, continue to satisfy input policies. This technique eliminates the need to resynthesize the forwarding rules for every failure event, but it requires extra backup rules on switches, and cannot capture global operator policies.

Thus, we propose a second mechanism called *minimal repair* (§5.3.2), which can transition from the disrupted data plane to a new policy-compliant one with minimal overhead by minimizing the number of switches whose rule tables are modified. Repair does not incur the extra rule cost of the first approach and can capture all GENESIS policies. It is also useful for accommodating incremental policy changes, which occur frequently in cloud datacenters [14]. The main drawback is that it still requires removal/installation of rules when a failure occurs, which can end up being expensive depending on the number of switches involved.

5.3.1 Dataplane Resiliency

In this section, we describe the transformation of input policies to provide dataplane *t-resilience* [33], i.e., in the event of up to t arbitrary link failures, the synthesized data plane still has a path for each packet class satisfying all policies which is achieved by synthesizing backup paths that satisfy input policies. This approach differs from randomized routing algorithms which provide resiliency [8] but do not take into account policy-compliance of the backup paths.

We only consider reachability, waypoint, and isolation policies in the input. Global policies like capacity policies and traffic engineering pose a difficulty in synthesis. For example, consider a packet class pc with a traffic rate of $\sigma(pc)$. By considering the backup paths with the same traffic characteristics for synthesis, the total traffic accounted for pc would be $c \times \sigma(pc)$ (for some constant c), leading to under-provisioning of resources. Our current resilience transformation has no provisions to avoid or minimize the under-provisioning of resources which affect capacity policies and TE objectives.

Given the physical topology $T = (S, L)$, we define a link-failure scenario θ as the set of failed links such that $\theta \subseteq L$. We define $\Theta(t)$ as the set of all failure scenarios where no more than t arbitrary links fail,—i.e. $\Theta(t) = \{\ \theta \ \mid \ |\theta| \leq t\}$. Given a packet class pc, we construct the induced data plane graph $\xi = (S, L_{pc})$ from the links of the paths returned by the synthesis algorithm for class pc. For a failure scenario θ, the active data plane $\xi_\theta = (S, L_{pc} \setminus \theta)$ represents all the links used by ξ which are unaffected by the failure scenario. A data plane ξ is *resilient* to θ if it contains a path from the source to destination for the packet class in the active data plane ξ_θ.

Definition 3 (Resilience). *A data plane $\xi = (S, L_{pc})$ for class pc is t-resilient if ξ is resilient to all $\theta \in \Theta(t)$.*

While resilience deals with existence of paths during failure scenarios, we extend the notion to include policy compliance.

Definition 4 (Policy-compliance). *A t-resilient data plane $\xi = (S, L_{pc})$ for class pc is policy-compliant if under any failure scenario $\theta \in \Theta(t)$, any path for pc in $\xi_\theta = (S, L_{pc} \setminus \theta)$ satisfies the input policies.*

Algorithm 1 Resilience Transformation

1: [Input] PC: Packet classes (Reachability/Waypoint policies)
2: [Input] I: Isolation policies (Traffic and Link types)
3: [Input] t: Maximum number of arbitrary link failures
4: [Output] PC^R, I^R: Transformed set of policies such that the synthesized data plane is *t-resilient* and policy-compliant

5: $PC^R, I^R \leftarrow \emptyset$
6: **for** $pc : \{src_{pc}, dst_{pc}, W_{pc}\} \in PC$ **do**
7:     // Create $t+1$ edge-disjoint paths of pc
8:     $\hat{pc} = \{rc_1, \ldots, rc_{t+1}\}$ s.t $\forall m.\ rc_m : \{src_{pc}, dst_{pc}, W_{pc}\}$
9:     $PC^R = PC^R \cup \hat{pc}$
10:     $I_{pc} = \{rc_m <> rc_n \mid \forall m, n \leq t+1 \land m < n\}$
11:     $I^R = I^R \cup I_{pc}$
12: **for** $i : pc_m < op > pc_n \in I$ **do**
13:     $\hat{i} = \{rc_1 < op > rc_2 \mid \forall rc_1 \in \hat{pc_m}, \forall rc_2 \in \hat{pc_n}\}$
14:     $I^R = I^R \cup \hat{i}$
15: **return** PC^R, I^R

Algorithm 1 shows how GENESIS can be used to provide *t-resilience*. The idea is to modify the input policies such that multiple disjoint paths satisfying the original policies are synthesized for each packet class. For *t-resilience*, a packet class pc needs at least $t + 1$ edge-disjoint paths from source to destination. We ensure this property holds by creating $t + 1$ new packet classes ($\hat{pc}$ in line 8) and use *link-isolation policies* amongst all pairs in $\hat{pc}$ (line 10) to create $t + 1$ edge-disjoint paths for pc. The synthesized data plane $\hat{\xi} = (S, L_{pc})$ for class pc is constructed from the paths in the resilient packet class set $\hat{pc} = \{rc_1, \ldots, rc_{t+1}\}$, i.e., $L_{pc} = \bigcup_{rc \in \hat{pc}} L_{rc}$. Each path of $\hat{pc}$ satisfies the reachability policy, and any arbitrary t link failure scenario cannot affect all $t + 1$ paths.

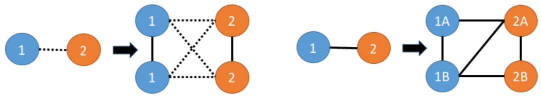
However, the resilient paths need to satisfy the input isolation policies with other packet classes (which themselves have $t+1$ paths for resilience). Thus, for a given policy $pc_1 || pc_2$, we add isolation policies to every pair of classes of $\hat{pc}_1$ and $\hat{pc}_2$ (line 13). This ensures that any path chosen in the data planes of pc_1 and pc_2 will be isolated from one another, thus providing policy-compliance under any arbitrary $t - link$ failure scenario. Figure 3(a) demonstrates an example transformation for providing $1 - resilience$.
We now that Algorithm 1 is sound.

Theorem 5.1 (Soundness). *Given input policies (PC, I), the data plane $\hat{\xi}_{pc}$ for every packet class $pc \in PC$ synthesized from transformed policies (PC^R, I^R) is* t-resilient *and policy-compliant.*

If there are no isolation policies in the input, the resilience transformation in lines 8-11 of Algorithm 1 is complete.

Theorem 5.2 (Completeness). *Given input policies (PC, I) such that $I = \emptyset$, the synthesized data plane ξ for a packet class pc is* t-resilient *if and only if it contains $t + 1$ edge-disjoint paths from source to destination for pc.*

When the original policies contain link-isolation policies, the policies from Algorithm 1 may return `unsat` even when a resilient data plane exists. Specifically, line 13 can add additional policies than is required for resilience. Figure 3(b) shows a transformation required for $1 - resilience$ with a smaller number of link-isolation policies among different classes of pc_1 and pc_2 than one obtained



(a) Traffic Isolation (b) Link Isolation

Figure 3: (a) Resilience Transformation for $pc_1 || \, pc_2$ for providing $1 - resilience$. The dotted lines represent traffic isolation policies, while the solid lines represent link isolation. (b) Example of a sufficient transformation for 1-resilience in the case of a link-isolation policy.

from Algorithm 1. Consider a failure scenario which disables path of pc_{1A}. By virtue of the link-isolation policies, pc_{1B} and pc_{2A} will be unaffected and can be used as paths for pc_1 and pc_2 respectively, and $pc_1 <> pc_2$ holds. Now suppose pc_{1B} is affected. Similarly, pc_{1A} and pc_{2A} can be used as the paths for the original packet classes. The same scenarios hold symmetrically for pc_2, and thus the resilience transformation can be achieved without adding link-isolation policies amongst all the packet classes.

5.3.2 Minimal Repair

Dataplane resiliency imposes high rule storage overhead on switches, and cannot accommodate global policies like link capacity bounds. As an alternative to it, we extend GENESIS's synthesis algorithm to perform minimal network repair using MaxSMT.

Formally, the MaxSMT problem is as follows: given a set of formulas $\Psi_0, \Psi_1, \ldots, \Psi_n$ with associated weights $w_1, \ldots, w_n$, find a subset $M \subseteq \{1, \ldots, n\}$ s.t: 1) $\Psi_0 \wedge \bigwedge_{i \in M} \Psi_i$ is satisfiable, and 2) The *award* $\sum_{i \in M} w_i$ is maximized. The constraints $\Psi_1, \ldots, \Psi_n$ denote *soft* constraints, and the associated weights w_i encode the award for including Ψ_i in the satisfying assignment.

We reduce the network repair problem to a MaxSMT problem and use soft constraints to minimize the number of switches on which rules need to be updated. Note that the disadvantage w.r.t. dataplane resiliency is that switches still require rule updates, which may take time depending on the number of switches involved.

Let the policy constraints generated by GENESIS for the new network state be Ψ_0, and let $\overline{Fwd}$ be the present data plane that does not satisfy Ψ_0. The objective is to find new Fwd which satisfies Ψ_0 while maximizing the number of *preserved switches* (switches whose rules are unchanged). If the rules on switch sw_i are preserved, then Fwd and $\overline{Fwd}$ have the same forwarding rules for all packet classes which traverse through sw_i. The following soft constraints capture this idea:

$$\Psi_{sw_i} = \bigvee_{\substack{\forall sw_j, pc \\ (sw_i, sw_j, pc) \in \overline{Fwd}}} Fwd(sw_i, sw_j, pc) \qquad w_{sw_i} = 1 \qquad (17)$$

The solution to this MaxSMT problem is a data plane that minimizes the number of switches whose rules have to be changed. Alternate repair objectives like minimizing the number of changed forwarding rules can be expressed similarly. Interestingly, the GENESIS's network repair mechanism can also be used to transform an existing non-compliant data plane to a policy-compliant one.

6. Tactics

Synthesizing a data plane translates to choosing paths from the solution space of all paths for each reachability policy such that the chosen paths satisfy all policies, e.g., waypoint traversal and isolation. Datacenter topologies, e.g., fat-trees [4], have numerous paths between edge switches to provide full bisection bandwidth. Thus, the solution space of paths for a pair of endpoints is large. For example, consider the fat-tree topology in Figure 4. The number of paths under length 10 between two edge switches in the same

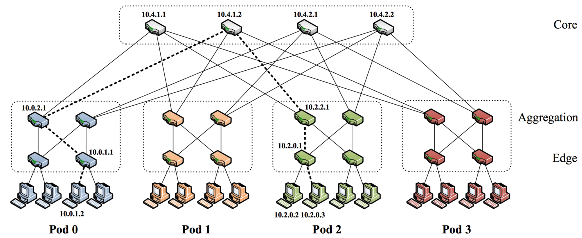


Figure 4: Fat Tree Topology with three-level hierarchy.

pod is 242 and between two edge switches in different pods is 272. If we consider the synthesis of n packet classes, the problem roughly translates to finding a solution in the space of size $(242)^n$. Operators can leverage the network structure of topologies to reduce the solution space by specifying undesirable path patterns. For example, the operator might require that a path between two edge switches in a fat-tree does not traverse another edge switch. This pattern doesn't drastically reduce the set of possible paths due to the dense interconnect between aggregate and core switches.

We introduce *tactics* (the name is inspired from the usage in SMT solvers, not proof assistants) on labels; abstractions that allow a network operator to impose restrictions on paths. We use the notion of mapping the set of switches to labels to have a coarse-grained way for specifying path patterns. Tactics on labels help create search strategies which can be used for groups of packet classes instead of individual switch-level patterns which lack generality.

Let Lb be the set of labels and S be the set of switches in the topology. Let $\phi : S \to Lb$ be the labeling function that maps each switch to a label in Lb. For e.g., we can leverage the hierarchical structure of the fat-tree by mapping all switches in the same level (core, aggregate or edge) to the same label. A path p is a word over the alphabet S. We define the path-labeling function $\Phi : S^* \to Lb^*$, which maps each switch in the path to its corresponding label. For example, given the path $p = e1 \; a2 \; c3 \; a4 \; e2$, the path-labeling function function produces $\Phi(p) = eacae$—maps each switch to its corresponding label. Here, e, a, and c stand for edge, aggregate, and core respectively.

6.1 Synthesis with Tactics

Tactics are simple regular expressions over the set of labels and are used to blacklist certain path patterns. Regular expressions have been previously used in tools like NetGen [29] to specify the paths for a packet class. While supporting full regular expressions is possible, it causes a blow-up in the solving time as further constraints need to be added to the solver to ensure that a path satisfies the regular expression. Rather than specifying how the path must look like, we use regular expressions on switch labels to specify *blacklists* i.e., what the path must *not* look like. A tactic, for example, can blacklist paths from an edge switch to an edge switch that go through another edge switch.

6.1.1 Restricted Tactic Syntax

We specify tactics using a restricted set of regular expressions[3] that not only do not require extra constraints to be added, but actually allow us to reduce the number of constraints in Ψ. Tactics are regular expressions described by the following grammar:

$$\begin{array}{rcl} R & := & \neg(l_{src}.^{i}C.^{*}l_{dst}) \\ C & := & \varepsilon \mid l_1 \mid l_1 l_2 \end{array}$$

where $l_i \in Lb$ and l_{src}, l_{dst} are used to specify the labels of the source switch and destination switch, respectively. Since our goal is

[3] It is a subset of star-free languages [12].

to blacklist paths, we allow regular expressions to be negated at the outer level.

Example 1. *$\neg(e.^{i}c.^{*}e)$ indicates that the path must not contain a core switch at the $(i+1)^{th}$ step. Similarly, $\neg(e.^{i}.^{*}e)$ indicates that the path connecting two edge switches should have a length $< i+1$.*

Let $\pi = sw_0 \dots sw_k$ be a path for packet class pc and let its labeling be $\Phi(\pi) = a_0 \dots a_k$. We say that π satisfies a tactic R, $\Phi(\pi) \in L(R)$, if the following holds:

- $\Phi(\pi) \in L(\neg R)$ iff $\Phi(\pi) \notin L(R)$;
- $\Phi(\pi) \in L(l_{src}.^{i}.^{*}l_{dst})$ iff $k \geq i+1$, $l_{src} = a_0$, and $l_{dst} = a_k$;
- $\Phi(\pi) \in L(l_{src}.^{i}l.^{*}l_{dst})$ iff $k \geq i+2$, $l_{src} = a_0$, $l_{dst} = a_k$, and $a_{i+1} = l$;
- $\Phi(\pi) \in L(l_{src}.^{i}l_1 l_2.^{*}l_{dst})$ iff $k \geq i+3$, $l_{src} = a_0$, $l_{dst} = a_k$, $a_{i+1} = l_1$, and $a_{i+2} = l_2$.

In GENESIS, operators can specify conjunctions of tactics which adhere to the restricted syntax and the synthesis algorithm is modified to enforce the tactics.

Example 2. *The "No Edge" tactic ensures that a edge-edge path cannot traverse another edge switch. It is expressed using conjunctions of tactics: $\neg(e.^{*}e.^{*}e) \equiv \bigwedge_{i=0}^{i=\mu-2} \neg(e.^{i}e.^{*}e)$ where μ is the limit on path length. The "Valley-free" Tactic: $\neg(e.^{5}.^{*}e) \wedge \neg(e.^{*}e.^{*}e)$ ensures that a edge-edge path is of the form $e-a-c-a-e$.*

Paths used in production datacenter networks adhere to both these tactics [16, 30]. Such paths are simple and make networks easy to manage and troubleshoot [6].

6.1.2 Modified Synthesis Algorithm with Tactics

In our synthesis algorithm, the reachability-propagation constraints (Equation (3)) construct the path from destination to source. We use tactics to prune these constraints, so that the path synthesized satisfies the tactic regular expression.

The tactic set $\Gamma = \{(R_1, pc_1), \dots, (R_n, pc_n)\}$ specifies that tactic R_i is applied on packet class pc_i where $pc_1, \dots, pc_n \in PC$ and $R_1, \dots, R_n$ are regular expressions satisfying the restricted tactic syntax. Given a tactic R applied to a packet class pc, we define $\Psi_T(R, pc)$ as the additional SMT constraints used for synthesis such that $(\Psi \wedge \bigwedge_{(R,pc)\in\Gamma} \Psi_T(R, pc))$ is provided as input to the SMT solver. Note that $\Psi_T(R, pc)$ is presented as additional SMT constraints only for clarity. In practice, the modified synthesis algorithm will remove constraints for each $(R, pc) \in \Gamma$.

Type 1. For a tactic R of the form $\neg(l_{src}.^{i}.^{*}l_{dst})$ applied to pc:

$$\Psi_T(R, pc) = \forall sw, k \geq i+1. (sw, pc, k) \notin Reach \tag{18}$$

This tactic restricts the path to a length $< i+1$. Thus, we can remove the reachability constraints of Equation (3) for all the tuples $(sw, pc, k) \notin Reach$ satisfying Equation (18) as they cannot contribute to any path satisfying the tactic.

Type 2. For a tactic R of the form $\neg(l_{src}.^{i}l.^{*}l_{dst})$ applied to pc:

$$\Psi_T(R, pc) = \forall sw.\ \phi(sw) = l \wedge sw \neq dst \implies (sw, pc, i+1) \notin Reach \tag{19}$$

The tactic ensures that a switch with label l cannot be reached in $i+1$ steps, except if $l = l_{dst}$. In that case, only the destination switch with label l can be reached in $i+1$ steps as the path with labeling $l_{src}.^{i}l_{dst}$ satisfies the tactic. If $l \neq l_{dst}$, then all switches with label l cannot be reached in $i+1$ steps. For all tuples $(sw, pc, i+1) \notin Reach$ satisfying Equation (19), we can remove the reachability constraints of Equation (3).

Type 3. For a tactic R of the form $\neg(l_{src}.^{i}l_1 l_2.^{*}l_{dst})$ applied to pc:

$$\Psi_T(R, pc) = \forall n_1, n_2.\ \phi(n_1) = l_1 \wedge \phi(n_2) = l_2 \wedge n_2 \neq dst \implies \neg(Reach(n_1, pc, i+1) \wedge Fwd(n_1, n_2, pc)) \tag{20}$$

This tactic ensures that a switch with label l_1 at $i+1$ in the path will not forward the packet to a switch with label l_2 (unless n_2 is the destination). To enforce this, we modify the Equation (3) and remove all $l_1 \to l_2$ edges at position $i+1$ in the path for which the switch with label l_2 is not the destination.

We now state the soundness and completeness of the synthesis algorithm with tactics. Let $(Fwd, Reach)$ be a model of Ψ and $\Pi = \texttt{paths}\ (Fwd, Reach)$ be the set of induced paths (from Definition 1).

Theorem 6.1 (Soundness)**.** *For a tactic set Γ, if $(Fwd, Reach) \models \Psi \wedge \bigwedge_{(R,pc)\in\Gamma} \Psi_T(R, pc)$), then $\forall (R, pc) \in \Gamma.\ \forall (\pi', pc') \in \Pi.\ pc = pc' \implies \Phi(\pi') \in L(R)$.*

Theorem 6.2 (Completeness)**.** *For a tactic set Γ, if $\Pi \models \Psi$ and $\forall (R, pc) \in \Gamma.\ \forall (\pi', pc') \in \Pi.\ pc = pc' \implies \Phi(\pi') \in L(R)$ then $\forall (R, pc) \in \Gamma. (Fwd, Reach) \models \Psi_T(R, pc)$.*

The intuition behind the restricted tactic syntax comes from the structure of the reachability propagation constraints (Equation (3)) which construct the path for a packet class. Each constraint enforces that if a switch is reachable in k steps in a path, there must a neighbour switch in the path reachable at $k-1$ steps. Using the $Reach$ relations, we can specify path length restrictions (Type 1) or prevent switches with a certain label at some position (Type 2). The structure of Equation (3) restricts regular expressions on only local switch neighbours (Type 3). The structure of these constraints prevents us from being able to specify unrestricted regular expressions (supporting these would require adding additional constraints).

Example 3. *Consider a tactic $\neg(e.^{i}aca.*e)$. To enforce this tactic, we need to have constraints which prevent the path reaching an aggregate switch in $i+3$ steps when the path traverses an aggregate and core switch at $i+1$ and $i+2$ steps respectively. This cannot be specified by modifying the reachability constraints in its current form, because the constraints for reachability for a switch in $i+3$ steps only depends on the constraints for reachability in $i+2$ steps.*

Tactics are heuristics, but well-defined ones with formal semantics and provable soundness properties. In practice, tactics can be used to specify restrictions which would not reduce the search space dramatically, but are still useful toward speeding up the synthesis, especially in datacenter topologies which are hierarchical and can be used to specify interesting tactics. One of the biggest advantages of tactics is that it is *policy-agnostic* since it enforces conditions on the path, and can be used in conjunction with the different policies supported by GENESIS (isolation, traffic engineering etc.). Thus, we have provided a framework for the development of search strategies based on path properties, and operators can design tactics based on the physical topologies (datacenter topologies are hierarchical) to create a library of tactics that can be reused for workloads. While tactics sacrifice completeness, operators can discard the tactic if synthesis fails and use GENESIS without tactics.

7. Divide-and-Conquer Synthesis

Since the complexity of finding a data plane enforcing policies is *exponential* in the number of packet classes, the synthesis time shoots up with increasing packet classes. However, since datacenter topologies have a dense interconnection of links between layers there can be numerous different data planes as solutions. We propose to speed up synthesis by partitioning the problem into smaller components.

Pseudocode 2 Divide-and-Conquer Synthesis

```
1:  procedure DCSyn(P)
2:      if size(P) < P_thres then
3:          Apply normal synthesis on P
4:      else
5:          Partition P into P_1 and P_2
6:          if interpartition edges > E_thres then
7:              Apply normal synthesis on P
8:          F = []          /* Failed solutions */
9:          attempts = 0
10:         while attempts < RA_max do
11:             sol_1 = Apply synthesis on P_1 such that sol_1 ∉ F
12:             if synthesis(P_1) fails then
13:                 return DCSyn failure
14:             sol_2 = Apply synthesis on P_2 such that
                    sol_1 ∪ sol_2 is a solution for P
15:             if synthesis(P_2) fails then
16:                 F.append(unsat-cores(P_2))
17:                 attempts++
18:             else
19:                 return DCSyn success
20:         return DCSyn failure
```

Suppose we have two packet classes pc_1, pc_2 isolated from one another; the standard synthesis algorithm adds constraints for both packet classes to Ψ and finds the solution for both classes. Instead, we could synthesize pc_1 independently and, after that, find a solution for pc_2 that is isolated from the path obtained for pc_1. However, synthesis of pc_2 can fail because the solution of pc_1 may be such that there is no way to place an isolated path for pc_2 but if they had been synthesized together, a solution might exist. For combat this, we use a recovery mechanism which uses unsatisfiable cores extracted from the solver. We term this procedure *divide-and-conquer* synthesis.

We define a policy graph $P = (R, I)$ where every vertex $r \in R$ is a packet class for a reachability/waypoint policy and edges denote isolation constraints between packet classes. An edge $(r_1, r_2) \in I$ means that the paths of $r1$ and $r2$ are isolated from each other. We assume that there are no capacity policies in the input specifications. Given the policy graph P, we can synthesize each connected component independently, since packet classes in different connected components are not related by any isolation policy, and therefore are independent of each other.

We describe the divide-and-conquer synthesis procedure in Pseudocode 2, which takes as input a connected component of the policy graph. The crux of the procedure is that we partition each connected component P into two smaller components P_1 and P_2 and do the following: 1) synthesize P_1 and obtain a solution S_1, and 2) for packet classes of P_2 that are isolated to packet classes in P_1, add the solution S_1 as a constraint to ensure that the packet classes in P_2 will not share the edges of the respective paths obtained in P_1.

We use the *min-cut* to partition the policy graph connected component into two such that the number of edges between the partitions is minimized. The rationale is that since we intend to perform the synthesis of both partitions separately, the partitioning should maximize isolation policies within components and minimize those across components. By maximizing isolation policies during synthesis of the component, the partial solution is more likely to be compatible with a complete solution. However, if the min-cut partitioning produces a component smaller than a threshold size, we perform partitioning of the graph into two *equal sized* partitions and minimize the cut edges between the partitions. We need to ensure that we don't partition the graph smaller than a threshold, as the partial solutions obtained by synthesis of very small partitions are more likely to conflict with other packet classes. GENESIS performs divide-and-conquer synthesis recursively on the components till we cannot partition the component further.

Solution Recovery. While in the best case divide-and-conquer synthesis leads to a great increase in performance, we need a recovery mechanism in case we cannot find compatible partial solutions. Many SMT solvers track constraints and return an unsatisfiable core [9] when synthesis fails. Informally, the unsatisfiable core is a set of tracked constraints that describes why there wasn't a feasible solution. This helps us track failed partial solutions. Thus, if synthesis of P_2 fails, the unsatisfiable cores describe what paths of the solution of P_1 are causing the synthesis of P_2 to fail. When performing synthesis of P_1 again, we therefore ensure that we get *different* paths from those extracted from the unsatisfiable cores. Basically, we perform a *solver-guided* enumeration of different solutions of P_1 to find a satisfying solution for P_2. The solution recovery procedure is described in lines 10- 19.

Since, recovery is a form of enumeration, in cases where the graph has greater number of policies (clique), finding a solution could take a large number of enumerations, while synthesis without partitioning would provide a solution faster. Thus, we bound the number of enumerations performed by the recovery mechanism and return failure if we don't obtain a solution.

Divide-and-conquer synthesis with recovery is sound, but it is incomplete as we bound the number of enumerations. The success of this approach is directly related to the size of the components (determined by P_{thres}). This is because, by synthesizing more packet classes together, we decrease the conflicts arising between partial solutions. The extreme case of when we do not partition the component at all (normal synthesis) is complete. To make the synthesis complete with faster convergence, we perform iterations of divide-and-conquer synthesis, and at each iteration we double the partition threshold P_{thres} if the previous iteration failed. This scheme tries to balance the trade-off between completeness, which requires larger components, and performance, as synthesis is faster on smaller components. In the extreme case, after $O(log\ P)$ iterations, $P_{thres} > P$ and divide-and-conquer will not partition P and yield the solution (if one exists).

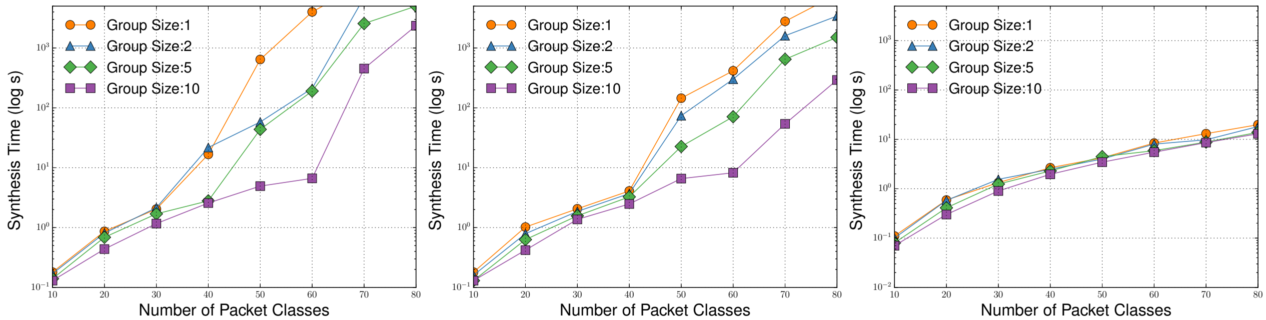
Divide-and-conquer is more *effective* when there is a large number of solutions and partial solutions do not fail. When the problem is highly constrained and the number of solutions is low, the recovery mechanisms and multiple iterations could lead to a degraded performance. A drawback of the divide-and-conquer approach is that it is difficult to apply to global policies (like traffic engineering) primarily because splitting the input problem isn't easy; development of strategies to speed-up global policies is future work.

8. Evaluation

We implemented a full working prototype of GENESIS in Python. We have implemented an interpreter for the Genesis Programming Language using PLY [3] and the synthesizer using the SMT solver Z3 [11] and its νZ extension for MaxSMT and linear optimization [7]; GENESIS outputs the forwarding rules for the switches, which can be provided as input to a SDN controller (e.g., Floodlight [1]) to install over the network. GENESIS uses the Metis graph-partitioning library [21] to perform equi-sized partitioning used by divide-and-conquer synthesis.

In this section, we evaluate GENESIS using enterprise-scale multi-tenant data center settings. Specifically, we ask:

- What is the performance of GENESIS's baseline synthesis algorithm for tenant policies? How does the performance vary with size of the network, number and the nature of policies in use? (§8.1)



(a) Baseline (b) No Edge Tactic (c) Valley-free Routing Tactic

Figure 5: Total synthesis time (log scale) for isolation workloads over range of packet classes and different tenant-group sizes.

- What is the performance of GENESIS for operator policies like capacity bounds, traffic engineering, and network repair which use SMT with linear optimization objectives and MaxSMT? (§8.2)
- How much do tactics help improve GENESIS's performance? Which tactics offer the best improvement? (§8.3)
- To what extent does the divide-and-conquer synthesis improve GENESIS's performance? When does it lead to degraded synthesis times? (§8.4)

Our experiment settings have a few thousand servers, tens of switches, and hierarchical fat-tree network topologies which reflect a private datacenter. Our experiments are parameterized by: (a) total size of the fat-tree network (45-180 switches), (b) number of tenants (1-80), and (c) number of packet classes in a tenant (1-10). Note that a single packet class can be used to specify policy for multiple host-pairs of a tenant connected to the same edge switches, and placement of the hosts can take uniformity of policy in account to reduce the explosion of packet classes with increasing hosts.

Our primary metric of interest is synthesis time, measured in seconds. In measuring this, we focus on the time the Z3 solver takes to solve the constraints[4]. All experiments were conducted using a 32-core Intel-Xeon 2.40GHz CPU machine and 128GB of RAM. For evaluating the baseline performance, we impose a synthetic limit on the path length μ to be 10, which is adequate for a fat-tree topology with three levels.

8.1 Baseline Synthesis Performance for Tenant Policies

Multi-Tenant Isolation. To evaluate the baseline performance of GENESIS, we model a multi-tenant 80 switch topology with tenant-isolation in Figure 5(a). For each workload we have n tenants with group size g which is the number of packet classes for each tenant. The x-axis shows the total packet classes $n * g$. Packet classes of a tenant are not isolated (and they implement simple reachability within the tenant), while packet classes of different tenants are traffic-isolated. Thus, no two tenants share a link in the same direction, and can never affect each other's performance. We randomly[5] place endpoints for the tenants' packet classes, ensuring that no more than 4 tenants share a single edge switch. Operators can aggregate a tenant's traffic from multiple instances connected to the same switches as a single reachability policy and establish pathways for communication amongst different switches.

[4] We do not account for constraint generation time in our evaluation, as it has polynomial time complexity and thus, can scale well unlike constraint solving time; a well-engineered system can considerably reduce the constraint generation overheads.

[5] Smarter placement of tenants could speed-up synthesis as tenant endpoints would be located closer to each other. The placement algorithm can be used to develop specialized tactics.

For a fixed group size, we observe that the total synthesis time increases exponentially with number of packet classes. As group size decreases, for the same number of classes, the number of tenants increases, increasing the number of isolation policies and the synthesis times. Group size 1 denotes the extreme case where all flows are isolated with each other.

While we evaluated a multi-tenant isolation setting, there are other scenarios that translate to these workloads. Consider an example where specific flows of tenants require QoS guarantees and these flows must be isolated w.r.t. all other flows. This translates to a two-tenant isolation setting. Operators can provide weaker isolation such that two flows must be isolated on only certain "special" links. This is an easier problem to tackle than isolation over all links, and the performance of such scenarios would be better. Failure resiliency uses link-isolation policies which exhibit a similar performance compared to the workloads considered here.

Effect of Topology Size. To evaluate GENESIS across increasing topology sizes for isolation workloads, we fix the tenant-group size to 5, and for each topology, we maintain the ratio of packet classes to number of edge-aggregate links to 0.25. We choose this metric because if we keep the number of classes constant, as topology sizes increases, it is easier to find isolated paths due to more links. Thus, by keeping the number of packet classes proportional to size of the topology, we maintain the relative difficulty of the workload across topologies. We show the average synthesis time per class with increasing topology sizes in Figure 6 (baseline trace). We are able to synthesize forwarding rules for 12 tenants with group size 5 in a 125 switch topology in 124 seconds (avg. 2 seconds per traffic class). We also observe that average time per flow increases exponentially with larger topologies, thus synthesis times are also exponential in the number of switches.

Number of Way-points	Avg. Synthesis time per Class (s)
1	0.034
2	0.138
3	0.983
4	15.41
5	32.93

Table 2: Average synthesis time per class for waypoint policies with increasing number of waypoints.

Waypoint Policies. To evaluate GENESIS's performance for ordered sets of waypoints, we fix the number of waypoints (range:1-5) and generate 100 waypoint policies with different sizes and permutations of the ordered waypoint sets for a 80 switch topology. Each policy has edge switches as endpoints and randomly picked core or aggregate switches for waypoints. The synthetic limit μ on the path length is increased to 15 and no tactics are used (difficult to devise a tactic for the path satisfying a waypoint policy). The average synthesis time for a waypoint policy is reported in Table 2. We observe that synthesis time increases exponentially with total number of waypoints in a packet class's policy, owing to the com-

Workload Type	Description	Time (s)
minimize-avg-te	100 packet classes	425
	200 packet classes	2002
minimize-max-te	25 packet classes	522
	50 packet classes	4192
Network repair	8 tenants, group size 10, tenant-isolation, 1-switch failure	219

Table 3: Synthesis times for workloads on a 80-node fat-tree topology with different optimization objectives.

plexity of the problem. GENESIS can synthesize rules for a path with 3 total waypoints in less than a second, on average.

8.2 Baseline Synthesis Performance for Operator Policies

Isolation with Link Capacity Policies. Figure 8 (baseline trace) shows the average synthesis time per flow for the same setting as above, but additionally, there are 10 low-bandwidth links in the network for which the operator specifies capacity policies (all packet classes have uniform capacity). Since we use LRA for link capacity constraints, we see an increase in average time for synthesis when compared to pure isolation which is completely encoded using SAT.

Traffic Engineering. Table 3 shows the synthesis time for workloads on a 80-node fat-tree topology with different traffic engineering (TE) objectives. GENESIS can synthesize a data plane minimizing average utilization for 200 packet classes in approximately 2000 seconds. However, for minimizing the maximum link utilization, GENESIS can only synthesize 50 packet classes in close to 4000 seconds. For both objectives, the synthesis time increases exponentially with the number of packet classes. SMT with optimization objectives is an emerging field of research, and we envision that solvers in the future will become fast and handle larger workloads.

Minimal Repair. To evaluate the performance of minimal repair using MaxSMT, we consider a setting with 8 tenants, each with 10 packet classes (total classes=80), and tenant flows are isolated from one another. Now, we disable the switch with the largest number of rules, and try to find a new data plane satisfying the original tenant isolation policies such that the number of switches unaffected is maximized. We can synthesize the minimal repair in nearly 200 seconds on average. With repair, the new data plane only changes rules on 2-3 switches on average, while naive synthesis results in nearly 60 switches being updated, which is very expensive.

8.3 Tactic Reductions

We demonstrate the improvements from using tactics for isolation workloads with different number of tenants and group sizes on a 80 switch topology.

"No Edge" Tactic: Figure 5(b) shows the synthesis time for isolation workloads using the no edge tactic $\neg(e.^*e.^*e)$, which has a best-case speedup of 9.5× over baseline synthesis. Using this tactic, GENESIS can synthesize forwarding rules for 12 tenants with group size 5 in under 200 seconds.

"Valley-free" Tactic: For the same isolation workloads as above, we use the tactic $\neg(e.^5.^*e) \wedge \neg(e.^*e.^*e)$ which ensures *valley-free routing*, that is paths are of the form $eacae$. The results are shown in Figure 5(c). Using this tactic, GENESIS synthesizes forwarding rules for each workload in under 20 seconds and can achieve a best-case reduction of 400× compared to synthesis without tactics.

Effect of Topology Size: In Figure 6, we evaluate the performance of different tactics for different topology sizes. There is a significant reduction in synthesis time for each tactic when compared to the baseline synthesis. The performance of each tactic is directly related to the reduction of the search space: more restrictive tactics have lower synthesis times. Using the $length \leq 7$ tactic and "no edge"

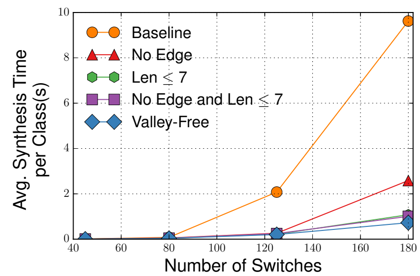


Figure 6: Average synthesis time per packet class versus topology size for isolation workloads w/o different tactics with the ratio of packet classes to number of edge-aggregate links 0.25.

tactic, GENESIS synthesizes forwarding rules for 20 tenants of group-size 5 in 100 seconds in a 180 switch topology (9× speedup over synthesis without tactics).

Isolation with Link Capacity Policies: A similar setup with additional link capacity constraints for 10 links is evaluated using the no edge tactic, and we get a best-case 14× improvement over baseline synthesis. Tactics can provide a considerable improvement over the baseline performance as illustrated by these experiments, and demonstrate the viability of synthesis approach of GENESIS to real-world networks.

8.4 Divide-and-Conquer (DC) Synthesis Performance

To evaluate the divide-and-conquer (DC) synthesis procedure, we perform 100 runs of DC and non-DC synthesis (with the no edge tactic in both cases) on isolation workloads with varying number of tenants and different group sizes used in §8.1. We compute the speedup (time of non-DC synthesis/time of DC synthesis) and plot its cumulative frequency distribution in Figure 7 to quantify the benefits of DC synthesis. For more than 80% of the workloads, divide-and-conquer offers better or comparable performance to non-DC synthesis, achieving a speedup of 2× for nearly 40% of the workloads. For 20% of the workloads, divide-and-conquer performs worse than the non-DC approach, especially for workloads with tenant group size 1 due to greater number of recovery attempts.

Summary. The key points of our evaluation are: 1) For a representative tenant-group size of 10 in a 80 switch fat-tree, the baseline synthesis performance for synthesizing the forwarding rules for 1 to 8 tenants with complete tenant-isolation is in 0.1-2000s. 2) Operator policies like optimization objectives for TE and network repair is more expensive than synthesis without objectives. 3) Tactics provide considerable speedup over the baseline synthesis. We can synthesize the above workloads in 0.1-300s using the no edge tactic, and under 12s using the valley-free routing tactic. 4) GENESIS can further benefit from divide-and-conquer (DC) synthesis, which provides a 2.0× speed-up over non-DC synthesis in 40% of the workloads, in addition to the tactic improvements.

9. Related & Future Work

One Big Switch: Kang et. al [20] tackle a similar problem of flow policy enforcement. However, their end-point policies deal with simple reachability. Their rule placement algorithm takes the path of the flow in the network (called the routing policy) as an input. Zhang et. al [35] build on the "one big switch" abstraction [20] to optimize for the specific case of distributed firewall policy enforcement using ILP. PGA [26] provides a graph-level abstraction for specifying network policies like ACLs and middlebox service chaining. However, PGA abstracts the underlying network as "one big switch" and cannot be used to compose policies like tenant isolation or traffic engineering.

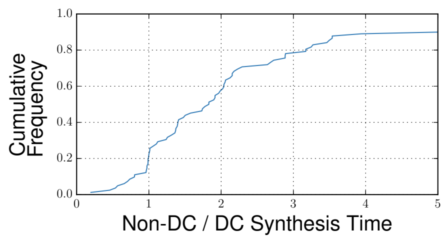


Figure 7: CDF for speedup achieved by divide-and-conquer synthesis.

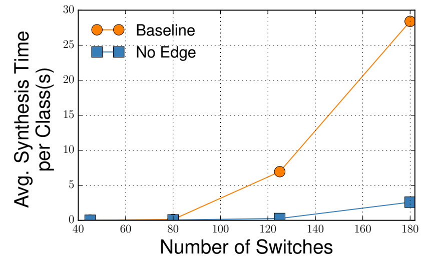


Figure 8: Average synthesis time per packet class versus topology size for isolation workloads with the ratio of packet classes to number of edge-aggregate links 0.25 and 10 low bandwidth links in the topology have capacity policies.

Controller synthesis: Program synthesis has seen limited applications to SDN controllers [24, 34]. These systems synthesize the behavior of individual switches (e.g., learning switches or firewalls); furthermore, these techniques apply to networks operating in a reactive mode (where the first packet of a connection is processed by the controller to determine the actions to employ). Such switch-centric approaches are too constraining and cannot be applied to realize network-wide objectives considered in GENESIS. Synthesis has been also used for generating consistent network updates [22, 36]. But this problem is orthogonal to policy enforcement.

Policy languages: The closest approaches to ours are Merlin [32], NetGen [29] and NetKAT [5]. In Merlin, data planes that adhere to policies expressed using regular expressions are synthesized by first intersecting the topology with the regular expressions appearing in the policies and then encoding reachability in the intersected graph using mixed integer linear programming (ILP). Merlin supports minimum and maximum bandwidth guarantees. In its current iteration, Merlin's encoding does not support isolation policies, but we believe that it could be extended to support them. A more prominent difference arises with unordered waypoint policies: expressing a policy including a waypoint set W of size k requires a regular expression of size exponential in k as all the possible permutations of the elements of W must be considered. This fact clearly impacts the performance of Merlin's compiler that would have to generate a mixed ILP with a large number of variables. In GENESIS, this is not the case as waypoints can be encoded with polynomially many constraints. While this does not affect the theoretical complexity, our compiler does not incur an a-priori exponential blow-up; it rather relies on the power of SMT solvers to guide the search. This is one of the main aspects behind our decision of not using regular expressions to express policies. GENESIS uses a restricted form of regular expressions as tactics that leverage the network topology. While in Merlin, regular expressions *increase* the number of constraints generated by the compiler, tactics *decrease* the number of generated constraints, therefore speeding up the search. To the best of our knowledge, this is the first use of constraints that leverages the topology structure to simplify the search.

In NetGen, network updates that adhere to policies expressed using regular expressions are synthesized using SMT solvers. Given a specification which mentions the packet classes, the old path, and the new path, NetGen solves the network change problem using an SMT solver. Due to the use of regular expressions NetGen also suffers the limitations we just discussed for Merlin. Interestingly, NetGen uses a specific encoding of regular expressions based on uninterpreted functions that helps reduce the number of constraints. While this encoding is fast when updating a single path, we do not see a way to extend it to our global synthesis setting. A crucial aspect of NetGen is that in its problem formulation each path can be synthesized independently and without affecting the other already synthesized paths. This is not the case when supporting isolation policies: if an old path needs to be moved to satisfy a new policy (e.g., because a link is under maintenance), re-synthesizing such a path can require re-synthesizing other paths.

NetKAT is a domain-specific language and logic for specifying and verifying network packet-processing functions for SDN, based on Kleene algebra with tests (KAT). Semantically, a NetKAT predicate and policy is a function that takes a packet history and produces a set of (possibly empty) packet histories. NetKAT can be used to express certain network-wide policies like reachability, waypoints using regular expressions for describing the paths, and programs on virtual topologies; it uses BDDs and symbolic automata to translate global programs to local switch programs [31]. However, the NetKAT semantics cannot be used to express policies based on hyperproperties [10], i.e., the packet processing function requires multiple packet histories as input. Traffic engineering or isolated paths are policies based on hyperproperties.

Future directions: Fine-grained traffic engineering based on online demand/flow size estimation and rapid rerouting is also crucial for datacenter workloads, and extending GENESIS's TE policies to fine-grained timescales is subject of future work. Also, the performance of SMT solvers with optimization objectives is quite slow, and calls for domain-specific techniques to speed up the synthesis. Also, datacenter networks are highly symmetrical, and this symmetry can be leveraged to speed up synthesis (similar to the work of Plotkin et. al [25] to speed up network verification using symmetry). The main challenges of using symmetry in synthesis is considering two aspects of symmetry: network symmetry and policy symmetry. Also, our treatment of resilience synthesis is preliminary and future work will be geared towards synthesizing resilient forwarding planes incorporating capacity constraints and traffic engineering.

10. Conclusion

We presented GENESIS, a general and extensible network management system for multi-tenant datacenter networks. It allows rich policies to be specified declaratively. It leverages the formal reasoning foundations of constraint solving together with fast SMT solvers to synthesize data plane configuration from high level policies. GENESIS abstracts away the difficult task of programming or configuring individual switches. GENESIS incorporates novel ideas to significantly speed up synthesis, leveraging the hierarchical nature of datacenter network topologies and the structure of the interaction between tenants' policies.

Acknowledgments

We thank the anonymous reviewers, Nate Foster, Aaron Gember-Jacobson, Raajay Viswanathan and Brent Stephens for their insightful feedback and suggestions. Kausik, Loris and Aditya are supported by the Wisconsin Institute on Software-defined Datacenters of Madison and grants from Google and National Science Foundation (CCF-1637516, CNS-1302041, CNS-1330308, CNS-1345249).

References

[1] Floodlight sdn controller. http://www.projectfloodlight.org/floodlight/.

[2] Intent: Don't tell me what to do! (tell me what you want). https://www.sdxcentral.com/articles/contributed/network-intent-summit-perspective-david-lenrow/2015/02/.

[3] Python lex-yacc. http://www.dabeaz.com/ply/.

[4] M. Al-Fares, A. Loukissas, and A. Vahdat. A scalable, commodity data center network architecture. In *Proceedings of the ACM SIGCOMM 2008 Conference on Data Communication*, SIGCOMM '08, pages 63–74, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-175-0. doi: 10.1145/1402958.1402967. URL http://doi.acm.org/10.1145/1402958.1402967.

[5] C. J. Anderson, N. Foster, A. Guha, J.-B. Jeannin, D. Kozen, C. Schlesinger, and D. Walker. Netkat: Semantic foundations for networks. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '14, pages 113–126, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2544-8. doi: 10.1145/2535838.2535862. URL http://doi.acm.org/10.1145/2535838.2535862.

[6] T. Benson, A. Akella, and D. Maltz. Unraveling the complexity of network management. In *Proceedings of the 6th USENIX Symposium on Networked Systems Design and Implementation*, NSDI'09, pages 335–348, Berkeley, CA, USA, 2009. USENIX Association. URL http://dl.acm.org/citation.cfm?id=1558977.1559000.

[7] N. Bjorner and A.-D. Phan. νz - maximal satisfaction with z3. In T. Kutsia and A. Voronkov, editors, *SCSS 2014. 6th International Symposium on Symbolic Computation in Software Science*, volume 30 of *EPiC Series in Computer Science*, pages 1–9. EasyChair, 2014.

[8] M. Chiesa, A. Gurtov, A. Madry, S. Mitrovic, I. Nikolaevskiy, M. Shapira, and S. Shenker. On the Resiliency of Randomized Routing Against Multiple Edge Failures. In *43rd International Colloquium on Automata, Languages, and Programming (ICALP 2016)*, volume 55 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 134:1–134:15, Dagstuhl, Germany, 2016. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. ISBN 978-3-95977-013-2. doi: http://dx.doi.org/10.4230/LIPIcs.ICALP.2016.134. URL http://drops.dagstuhl.de/opus/volltexte/2016/6269.

[9] A. Cimatti, A. Griggio, and R. Sebastiani. Computing small unsatisfiable cores in satisfiability modulo theories. *J. Artif. Int. Res.*, 40 (1):701–728, Jan. 2011. ISSN 1076-9757. URL http://dl.acm.org/citation.cfm?id=2016945.2016964.

[10] M. R. Clarkson and F. B. Schneider. Hyperproperties. *Journal of Computer Security*, 18(6):1157–1210, 2010.

[11] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3-540-78799-2, 978-3-540-78799-0. URL http://dl.acm.org/citation.cfm?id=1792734.1792766.

[12] V. Diekert and P. Gastin. First-order definable languages. In *Logic and Automata: History and Perspectives, Texts in Logic and Games*, pages 261–306. Amsterdam University Press, 2008.

[13] N. Foster, R. Harrison, M. J. Freedman, C. Monsanto, J. Rexford, A. Story, and D. Walker. Frenetic: A network programming language. In *Proceedings of the 16th ACM SIGPLAN International Conference on Functional Programming*, ICFP '11, pages 279–291, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0865-6. doi: 10.1145/2034773.2034812. URL http://doi.acm.org/10.1145/2034773.2034812.

[14] A. Gember-Jacobson, W. Wu, X. Li, A. Akella, and R. Mahajan. Management plane analytics. In *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, IMC '15, pages 395–408, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3848-6. doi: 10.1145/2815675.2815684. URL http://doi.acm.org/10.1145/2815675.2815684.

[15] 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. ISBN 978-1-4503-0797-0. doi: 10.1145/2018436.2018477. URL http://doi.acm.org/10.1145/2018436.2018477.

[16] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 51–62, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-594-9. doi: 10.1145/1592568.1592576. URL http://doi.acm.org/10.1145/1592568.1592576.

[17] K. He, J. Khalid, A. Gember-Jacobson, S. Das, C. Prakash, A. Akella, L. E. Li, and M. Thottan. Measuring control plane latency in sdn-enabled switches. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 25:1–25:6, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3451-8. doi: 10.1145/2774993.2775069. URL http://doi.acm.org/10.1145/2774993.2775069.

[18] V. Heorhiadi, M. K. Reiter, and V. Sekar. Simplifying software-defined network optimization using sol. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 223–237, 2016.

[19] X. Jin, H. H. Liu, R. Gandhi, S. Kandula, R. Mahajan, M. Zhang, J. Rexford, and R. Wattenhofer. Dynamic scheduling of network updates. In *Proceedings of the 2014 ACM Conference on SIGCOMM*, SIGCOMM '14, pages 539–550, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2836-4. doi: 10.1145/2619239.2626307. URL http://doi.acm.org/10.1145/2619239.2626307.

[20] N. Kang, Z. Liu, J. Rexford, and D. Walker. Optimizing the "one big switch" abstraction in software-defined networks. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 13–24, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2101-3. doi: 10.1145/2535372.2535373. URL http://doi.acm.org/10.1145/2535372.2535373.

[21] G. Karypis and V. Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM J. Sci. Comput.*, 20(1):359–392, Dec. 1998. ISSN 1064-8275. doi: 10.1137/S1064827595287997. URL http://dx.doi.org/10.1137/S1064827595287997.

[22] J. McClurg, H. Hojjat, P. Černý, and N. Foster. Efficient synthesis of network updates. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2015, pages 196–207, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3468-6. doi: 10.1145/2737924.2737980. URL http://doi.acm.org/10.1145/2737924.2737980.

[23] C. Monsanto, J. Reich, N. Foster, J. Rexford, and D. Walker. Composing software-defined networks. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, nsdi'13, pages 1–14, Berkeley, CA, USA, 2013. USENIX Association. URL http://dl.acm.org/citation.cfm?id=2482626.2482629.

[24] O. Padon, N. Immerman, A. Karbyshev, O. Lahav, M. Sagiv, and S. Shoham. Decentralizing sdn policies. In *Proceedings of the 42Nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '15, pages 663–676, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3300-9. doi: 10.1145/2676726.2676990. URL http://doi.acm.org/10.1145/2676726.2676990.

[25] G. D. Plotkin, N. Bjørner, N. P. Lopes, A. Rybalchenko, and G. Varghese. Scaling network verification using symmetry and surgery. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL 2016, pages 69–83, New York, NY, USA, 2016. ACM. ISBN 978-1-4503-3549-2. doi: 10.1145/2837614.2837657. URL http://doi.acm.org/10.1145/2837614.2837657.

[26] C. Prakash, J. Lee, Y. Turner, J.-M. Kang, A. Akella, S. Banerjee, C. Clark, Y. Ma, P. Sharma, and Y. Zhang. Pga: Using graphs to express and automatically reconcile network policies. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data*

Communication, SIGCOMM '15, pages 29–42, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3542-3. doi: 10.1145/2785956.2787506. URL http://doi.acm.org/10.1145/2785956.2787506.

[27] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using sdn. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 27–38, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2056-6. doi: 10.1145/2486001.2486022. URL http://doi.acm.org/10.1145/2486001.2486022.

[28] M. Reitblatt, M. Canini, A. Guha, and N. Foster. Fattire: Declarative fault tolerance for software-defined networks. In *Proceedings of the second ACM SIGCOMM workshop on Hot topics in software defined networking*, pages 109–114. ACM, 2013.

[29] S. Saha, S. Prabhu, and P. Madhusudan. Netgen: Synthesizing data-plane configurations for network policies. In *Proceedings of the 1st ACM SIGCOMM Symposium on Software Defined Networking Research*, SOSR '15, pages 17:1–17:6, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3451-8. doi: 10.1145/2774993.2775006. URL http://doi.acm.org/10.1145/2774993.2775006.

[30] A. Singh, J. Ong, A. Agarwal, G. Anderson, A. Armistead, R. Bannon, S. Boving, G. Desai, B. Felderman, P. Germano, A. Kanagala, J. Provost, J. Simmons, E. Tanda, J. Wanderer, U. Hölzle, S. Stuart, and A. Vahdat. Jupiter rising: A decade of clos topologies and centralized control in google's datacenter network. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, SIGCOMM '15, pages 183–197, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3542-3. doi: 10.1145/2785956.2787508. URL http://doi.acm.org/10.1145/2785956.2787508.

[31] S. Smolka, S. Eliopoulos, N. Foster, and A. Guha. A fast compiler for netkat. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming*, ICFP 2015, pages 328–341, New York, NY, USA, 2015. ACM. ISBN 978-1-4503-3669-7. doi: 10.1145/2784731.2784761. URL http://doi.acm.org/10.1145/2784731.2784761.

[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. 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. ISBN 978-1-4503-3279-8. doi: 10.1145/2674005.2674989. URL http://doi.acm.org/10.1145/2674005.2674989.

[33] 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. ISBN 978-1-4503-2596-7. doi: 10.1145/2535771.2535774. URL http://doi.acm.org/10.1145/2535771.2535774.

[34] Y. Yuan, R. Alur, and B. T. Loo. Netegg: Programming network policies by examples. In *Proceedings of the 13th ACM Workshop on Hot Topics in Networks*, HotNets-XIII, pages 20:1–20:7, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-3256-9. doi: 10.1145/2670518.2673879. URL http://doi.acm.org/10.1145/2670518.2673879.

[35] S. Zhang, F. Ivancic, C. Lumezanu, Y. Yuan, A. Gupta, and S. Malik. An adaptable rule placement for software-defined networks. In *Dependable Systems and Networks (DSN), 2014 44th Annual IEEE/IFIP International Conference on*, pages 88–99, June 2014. doi: 10.1109/DSN.2014.24.

[36] W. Zhou, D. Jin, J. Croft, M. Caesar, and P. B. Godfrey. Enforcing customizable consistency properties in software-defined networks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 73–85, Berkeley, CA, USA, 2015. USENIX Association. ISBN 978-1-931971-218. URL http://dl.acm.org/citation.cfm?id=2789770.2789776.