Network-wide Configuration Synthesis

Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev

ETH Zürich

Abstract. Computer networks are hard to manage. Given a set of high-level requirements (e.g., reachability, security), operators have to manually figure out the individual configuration of potentially hundreds of devices running complex distributed protocols so that they, collectively, compute a compatible forwarding state. Not surprisingly, operators often make mistakes which lead to downtimes.

To address this problem, we present a novel synthesis approach that automatically computes correct network configurations that comply with the operator's requirements. We capture the behavior of existing routers along with the distributed protocols they run in stratified Datalog. Our key insight is to reduce the problem of finding correct input configurations to the task of synthesizing inputs for a stratified Datalog program. To solve this synthesis task, we introduce a new algorithm that synthesizes inputs for stratified Datalog programs. This algorithm is applicable beyond the domain of networks.

We leverage our synthesis algorithm to construct the first network-wide configuration synthesis system, called `SyNET`, that support multiple interacting routing protocols (OSPF and BGP) and static routes. We show that our system is practical and can infer correct input configurations, in a reasonable amount time, for networks of realistic size (> 50 routers) that forward packets for multiple traffic classes.

1 Introduction

Despite being mission-critical for most organizations, managing a network is surprisingly hard and brittle.

A key reason is that network operators have to manually come up with a configuration, which ensures that the underlying distributed protocols compute a forwarding state that satisfies the operator's requirements.

Doing so requires operators to precisely understand: *(i)* the behavior of each distributed protocol; *(ii)* how the protocols interact with each other; and *(iii)* how each parameter in the configuration affects the distributed computation.

Because of this complexity, operators often make mistakes that can lead to severe network downtimes. As an illustration, Facebook (and Instagram) recently suffered from widespread issues for about an hour due to a misconfiguration [1]. In fact, studies show that most network downtimes are caused by humans, not equipment failures [2]. Such misconfigurations can have Internet-wide effects [3].

To prevent misconfigurations, researchers have developed tools that check if a given configuration is correct [4, 5, 6, 7]. While useful, these works still require network operators to produce the configurations in the first place. Template-based approaches [8, 9, 10, 11] along with vendor-agnostic abstractions [12, 13, 14] have been proposed to reduce the configuration burden. However, they still require operators to understand precisely the details of each protocol. Recently, Software-Defined Networks (SDNs) have emerged as another paradigm to manage networks by *programming* them from a central controller. Deploying SDN is, however, a major hurdle as it requires new network devices *and* management tools. Further, designing correct, robust and yet, scalable, SDN controllers is challenging [15, 16, 17, 18]. Because of this, only a handful of networks are using SDN in production. As a result, configuring individual devices is by far the most widespread (and default) way to manage networks.

Problem Statement: Network-Wide Configuration Synthesis. Ideally, from a network operator perspective, one would like to solve what we refer to as the *Network-Wide Configuration Synthesis* problem: *Given a network specification $\mathcal{N}$, which defines the behavior of all routing protocols run by the routers, and a set $\mathcal{R}$ of requirements on the network-wide forwarding state, discover a configuration $\mathcal{C}$ such that the routers converge to a forwarding state compatible with $\mathcal{R}$.* That is, the operator simply provides the high-level requirements $\mathcal{R}$, and the configuration $\mathcal{C}$ is obtained automatically.

Distributed vs. Static routing. Relying as much as possible on distributed protocols to compute the forwarding state is critical to ensure network reliability and scalability. A simpler problem would be to statically configure the forwarding entries of each router via static routes (e.g. see [19, 20]). Relying solely on static routes is, however, undesirable for two reasons. First, they prevent routers from reacting locally upon failure. Second, they can be costly to update as routers often have a large number of static entries.

Key Challenges. Coming up with a solution to the network-wide synthesis problem is challenging for at least three reasons: *(i) Diversity*: protocols have different expressiveness in terms of the forwarding entries they compute. For instance, the Open Shortest Path First protocol (OSPF) can only direct traffic along shortest-paths, while the Border Gateway Protocol (BGP) can direct traffic along non-shortest paths. Conversely, BGP cannot forward traffic along multiple paths by default[1], while OSPF supports multi-path routing and is thus better suited for load-balancing traffic, a feature heavily used in practice. *(ii) Dependence:* distinct protocols often depend on one another, making it challenging to ensure that they collectively compute a compatible forwarding state. For instance, BGP depends on the network-wide intra-domain configuration; and *(iii) Feasibility*: the search space of configurations is massive and it is thus difficult to find one that leads to a forwarding state satisfying the requirements.

This Work. In this paper, we provide the first solution to the network-wide synthesis problem. Our approach is based on two steps. First, we use stratified Datalog to capture the behavior of the network, i.e. the distributed protocols

[1] While vendor-specific workarounds to make BGP multipath exist, these break the congruency between the control and data plane and could lead to correctness issues.

ran by the routers together with any protocol dependencies. Datalog is indeed particularly well-suited for describing these protocols in a clear and declarative way. Here, the fixed point of a Datalog program represents the stable forwarding state of the network. Second, and a key insight of our work: we pose the network-wide synthesis problem as an instance of finding an input for a stratified Datalog program where the program's fixed point satisfies a given property. That is, the network operator simply provides the high-level requirements $\mathcal{R}$ on the forwarding state (i.e., which is the same as requiring the Datalog program' fixed point to satisfy $\mathcal{R}$), and our synthesizer automatically finds an input $\mathcal{C}$ to the Datalog program (i.e., which identifies the wanted network-wide configuration). We remark that our Datalog input synthesis algorithm is a general, independent contribution, and is applicable beyond networks.

Main Contributions. To summarize, our main contributions are:

– A formulation of the network-wide synthesis problem in terms of input synthesis for stratified Datalog (Section 2).
– The first input synthesis algorithm for stratified Datalog. This algorithm is of broader interest and is applicable beyond networks (Section 5).
– An instantiation and an end-to-end implementation of our input synthesis algorithm to the network-wide synthesis problem, along with network-specific optimizations, in a system called `SyNET`.
– An evaluation of `SyNET` on networks with multiple interacting widely-used protocols. In addition, we test the correctness of the generated configurations on an emulated network environment. Our results show that `SyNET` can automatically synthesize input configurations for networks of realistic size (> 50 routers) carrying multiple traffic classes (Section 6).

2 Network-wide Configuration Synthesis

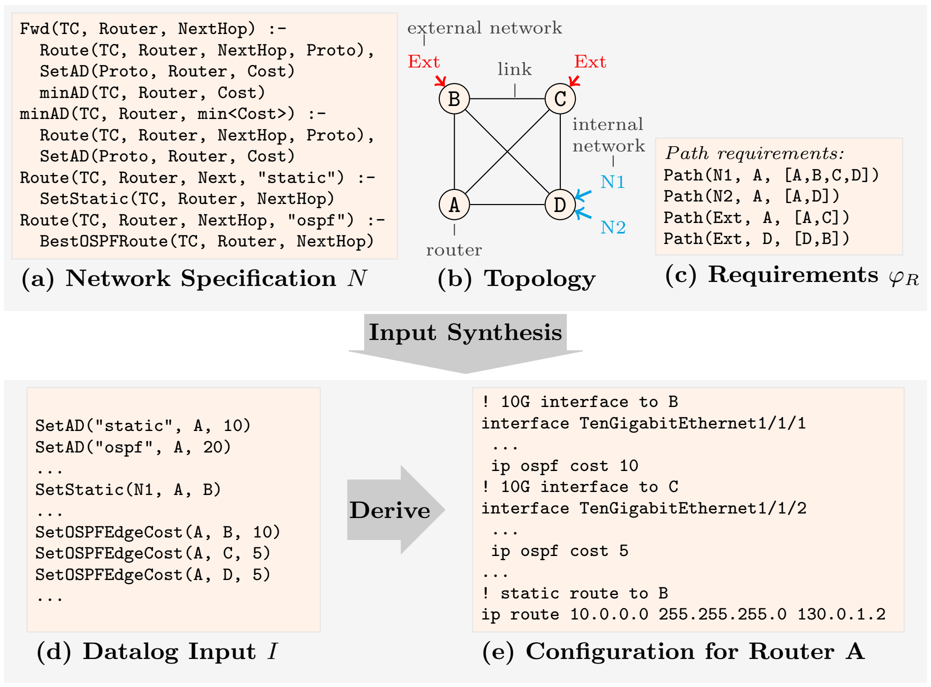
We now illustrate our configuration synthesis approach on a simple example. We highlight how, given a network and a set of requirements, we can pose the synthesis problem as an instance of input synthesis for stratified Datalog.

2.1 Motivating Example

We consider the simple network topology, depicted in Figure 1(b), composed of 4 routers denoted A, B, C and D. Routers B and C can reach the external network `Ext`, and router D is directly connected to two internal networks `N1` and `N2`. In the following, we use the term traffic class to refer to a set of packets (e.g. packets destined to `N1`) that are handled analogously according to the requirements. In practice, each traffic class may contain thousands of IP prefixes [21].

Computation of Forwarding State. We first informally describe how each router's forwarding entries are computed, assuming the configuration is provided.

Each router runs both, OSPF and BGP protocols, and in addition can also be configured with static routes. The computation of OSPF is based on finding least-cost paths to the internal destinations as well as to all routers in the network,



```
Fwd(TC, Router, NextHop) :-
  Route(TC, Router, NextHop, Proto),
  SetAD(Proto, Router, Cost)
  minAD(TC, Router, Cost)
minAD(TC, Router, min<Cost>) :-
  Route(TC, Router, NextHop, Proto),
  SetAD(Proto, Router, Cost)
Route(TC, Router, Next, "static") :-
  SetStatic(TC, Router, NextHop)
Route(TC, Router, NextHop, "ospf") :-
  BestOSPFRoute(TC, Router, NextHop)
```

(a) Network Specification N

(b) Topology

Path requirements:

```
Path(N1, A, [A,B,C,D])
Path(N2, A, [A,D])
Path(Ext, A, [A,C])
Path(Ext, D, [D,B])
```

(c) Requirements φ_R

```
SetAD("static", A, 10)
SetAD("ospf", A, 20)
...
SetStatic(N1, A, B)
...
SetOSPFEdgeCost(A, B, 10)
SetOSPFEdgeCost(A, C, 5)
SetOSPFEdgeCost(A, D, 5)
...
```

(d) Datalog Input I

```
! 10G interface to B
interface TenGigabitEthernet1/1/1
 ...
 ip ospf cost 10
! 10G interface to C
interface TenGigabitEthernet1/1/2
 ...
 ip ospf cost 5
...
! static route to B
ip route 10.0.0.0 255.255.255.0 130.0.1.2
```

(e) Configuration for Router A

Fig. 1: Network-wide Configuration Synthesis. The input is: **(a)** declarative network specification N in stratified Datalog **(b)** network topology, and **(c)** routing requirements φ_R. The output is: **(d)** a Datalog input I that results in a forwarding state satisfying the requirements. Configurations **(e)** are derived from I.

where cost is the sum of the link weights defined in router configurations. The least-cost paths are then used to generate forwarding entries at each router to all internal destinations. In our example, these internal destinations are `N1` and `N2`. In contrast, BGP computes forwarding entries to reach external destinations, `Ext` in our example. The computed forwarding entries define the next hop router for each destination. For example, BGP computes an entry at router A for `Ext` which forwards packets to a border router (i.e., either B or C). To decide which router the entry should forward to, each BGP router selects the egress point (i.e., border router) to reach an externally-learned prefix based on a preference value. This preference is (typically) defined in the configuration of each border router and propagated network-wide. If multiple routers announce the same preference for a prefix, internal BGP routers directs traffic to the closest egress point, according to the OSPF costs.

Once BGP and OSPF have finished computing their forwarding entries, each router takes these entries (along with those defined via static routes) and selects the OSPF-, BGP-, static route- produced forwarding entry with the highest preference (in networking terms, higher preference means lower administrative cost) defined in its local configuration. The union of all forwarding entries obtained at the routers is referred to as the forwarding state of the network.

Configuration Synthesis. Next, we illustrate the opposite direction (and one this work focuses on): given requirements φ_R, find a configuration which the protocols use to compute a forwarding state (as described above) that satisfies φ_R.

Let us consider the four path requirements given in Figure 1(c). The first two state that A must forward packets for the traffic classes `N1` and `N2` along the paths $A \rightarrow B \rightarrow C \rightarrow D$ and $A \rightarrow D$, respectively. Note that these two requirements might reflect a security policy in the network or generated by a traffic engineering optimization tool [22, 23]. These two requirements cannot be enforced using OSPF alone. The reason is that, as discussed, OSPF works by selecting the least-cost path (by summing the weights on the links) and there is no assignment of weights to links which would lead to least-cost paths that exactly match the two path requirements.

Yet, the two requirements can be enforced by: *(i)* generating a static route-based forwarding entry at A to forward packets for `N1` to B; *(ii)* configuring link weights so paths $A \rightarrow D$ and $B \rightarrow C \rightarrow D$ have the lowest OSPF costs from A to D and, respectively, from B to D; and *(iii)* on router A, configure a higher preference for forwarding entries based on static routes than OSPF forwarding entries. Because a static route forwarding entry is only generated for destination `N1` (from *(i)*) and not `N2`, this means the entry for `N1` will forward the traffic to router B while the entry for `N2` will be the OSPF generated one (from *(ii)*).

The last two path requirements state that A and D must forward packets destined to the traffic class `Ext` to C and B, respectively. The two path requirements can by satisfied by: *(i)* setting identical BGP router preferences at the local configurations of B and C; and *(ii)* configuring link weights so that paths $A \rightarrow C$ and $D \rightarrow B$ have the lowest costs from A to C and from D to B, respectively. In this way, BGP will use the results from the OSPF least-cost paths to compute its forwarding entries to `Ext`. This is an example where BGP interacts with OSPF and uses information from its computation.

The following is the final configuration produced by our synthesizer (the synthesizer is discussed in later sections):

- weight 10 is assigned to link $A \rightarrow B$,
- weight 5 is assigned to links $B \rightarrow C$, $C \rightarrow D$, and $A \rightarrow C$,
- weight 4 is assigned to link $D \rightarrow B$,
- weight 100 is assigned to the remaining links,
- a static route- based forwarding entry is defined at router A to forward traffic for $N1$ to B, and
- the router preference for all routers is set to 100.

In Figure 1(e), we illustrate an excerpt of router A's local configuration.

Phrasing the Problem as Inputs Synthesis for Stratified Datalog. A key insight of our work is to pose the question of finding a network configuration as an instance of input synthesis for stratified Datalog.

First, we declaratively specify the behavior of the network, i.e. the distributed protocols that the routers run, the protocol interactions, and the network topology, as a stratified Datalog program N. As requirements usually pertain to the stable forwarding state, the stratified Datalog encoding captures the stable state of these routing protocols as opposed to intermediate computation steps. Few relevant Datalog rules are given in Figure 1(a); we detail this specification step in Section 4. The resulting Datalog program derives a predicate `Fwd` that defines the forwarding entries computed by all routers, where `Fwd(TC, Router, NextHop)` is derived if `Router` forwards packets for traffic class `TC` to router `NextHop`.

Second, we can directly express routing requirements as constraints over the predicate `Fwd`. We denote these constraints with φ_R in Figure 1.

Finally, an input I to the Datalog program N identifies a network-wide configuration. We formalize the network-wide configuration synthesis problem as:

Definition 1. *The network-wide configuration synthesis problem is:*
Input *A declarative network specification N and routing requirements φ_R.*
Output *A Datalog input I such that $[\![N]\!]_I \models \varphi_R$, if such an input exists; otherwise, return unsat.*

In our definition, $[\![N]\!]_I$ denotes the fixed point of the Datalog program N for the input I, and $[\![N]\!]_I \models \varphi_R$ holds if this fixed point satisfies the constraints φ_R.

Synthesizing inputs for stratified Datalog is, however, a difficult (and, in general, undecidable) problem [24]. The problem is, however, decidable if we fix a finite set of values to bound the set of inputs. This is reasonable in the context of networks, where values represent finitely many routers, interfaces, and configuration parameters.

To address the problem, we introduce a new iterative synthesis algorithm that partitions the Datalog program P into strata $P_1, \ldots, P_n$, finds an input I_i for each stratum P_i and then construct an input I for the Datalog program P. Each stratum P_i is a semi-positive Datalog program that enjoys the property that if a predicate is derived by the rules after some number of steps, then it must be contained in the fixed point. We describe this algorithm in Section 5.

3 Background: Stratified Datalog

We briefly overview the syntax and semantics of stratified Datalog.

Syntax. Datalog's syntax is given in Figure 2. We use $\overline{r}$, $\overline{l}$, and $\overline{t}$ to denote zero or more rules, literals, and terms separated by commas, respectively. A Datalog program is *well-formed* if for any rule $a \leftarrow \overline{l}$, we have $vars(a) \subseteq vars(\overline{l})$, where $vars(\overline{l})$ returns the set of variables in $\overline{l}$.

A predicate is called *extensional* if it appears only in the bodies of rules (right side of the rule), otherwise (if it appears at least once in a rule head) it is called *intensional*. We denote the sets of extensional and intensional predicates of a program P by $edb(P)$ and $idb(P)$, respectively.

A Datalog program P is *stratified* if its rules can be partitioned into strata $P_1, \ldots, P_n$ such that if a predicate p occurs in a positive (negative) literal in the

$$\begin{array}{rrcl rrcl rrcl}
\textit{(Program)} & P & ::= & \overline{r} & \textit{(Literal)} & l & ::= & a \mid \neg a & \textit{(Variables)} & X, Y & \in & \textit{Vars} \\
\textit{(Rule)} & r & ::= & a \leftarrow \overline{l} & \textit{(Predicates)} & p, q & \in & \textit{Preds} & \textit{(Values)} & v & \in & \textit{Vals} \\
\textit{(Atom)} & a & ::= & p(\overline{t}) & \textit{(Term)} & t & ::= & X \mid v & & & &
\end{array}$$

Fig. 2: Syntax of stratified Datalog

body of a rule in P_i, then all rules with p in their heads are in a stratum P_j with $j \leq i$ ($j < i$). Stratification ensures that predicates that appear in negative literals are fully defined in lower strata.

We syntactically extend stratified Datalog with aggregate functions such as `min` and `max`. This extension is possible as stratified Datalog is equally expressive to Datalog with stratified aggregate functions; for details see [25]

Semantics. Let $\mathcal{A} = \{p(\overline{t}) \mid \overline{t} \subseteq \textit{Vals}\}$ denote the set of all ground (i.e. variable-free) atoms. The complete lattice $(\mathcal{P}(\mathcal{A}), \subseteq, \cap, \cup, \emptyset, \mathcal{A})$ partially orders the set of interpretations $\mathcal{P}(\mathcal{A})$.

Given a substitution $\sigma \in \textit{Vars} \to \textit{Vals}$ mapping variables to values. Given an atom a, we will write $\sigma(a)$ for the ground atom obtained by replacing the variables in a according to σ; e.g., $\sigma(p(X))$ returns the ground atom $p(\sigma(X))$. The consequence operator $T_P \in \mathcal{P}(\mathcal{A}) \to \mathcal{P}(\mathcal{A})$ for a program P is defined as

$$T_P(A) = A \cup \{\sigma(a) \mid a \leftarrow l_1 \ldots l_n \in P, \forall l_i \in \overline{l}.\ A \vdash \sigma(l_i)\}$$

where $A \vdash \sigma(a)$ if $\sigma(a) \in A$ and $A \vdash \sigma(\neg a)$ if $\sigma(a) \notin A$.

An input for P is a set of ground atoms constructed using P's extensional predicates. Let P be a program with strata $P_1, \ldots, P_n$ and I be an input for P. The model of P for I, denoted by $[\![P]\!]_I$, is M_n, where $M_0 = I$ and $M_i = \bigcap\{A \in \mathsf{fp}\ T_{P_i} \mid M_{i-1} \subseteq A\}$ is the smallest fixed point of T_{P_i} that is greater than or equal to the lower stratum's model M_{i-1}.

4 Declarative Network Specification

In this section, we first describe how we declarative specify the behavior of the network as a Datalog program. Afterwards, we discuss how routing requirements are specified as constraints over the Datalog program's fixed point.

4.1 Specifying Networks

To faithfully capture a network's behavior, we model *(i)* the behavior of routing protocols and their interactions and *(ii)* the topology of the network.

Expressing Protocols in Stratified Datalog. We formalize individual routing protocols and how routers combine the forwarding entries computed by these protocols as a stratified Datalog program N. The Datalog program N derives the predicate `Fwd(TC, Router, NextHop)`, which represents the network's global forwarding state. In Figure 1(a), for example, we show the relevant rules that define how the forwarding entries computed by OSPF are combined with those defined via static routes. The predicate `Route(TC, Router, NextHop, Proto)` captures the forwarding entries of OSPF and static routes. The top Datalog rule states that routers select, for each traffic class `TC`, the forwarding entry with the minimal administrative cost (`minAD`) calculated over all protocols via the second Datalog rule in Figure 1(a). The bottom two rules define the predicate `Route`, which collects the forwarding entries defined via static routes and computed by OSPF. We remark that OSPF routes (represented by the predicate `BestOSPFRoute`) are defined through additional Datalog rules that capture the behavior of the OSPF protocol[2].

Network Topology. The network topology is also captured via Datalog rules in the program N. We model each router as a constant and use predicates to represent the topology. For example, the predicate `SetLink(R1, R2)` represents that two routers $R1$ and $R2$ are connected via a link, and we add the Datalog rule `SetLink(R1, R2)` $\leftarrow$ `true` to define such a link.

4.2 Specifying Requirements

We specify the requirements as function-free first-order constraints over the predicate `Fwd(TC, Router, NextHop)`, which defines the network's forwarding state. We write $A \models \varphi$ to denote that a Datalog interpretation A satisfies φ. For illustration, we describe how common routing requirements can be specified:

`Path(TC, R1, [R1, R2, .., Rn])` (Path requirement): packets for traffic class `TC` must follow the path `R1, .., Rn`. These requirements are specified as a conjunction over the predicate `Fwd`.

$\forall$`R1, R2`. `Fwd(TC1, R1, R2)` $\Rightarrow \neg$`Fwd(TC2, R1, R2)` (Traffic isolation): the paths for two distinct traffic classes `TC1` and `TC2` do not share links in the same direction.

`Reach(TC, R1, R2)` (Reachability): packets for traffic class `TC` can reach router `R2` from router `R1`. The predicate `Reach` is the transitive closure over the predicate `Fwd` (defined via Datalog rules).

$\forall$`TC,R`. ($\neg$`Reach(TC, R, R)`) (Loop-freeness): generic requirement stipulating that the forwarding plane has no loops.

More complex requirements, such as way pointing, can be specified based on the core function-free first-order constraints provided by `SyNET`. Further, `SyNET` can be used as a backend for a high-level requirements language that is easier to use by a network operator.

4.3 Network-wide Configurations

The input protocol configurations deployed at the network's routers are represented as input *edb* predicates to the Datalog programs that formalize the protocols. For example, the local OSPF configuration for a router specifies the weights associated with the links connected to that router; this is represented by the *edb* predicate `SetOSPFEdgeCost(Router, NextHop, Weight)`.

[2] A detailed OSPF model can be found in the technical report [26].

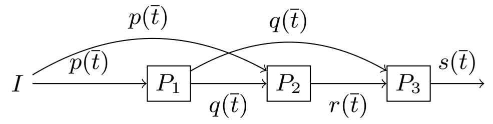

$$\begin{tikzcd} I \arrow[r, "p(\bar{t})"] \arrow[rr, bend left, "p(\bar{t})"] & \boxed{P_1} \arrow[r, "q(\bar{t})"'] \arrow[rr, bend left, "q(\bar{t})"] & \boxed{P_2} \arrow[r, "r(\bar{t})"'] & \boxed{P_3} \arrow[r, "s(\bar{t})"] & {} \end{tikzcd}$$

Fig. 3: A Datalog program with strata P_1, P_2, and P_3, and flow of predicates between the strata.

A subset of the synthesized Datalog input for our motivating example is given in Figure 1(d). Here, `SetAD` defines the administrative cost of static routes to be lower than that of OSPF (so static routes are prefered over forwarding entries computed by OSPF). The predicate `SetStatic(N1, A, B)`, which represents static routes, defines a static route for `N1` from A to B. The predicate `SetOSPFEdgeCost` defines the links' weights.

5 Input Synthesis for Stratified Datalog

We now present a new iterative algorithm for synthesizing inputs for stratified Datalog. We first describe the high-level flow of the algorithm before presenting the details.

High-Level Flow Consider the stratified Datalog program with strata P_1, P_2, and P_3, depicted in Figure 3. Incoming and outgoing edges of a stratum P_i indicate the *edb* predicates and, respectively, the *idb* predicates of that stratum. For example, the stratum P_3 takes as input predicates $q(\bar{t})$ and $r(\bar{t})$ and derives the predicate $s(\bar{t})$. Our iterative algorithm first synthesizes an input I_3 for P_3 which determines the predicates $q(\bar{t})$ and $r(\bar{t})$ that $P_1 \cup P_2$ must output. To synthesize such an input for a single stratum, we present an algorithm, called $\mathcal{S}_{SemiPos}$, that addresses the input synthesis problem for semi-positive Datalog programs [27, Chapter 15.2], i.e. Datalog programs where negation is restricted to *edb* predicates. After synthesizing an input I_3 for P_3, our iterative algorithm synthesizes an input I_2 for P_2 such that the fixed-point $[\![P_2]\!]_{I_2}$ produces the predicates $r(\bar{t})$ that are contained in the already synthesized input I_3 for P_3. We note that this iterative process may require backtracking, in case no input for P_2 can produce the desired predicates $r(\bar{t})$ contained in I_3. The algorithm terminates when it synthesizes inputs for all three strata.

In the following, we first present the algorithm $\mathcal{S}_{SemiPos}$ that is used to synthesize an input for a single stratum (which is a semi-positive program). Then, we present the general algorithm, called $\mathcal{S}_{Strat}$, that iteratively applies $\mathcal{S}_{SemiPos}$ for each stratum to synthesize inputs for stratified Datalog programs.

5.1 Input Synthesis for Semi-positive Datalog with SMT

The key idea is to reduce the input synthesis problem to satisfiability of SMT constraints: Given a semi-positive Datalog program P and a constraint φ, we encode the question $\exists I.\ [\![P]\!]_I \models \varphi$ into an SMT constraint ψ. If ψ is satisfiable, then from a model of ψ we can derive an input I such that $[\![P]\!]_I \models \varphi$.

SMT Encoding Challenges. Given a Datalog program P and a constraint φ, encoding the question $\exists I.\ [\![P]\!]_I \models \varphi$ with SMT constraints is non-trivial due to the mismatch between Datalog's program fixed point semantics and the classical semantics of first-order logic. This means that simply taking the conjunction of all Datalog rules into an SMT solver does not solve our problem. For example, consider the following Datalog program P_{tc}:

$$\begin{aligned} tc(X,Y) &\leftarrow e(X,Y) \\ tc(X,Y) &\leftarrow tc(X,Z), tc(Z,Y) \end{aligned}$$

which computes the transitive closure of the predicate $e(X,Y)$. A naive way of encoding these Datalog rules with SMT constraints:

$$\begin{aligned} \forall X,Y.\ (e(X,Y) &\Rightarrow tc(X,Y)) \\ \forall X,Y.\ ((\exists Z.\ tc(X,Z) \wedge tc(Z,Y)) &\Rightarrow tc(X,Y)) \end{aligned}$$

and we denote the conjunction of these two SMT constraints as $[P_{tc}]$. Now, suppose we have the fixed point constraint $\varphi_{tc} = (\neg e(v_0,v_2)) \wedge tc(v_0,v_2)$ and we want to generate an input I so that $[\![P_{tc}]\!]_I \models \varphi_{tc}$. A model that satisfies $[P_{tc}] \wedge \varphi_{tc}$ is

$$\mathcal{M} = \{e(v_0,v_1), tc(v_0,v_1), tc(v_0,v_2)\}$$

The input derived from this model, obtained by projecting $\mathcal{M}$ over the *edb* predicate e, is $I_{\mathcal{M}} = \{e(v_0,v_1)\}$. We get

$$[\![P_{tc}]\!]_{I_{\mathcal{M}}} = \{e(v_0,v_1), tc(v_0,v_1)\}$$

and so $[\![P_{tc}]\!]_{I_{\mathcal{M}}} \not\models \varphi_{tc}$, which is clearly not what is intended.

SMT Encoding. Our key insight is to split the constraint φ into a conjunction of positive and negative clauses, where a clause φ is positive (resp., negative) if $A \models \varphi$ implies that $A' \models \varphi$ for any interpretation $A' \supseteq A$ (resp., $A' \subseteq A$). We can then unroll recursive predicates to obtain a sound encoding for positive constraints, and we do not unroll them to get a sound encoding for negative constraints.

The encoding of a Datalog program P into an SMT constraint is defined in Figure 4. The resulting SMT constraint is denoted by $[P]_k$, where the parameter k defines the number of unroll steps. In the encoding we assume that *(i)* all terms in rules' heads are variables and *(ii)* rules' heads with the same predicate have identical variable names. Note that any Datalog program can be converted into this form using rectification [28] and variable renaming.

Function Encode. The constraint returned by $\textsc{Encode}(p,P)$ states that an atom $p(X)$ is derived if P has a rule that derives $p(\overline{X})$ and whose body evaluates to true. To capture Datalog's semantics, the variables in $p(\overline{X})$ are universally quantified, while those in the rules' bodies are existentially quantified. This constraint $\textsc{Encode}(p,P)$ is sound for negative requirements, but not for positive ones as it does not state that $p(\overline{X})$ is derived *only if* a rule body with $p(\overline{X})$ in the head evaluates to true.

$$
\begin{array}{lll}
[P]_k & = & \bigwedge_{p \in idb(P)} \textsc{Encode}(P, p) \wedge \textsc{Unroll}(P, p, k) \\
\textsc{Encode}(P, p) & = & \bigwedge_{p(\overline{X}) \leftarrow \bar{l} \in P} \forall \overline{X}.\ \big((\exists \overline{Y}.\ \bigwedge \bar{l}) \Rightarrow p(\overline{X})\big), \text{where } \overline{Y} = vars(\bar{l}) \setminus \overline{X} \\
\textsc{Unroll}(P, p, k) & = & \bigwedge_{0 \leq i \leq k} \textsc{Step}(P, p, i) \\
\textsc{Step}(P, p, i) & = & \forall \overline{X}.\ \big(p_i(\overline{X}) \Leftrightarrow (\bigvee_{p(\overline{X}) \leftarrow \bar{l} \in P} \exists \overline{Y}.\ \tau(\bar{l}, i-1))\big), \text{where } \overline{Y} = vars(\bar{l}) \setminus \overline{X} \\
\tau(\bar{l}, k) & = & \begin{cases} \tau(l_1, k) \wedge \cdots \tau(l_n, k) & \text{if } \bar{l} = l_1 \wedge \cdots \wedge l_n \\ \neg\tau(p(\bar{t}), k) & \text{if } \bar{l} = \neg p(\bar{t}) \\ \mathsf{false} & \text{if } \bar{l} = p(\bar{t}), p \in idb(P), k = 0 \\ p_k(\bar{t}) & \text{if } \bar{l} = p(\bar{t}), p \in idb(P), k > 0 \\ p(\bar{t}) & \text{if } \bar{l} = p(\bar{t}), p \in edb(p) \end{cases}
\end{array}
$$

Fig. 4: Encoding a Datalog program P with constraints $[P]_k$

Functions Unroll and Step. The constraint returned by $\textsc{Step}(P, p, i)$ encodes whether an atom $p(X)$ is derived after i applications of P's rules; e.g., $p(X)$'s truth value after 3 steps is represented with the atom $p_3(X)$. Intuitively, $p(X)$ is true iff there is a rule that derives $p(X)$ and whose body evaluates to true using the atoms derived in previous iterations. Which atoms are derived in previous iterations is captured by the literal renaming function τ. Note that $\tau(l, 0)$ returns false for any *idb* literal l since all intensional predicates are initially false. Further, $\tau(l, k)$ returns l for any extensional literal l (the last case in Figure 4) since their truth values do not change. Finally, the constraint returned by $\textsc{Unroll}(P, p, k)$ conjoins $\textsc{Step}(P, p, 0)$, ..., $\textsc{Step}(P, p, k)$ to capture the derivation of $p(X)$ after k steps. This is sound for positive requirements, but not for negative ones since more $p(X)$ atoms may be derived after k steps.

Example. To illustrate the encoding, we translate the Datalog program:

$$
\begin{array}{l}
tc(X, Y) \leftarrow e(X, Y) \\
tc(X, Y) \leftarrow tc(X, Z), tc(Z, Y)
\end{array}
$$

which computes the transitive closure of the predicate $e(X, Y)$. This program has one *idb* predicate, *tc*. The function $\textsc{Encode}(P, tc)$ returns

$$
\begin{array}{r}
(\forall X, Y.\ e(X, Y) \Rightarrow tc(X, Y)) \\
\wedge(\forall X, Y.\ (\exists Z.\ tc(X, Z) \wedge tc(Z, Y)) \Rightarrow tc(X, Y))
\end{array}
$$

We apply function $\textsc{Unroll}(P, tc, 2)$ for $k = 2$, which after simplifications returns

$$
\begin{array}{l}
\forall X, Y.\ (tc_1(X, Y) \Leftrightarrow e(X, Y)) \\
\forall X, Y.\ (tc_2(X, Y) \Leftrightarrow e(X, Y) \vee (\exists Z.\ tc_1(X, Z) \wedge tc_1(Z, Y))
\end{array}
$$

In the constraints, the predicates tc_1 and tc_2 encode the derived predicates tc after 1 and, respectively, 2, derivation steps.

Algorithm. Algorithm $\mathcal{S}_{SemiPos}(P, \varphi)$, given in Algorithm 1, first calls function $\textsc{Simplify}(\varphi)$ that *(i)* instantiates any quantifiers in φ and *(ii)* transforms the result into a conjunction of clauses, where each clause is a disjunction of literals.

Algorithm 1: Algorithm $\mathcal{S}_{SemiPos}$ for semi-positive Datalog

```
    Input: Semi-positive Datalog program $P$ and a constraint $\varphi$
    Output: An input $I$ such that $[\![P]\!]_I \models \varphi$ or $\bot$
1   begin
2       $\varphi' \leftarrow$ Simplify($\varphi$)
3       for $k \in [1..bound_k]$ do
4           $\varphi_k \leftarrow$ Rewrite($\varphi', k$)
5           $\psi \leftarrow [P]_k \wedge \varphi_k$
6           if $\exists J.\ J \models \psi$ then
7               $I \leftarrow \{p(\bar{t}) \in J \mid p \in edb(P)\}$
8               return $I$
9       return $\bot$
```

Then, the algorithm iteratively unrolls the Datalog rules, up to a pre-defined bound, called $bound_k$. In each step of the for-loop, the algorithm generates an SMT constraint that captures *(i)* which atoms are derived after k applications of P's rules and *(ii)* which atoms are never derived by P. The resulting SMT constraint is denoted by $[P]_k$. The algorithm also rewrites the simplified constraint φ' using the function Rewrite(φ', k) which recursively traverses conjunctions and disjunctions in the simplified constraint φ' and maps positive literals to the k-unrolled predicate $p_k(\bar{t})$ and negative literals to $\neg p(\bar{t})$:

$$\textsc{Rewrite}(\varphi, k) = \begin{cases} p_k(\bar{t}) & \text{if } \varphi = p(\bar{t}) \\ \neg p(\bar{t}) & \text{if } \varphi = \neg p(\bar{t}) \\ \textsc{Rewrite}(\varphi_1, k) \vee \cdots \vee \textsc{Rewrite}(\varphi_n, k) & \text{if } \varphi = \varphi_1 \vee .. \vee \varphi_n \\ \textsc{Rewrite}(\varphi_1, k) \wedge \cdots \wedge \textsc{Rewrite}(\varphi_n, k) & \text{if } \varphi = \varphi_1 \wedge .. \wedge \varphi_n \end{cases}$$

Note that since $\vee$ and $\wedge$ are monotone, negative literals constitute negative constraints and positive literals constitute positive constraints.

If the resulting constraint $[P]_k \wedge \psi_k$ is satisfiable, then an input is derived by projecting the interpretation I that satisfies the constraint over all *edb* predicates. Note that if there is an input I such that $[\![P]\!]_I \models \varphi$ and for which the fixed point $[\![P]\!]_I$ is reached in less than $bound_k$ steps, then $\mathcal{S}_{SemiPos}(P, \varphi)$ is guaranteed to return an input.

Theorem 1. *Let P be a semi-positive Datalog program, φ a constraint. If $\mathcal{S}_{SemiPos}(P, \varphi) = I$ then $[\![P]\!]_I \models \varphi$.* [3]

5.2 Iterative Input Synthesis for Stratified Datalog

Our iterative input synthesis algorithm for stratified Datalog, called $\mathcal{S}_{Strat}$, is given in Algorithm 2. We assume that the fixed point constraint φ is defined over predicates that appear in the highest stratum P_n; this is without any loss of generality, as any constraint can be expressed using Datalog rules in the highest

[3] The theorem's proof can be found in the technical report [26].

Algorithm 2: Input synthesis algorithm $\mathcal{S}_{Strat}$ for stratified Datalog

```
Input: Stratified Datalog program $P = P_1 \cup \cdots \cup P_n$, constraint $\varphi$ over $P_n$
Output: An input $I$ such that $[\![P]\!]_I \models \varphi$ or $\bot$
begin
    $\mathcal{F}_1 \leftarrow \emptyset, \ldots, \mathcal{F}_n \leftarrow \emptyset$; $I_1 \leftarrow \bot, \ldots, I_n \leftarrow \bot$; $i \leftarrow n$
    while $i > 0$ do
        if $|\mathcal{F}_i| > bound_{\mathcal{F}}$ then
            $\mathcal{F}_i \leftarrow \emptyset$; $\mathcal{F}_{i+1} \leftarrow \mathcal{F}_{i+1} \cup \{I_{i+1}\}$
            $i \leftarrow i + 1$;   // backtrack to higher stratum
            continue
        $\psi_{\mathcal{F}} \leftarrow \bigwedge_{I' \in \mathcal{F}_i} \big(\neg \bigwedge_{p \in edb(P_i)} \textsc{EncodePred}(I', p)\big)$
        if $i = n$ then
            $\psi_i \leftarrow \varphi$
        else
            $\psi_i \leftarrow \bigwedge_{p \in \Delta_i} \textsc{EncodePred}(I_{i+1} \cup \cdots \cup I_n, p)$
            where $\Delta_i = (edb(P_i) \cup idb(P_i)) \cap (edb(P_{i+1}) \cup \cdots \cup edb(P_n))$
        $I_i = \mathcal{S}_{SemiPos}(P_i, \psi_i \wedge \psi_{\mathcal{F}})$
        if $I_i \neq \bot$ then
            $i \leftarrow i - 1$
        else
            if $i < n$ then
                $\mathcal{F}_i \leftarrow \emptyset$; $\mathcal{F}_{i+1} \leftarrow \mathcal{F}_{i+1} \cup \{I_{i+1}\}$
                $i \leftarrow i + 1$   // backtrack to higher stratum
            else
                return $\bot$
    return $I = \{p(\bar{t}) \in I_1 \cup \cdots \cup I_n \mid p \in edb(P)\}$
```

stratum, using a standard reduction to query satisfiability; cf. [24]. Starting with the highest stratum P_n, $\mathcal{S}_{Strat}$ generates an input I_n for P_n such that $[\![P_n]\!]_{I_n} \models \varphi$. Then, it iteratively synthesizes an input for the lower strata $P_{n-1}, \ldots, P_1$ using the algorithm $\mathcal{S}_{SemiPos}$. Finally, to construct an input for P, the algorithm combines the inputs synthesized for all strata and returns this.

Recall that the fixed point of a stratum P_i is given as input to the higher strata $P_{i+1}, \ldots, P_n$. A key step when synthesizing an input I_i for P_i is thus to ensure that the *idb* predicates derived by P_i are identical to the *edb* predicates synthesized for the inputs $I_{i+1}, \ldots, I_n$ of the higher strata. Formally, let

$$\Delta_i = (edb(P_i) \cup idb(P_i)) \cap (edb(P_{i+1}) \cup \cdots \cup edb(P_n))$$

We must ensure that $\{p(\bar{t}) \in [\![P_i]\!]_{I_i} \mid p \in \Delta_i\} = \{p(\bar{t}) \in I_{i+1} \cup \cdots \cup I_n \mid p \in \Delta_i\}$.

Key Steps. The algorithm first partitions P into strata $P_1, \ldots P_n$. The strata can be computed using the predicates' dependency graph; see [27, Chapter 15.2]. For each stratum P_i, it maintains a set of inputs $\mathcal{F}_i$, which contains inputs for P_i for which the algorithm failed to synthesize inputs for the lower strata $P_1, \ldots, P_{i-1}$. We call the sets $\mathcal{F}_i$ *failed* inputs. All $\mathcal{F}_i$ are initially empty.

In each iteration of the while loop, the algorithm attempts to generate an input I_i for stratum P_i. At line 4, the algorithm checks whether $\mathcal{F}_i$ has exceeded a pre-defined bound $bound_{\mathcal{F}}$. If the bound is exceeded, it adds I_{i+1} to the failed inputs $\mathcal{F}_{i+1}$, re-initializes $\mathcal{F}_i$ to the empty set, and backtracks to a higher stratum by incrementing i. This avoids exhaustively searching through all inputs to find an input compatible with those synthesized for the higher strata.

At line 8, the algorithm uses the helper function $\text{EncodePred}(I', p)$. This function returns the constraint $\forall \overline{X}.\ \left(\bigvee_{p(\overline{t}) \in I'} \overline{X} = \overline{t}\right) \Leftrightarrow p(\overline{X})$, which is satisfied by an interpretation I iff I contains identical $p(\overline{t})$ predicates as those in I'. That is, if $I \models \text{EncodePred}(I', p)$ then for any $p(\overline{t})$ we have $p(\overline{t}) \in I$ iff $p(\overline{t}) \in I'$. Therefore, the constraint $\psi_{\mathcal{F}}$ constructed at line 8 is satisfied by an input I_i iff $I_i \notin \mathcal{F}_i$, which avoids synthesizing inputs from the set of failed inputs.

The constraint ψ_i in the algorithm constrains the fixed point of P_i. For the highest stratum P_n, ψ_i is set to the constraint φ given as input to the algorithm. For the remaining strata P_i, ψ_i is satisfied iff the fixed point of P_i is compatible with the synthesized inputs for the higher strata $P_{i+1}, \ldots, P_n$. In addition to constraining P_i's *idb* predicates, we also constraint the input *edb* predicates. This is necessary to eagerly constrain the inputs.

At line 14, the algorithm invokes $\mathcal{S}_{SemiPos}$ to generate an input I_i such that $[\![P_i]\!]_{I_i} \models \varphi_i \wedge \psi_{\mathcal{F}}$. The algorithm proceeds to the lower stratum if such an input is found ($I \neq \bot$); otherwise, if $i < n$ the algorithm backtracks to the higher stratum by increasing i and updating the sets $\mathcal{F}_{i+1}$, and if $i = n$ if returns $\bot$.

Finally, the while-loop terminates when the inputs of all strata have been generated. The algorithm constructs and returns the input I for P.

Theorem 2. *Let P be a stratified Datalog program with strata $P_1, \ldots, P_n$, and φ a constraint over predicates in P_n. If $\mathcal{S}_{Strat}(P, \varphi) = I$ then $[\![P]\!]_I \models \varphi$.* [4]

6 Implementation and Evaluation

In this section we first describe `SyNET`, and end-to-end implementation of our input synthesis algorithm applied to the network-wide synthesis problem. We then turn to our evaluation of `SyNET` on practical topologies and requirements.

6.1 Implementation

`SyNET` is implemented in Python and automatically encodes stratified Datalog programs specified in the LogicBlox language [29] into SMT constraints specified in the SMT-LIB v2 format [30]. It uses the Python API of Z3 [31] to check whether the generated SMT constraints are satisfiable and to obtain a model.

`SyNET` supports routers that run both, OSPF and BGP protocols, and that can be configured with static routes. `SyNET` uses natural splitting for protocols:

[4] The theorem's proof can be found in the technical report [26].

external routes are handled by BGP, while internal routes are handled by IGP protocols (OSPF and static, where static routes are preferred over OSPF). We have partitioned the Datalog rules that capture these protocols and their dependencies into 8 strata. SyNET relies on additional SMT constraints to ensure the well-formedness of the OSPF, BGP, and static route configurations output by our synthesizer. For most topologies and requirements, the Datalog program reaches a fixed point within 20 iterations, and so we fixed the unroll and backtracking bounds ($bound_k$ and $bound_{\mathcal{F}}$) to 20.

SyNET is vendor agnostic with respect to the synthesized configurations. A simple script can be used to convert the output of SyNET into any vendor specific configuration format and then deploy them in production routers. Indeed, to test the correctness of SyNET, we implemented a small script to convert the input synthesized by SyNET to Cisco router configurations.

SyNET supports two key optimizations that improve its performance. The first optimization is *partial evaluation*: SyNET partially-evaluates Datalog rules with predicates whose truth values are known apriori. For example, all SetLink predicates are known and can be eliminated. This reduces the number of variables in the rules and, in turn, in the generated SMT constraints. The second optimization is *network-specific constraints*: we have configured SyNET with generic constraints, which are true for all forwarding states, and with protocol-specific constraints, i.e. constraints that hold for any input to a particular protocol. An example constraint is: *"No packet is forwarded out of the router if the destination network is directly connected to the router"*. These constraints are not specific to particular requirements or topology. They are thus defined one time and can be used to synthesize configurations for any requirements and networks.

6.2 Experiments

To investigate SyNET's performance and scalability, we experimented with different: *(i)* topologies, *(ii)* requirements; and *(iii)* protocol combinations. Further to test correctness, we ran all synthesized configurations on an emulated environment of Cisco routers [32] and we verified that the forwarding paths computed match the requirements for each experiment.

Network Topologies. We used network topologies that have between 4 and 64 routers. The 4-router network is our overview example where we considered the same requirements as those described in Section 2. The 9-router network is Internet2 (see Figure 5), a US-based network that connects several major universities and research institutes. The remaining networks are $n \times n$ grids.

Routing Requirements. For each router and each traffic class, we generate a routing requirement that defines where the packets for that traffic class must be forwarded to. We consider 1, 5, and 10 traffic classes. For a topology with n routers and m traffic classes, we thus generate $n \times m$ requirements.

For topologies with multiple traffic classes, we add one external network announced by two randomly selected routers. We add requirements to enforce that all packets destined to the external networks are forwarded to one of the

		1 Traffic Class		5 Traffic Classes		10 Traffic Classes	
Protocol	# Routers	Avg	Std	Avg	Std	Avg	Std
Static	9	1.3s	(0.5)	2.0s	(0.1)	2.8s	(0.4)
	9 (Internet2)	1.3s	(0.5)	2.0s	(0.0)	4.0s	(0.8)
	16	5.9s	(0.3)	7.8s	(0.4)	11.2s	(0.4)
	25	32.0s	(0.6)	37.0s	(0.6)	46.1s	(0.9)
	36	2m49.7s	(3.0)	3m1.5s	(4.5)	3m27.0s	(4.4)
	49	12m29.2s	(7.0)	13m02.3s	(10.6)	14m10.7s	(15.0)
	64	46m36.2s	(49.0)	47m23.8s	(27.2)	49m22.2s	(39.3)
OSPF+Static	9	9.4s	(0.5)	19.8s	(0.4)	39.9s	(0.5)
	9 (Internet2)	9.0s	(1.4)	21.3s	(1.2)	49.3s	(0.5)
	16	43.5s	(0.7)	1m19.8s	(0.6)	4m5.8s	(1.6)
	25	2m55.2s	(6.1)	7m3.8s	(9.9)	15m56.4s	(38.1)
	36	10m00.5s	(9.5)	23m58.9s	(22.5)	1h11m38.2s	(127.5)
	49	24m11.6s	(43.5)	1h30m00.3s	(89.6)	5h22m55.8s	(421.2)
	64	2h22m13.2s	(209.9)	5h42m58.9s	(619.4)	21h13m16.0s	(1986.7)
BGP+OSPF+Static	9	15.3s	(0.5)	27.7s	(0.5)	1m0.5s	(2.6)
	9 (Internet2)	13.3s	(0.9)	22.7s	(0.9)	1m19.7s	(0.5)
	16	56.0s	(1.6)	2m24.7s	(0.9)	8m29.0s	(10.7)
	25	3m56.3s	(3.1)	8m46.3s	(5.3)	40m09.3s	(99.2)
	36	14m14.0s	(15.0)	43m38.0s	(5.7)	2h35m11.7s	(197.7)
	49	1h23m20.7s	(211.1)	2h15m18.0s	(12.8)	timeout (> 24h)	
	64	1h46m35.0s	(165.8)	7h24m51.3s	(519.2)	timeout (> 24h)	

Table 1: `SyNET`'s synthesis times (averaged over 10 runs) for different number of routers, protocol combinations, and traffic classes in the requirements.

two routers. This models a scenario where the operator is planning maintenance downtime for one of the two routers. Further, to show that `SyNET` synthesizes configurations with partially defined input and protocol dependencies, we assume the local BGP preferences are fixed by the network operator and thus `SyNET` has to synthesize correct OSPF costs to meet the BGP requirements.

Protocols. We consider three different combinations of protocols: *(i)* static routes; *(ii)* OSPF and static routes; and *(iii)* OSPF, BGP, and static routes. The protocol combinations *(i)* and *(ii)* ignore requirements for external networks since only BGP computes routes for them.

Experimental Setup. We run `SyNET` on a machine with 128GB of RAM and a modern 12-core dual-processors running at 2.3GHz.

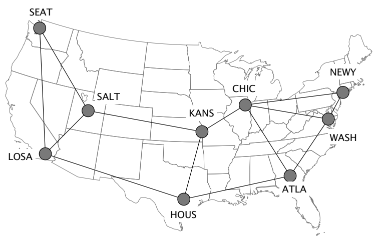


Fig. 5: Internet2 topology

Results. The synthesis times for the different networks and protocol combinations are shown in Table 1 (averaged over 10 runs). `SyNET` synthesizes the overview example's configuration described in Section 2 in 10 seconds. For the largest network (64 routers) and number of traffic classes (10 classes), `SyNET` synthesizes a configuration for static routes (protocol combination *(i)*) in less than 1h, and for the combination of static routes and OSPF, `SyNET` takes less than 22h. When using both OSPF and BGP protocols along with static routes, for all network topologies `SyNET` synthesizes configurations for 1 and 5 traffic classes within 8h; for 10 traffic classes, `SyNET` times out after 24h for the largest topologies with 49 and 64 routers.

Interpretation. Our results show that `SyNET` scales to real-world networks. Indeed, a longitudinal analysis of more than 260 production networks [33] revealed that 56% of them have less than 32 routers. `SyNET` would synthesize configurations for such networks within one hour. `SyNET` also already supports a reasonable amount of traffic classes. According to a study on real-world enterprise and WAN networks [21], even large networks with 100,000s of IP prefixes in their forwarding tables usually see less than 15 traffic classes in total.

While `SyNET` can take more than 24 hours to synthesize a configuration for the largest networks (with all protocols activated and 10 traffic classes), we believe that this time can be reduced through divide-and-conquer. Real networks tend to be hierarchically organized around few regions (to ensure the scalability of the protocols [34]) whose configurations can be synthesized independently. We plan to explore the synthesis of such hierarchical configurations in future work.

7 Related Work

Analysis of Datalog Programs. Datalog has been successfully used to declaratively specify variety of static analyzers [35, 36]. It has been also used to verify network-wide configurations for protocols such as OSPF and BGP [4]. Recent work [37] has extended Datalog to operate with richer classes of lattice structures. Further, the μZ tool [38] extends the Z3 SMT solver with support for fixed points. The focus of all these works is on computing the fixed point of a program P for a given input I and then checking a property φ on the fixed point. That is, they check whether $[\![P]\!]_I \models \varphi$. All of these works assume that the input is provided a priori. In contrast, our procedure discovers an input that produces a fixed point satisfying a given (user-provided) property on the fixed point.

The algorithm presented in [36] can be used to check whether certain tuples are not derived for a given set of inputs. Given a Datalog program P (without negation in the literals), a set Q of tuples, and a set $\mathcal{I}$ of inputs, the algorithm computes the set $Q \setminus \bigcap\{[\![P]\!]_I \mid I \in \mathcal{I}\}$. This algorithm cannot address our problem because it does not support stratified Datalog programs, which are not monotone. While their encoding can be used to synthesize inputs for each stratum of a stratified Datalog program, it supports only negative properties, which require that certain tuples are not derived. Our approach is thus more general than [36] and can be used in their application domain.

The FORMULA system [39, 40] can synthesize inputs for non-recursive Dataog programs, as it supports non-recursive Horn clauses with stratified negation (even though [41] which uses FORMULA shows examples of recursive Horn clauses w/o negation). Handling recursion with stratified negation is nontrivial as bounded unrolling is unsound if applied to all strata together. Note that virtually all network specifications require recursive rules, which our system supports.

Symbolic Analysis and Synthesis. Our algorithm is similar in spirit to symbolic (or concolic) execution, which is used to automatically generate inputs for programs that violate a given assertion (e.g. division by zero); see [42, 43, 44] for an overview. These approaches unroll loops up to a bound and find inputs by calling an SMT solver on the symbolic path. While we also find inputs for a symbolic formula, the entire setting, techniques and algorithms, are all different from the standard symbolic execution setting.

Counter-example guided synthesis approaches are also related [45]. Typically, the goal of synthesis is to discover a program, while in our case the program is given and we synthesize an input for it. There is a connection, however, as a program can be represented as a vector of bits. Most such approaches have a single counter-example generator (i.e., the oracle), while we use a sequence of oracles. It would be interesting to investigate domains where such layered oracle counter-example generation can benefit and improve the efficiency of synthesis.

Network configuration synthesis. Propane [46] and Genesis [19] also produce network-wide configurations out of routing requirements. Unlike our approach, however, Propane only supports BGP and Genesis only supports static routes. In contrast to our system, Propane and Genesis support failure-resilience requirements. While we could directly capture such requirements by quantifying over links, this would make synthesis more expensive. A more efficient way to handle such requirements would be to synthesize a failure-resilient forwarding plane using a system like Genesis [19], and to then feed this as input to our synthesizer to get a network-wide configuration. In contrast to these approaches, our system is more general: one can directly extended it with additional routing protocols, by specifying them in stratified Datalog, and synthesize configurations for *any combination* of routing protocols.

ConfigAssure [47] is a general system that takes as input requirements in first-order constraints and outputs a configuration conforming to the requirements. The fixed point computation performed by routing protocols cannot be captured using the formalism used in ConfigAssure. Therefore, ConfigAssure cannot be used to specify networks and, in turn, to synthesize protocol configurations for networks.

8 Conclusion

We formulated the network-wide configuration synthesis problem as a problem of finding inputs of a Datalog program, and presented a new input synthesis algorithm to solve this challenge. Our algorithm is based on decomposing the Datalog rules into strata and iteratively synthesizing inputs for the individual strata using off-the-shelf SMT solvers. We implemented our approach in a system called `SyNET` and showed that it scales to realistic network size using any combination of OSPF, BGP and static routes.

Bibliography

[1] Jenni Ryall. Facebook, Tinder, Instagram suffer widespread issues. `http://mashable.com/2015/01/27/facebook-tinder-instagram-issues/`.

[2] Juniper Networks. Whats Behind Network Downtime? Proactive Steps to Reduce Human Error and Improve Availability of Networks. Technical report, May 2008.

[3] BGPmon. Internet prefixes monitoring. `http://www.bgpmon.net/blog/`.

[4] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A General Approach to Network Configuration Analysis. In *NSDI'15*.

[5] Nick Feamster and Hari Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *NSDI'05*.

[6] Timothy Nelson, Christopher Barratt, Daniel J. Dougherty, Kathi Fisler, and Shriram Krishnamurthi. The Margrave Tool for Firewall Analysis. In *LISA'10*.

[7] Lihua Yuan, Hao Chen, Jianning Mai, Chen-Nee Chuah, Zhendong Su, and P. Mohapatra. FIREMAN: A Toolkit for FIREwall Modeling and ANalysis. In *S&P'06*.

[8] Laurent Vanbever, Bruno Quoitin, and Olivier Bonaventure. A hierarchical model for BGP routing policies. In *ACM SIGCOMM PRESTO'09*.

[9] X. Chen, M. Mao, and J. Van der Merwe. Pacman: a platform for automated and controlled network operations and configuration management. In *CoNEXT*, 2009.

[10] William Enck, Thomas Moyer, Patrick McDaniel, Subhabrata Sen, Panagiotis Sebos, Sylke Spoerel, Albert Greenberg, Yu-Wei Eric Sung, Sanjay Rao, and William Aiello. Configuration management at massive scale: system design and experience. *IEEE Journal on Selected Areas in Communications*, 2009.

[11] J. Gottlieb, A. Greenberg, J. Rexford, and J. Wang. Automated Provisioning of BGP Customers. *IEEE Network*, 2003.

[12] C. Alaettinoglu, C. Villamizar, E. Gerich, D. Kessens, D. Meyer, T. Bates, D. Karrenberg, and M. Terpstra. Routing Policy Specification Language. RFC 2622.

[13] M. Bjorklund. YANG - A Data Modeling Language for the Network Configuration Protocol (NETCONF). RFC 6020.

[14] R. Enns et al. Network Configuration Protocol (NETCONF). RFC 4741.

[15] Ahmed El-Hassany, Jeremie Miserez, Pavol Bielik, Laurent Vanbever, and Martin Vechev. SDNRacer: Concurrency Analysis for SDNs. In *PLDI'16*.

[16] Marco Canini, Daniele Venzano, Peter Peresini, Dejan Kostic, Jennifer Rexford, and others. A NICE Way to Test OpenFlow Applications. In *NSDI'12*.

[17] Colin Scott, Andreas Wundsam, Barath Raghavan, Aurojit Panda, Andrew Or, Jefferson Lai, Eugene Huang, Zhi Liu, Ahmed El-Hassany, Sam Whitlock, H.B. Acharya, Kyriakos Zarifis, and Scott Shenker. Troubleshooting Blackbox SDN Control Software with Minimal Causal Sequences. In *ACM SIGCOMM*, 2014.

[18] Thomas Ball, Nikolaj Bjørner, Aaron Gember, Shachar Itzhaky, Aleksandr Karbyshev, Mooly Sagiv, Michael Schapira, and Asaf Valadarsky. VeriCon: Towards Verifying Controller Programs in Software-defined Networks. In *PLDI*, 2014.

[19] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Genesis: Synthesizing Forwarding Tables in Multi-tenant Networks. In *POPL'17*.

[20] Nanxi Kang, Zhenming Liu, Jennifer Rexford, and David Walker. Optimizing the "One Big Switch" Abstraction in Software-defined Networks. In *CoNEXT'13*.

[21] Theophilus Benson, Aditya Akella, and David A. Maltz. Mining Policies from Enterprise Network Configuration. In *IMC'09*.

[22] D. Awduche et al. Overview and Principles of Internet Traffic Engineering. RFC3272.

[23] Bernard Fortz, Jennifer Rexford, and Mikkel Thorup. Traffic engineering with traditional ip routing protocols. *IEEE communications Magazine*, 2002.

[24] Alon Y. Halevy, Inderpal Singh Mumick, Yehoshua Sagiv, and Oded Shmueli. Static analysis in datalog extensions. *J. ACM*, 2001.

[25] Inderpal Singh Mumick and Oded Shmueli. How expressive is stratified aggregation? *Annals of Mathematics and Artificial Intelligence*, 1995.

[26] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin T. Vechev. Network-wide Configuration Synthesis. *CoRR*, abs/1611.02537, 2016. URL `http://arxiv.org/abs/1611.02537`.

[27] Serge Abiteboul, Richard Hull, and Victor Vianu, editors. *Foundations of Databases: The Logical Level.* 1995.

[28] Jeffrey D. Ullman. *Principles of Database and Knowledge-Base Systems.* Computer Science Press, 1989.

[29] `https://logicblox.com/content/docs4/corereference/html/index.html`,.

[30] C. Barrett et al. The SMT-LIB Standard: Version 2.0, 2010.

[31] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *TACAS'08*.

[32] Graphical Network Simulator-3 (GNS3). `https://www.gns3.com/`.

[33] Simon Knight, Hung X. Nguyen, Nick Falkner, Rhys Alistair Bowden, and Matthew Roughan. The Internet Topology Zoo. *IEEE Journal on Selected Areas in Communications*, 2011.

[34] Jeff Doyle and Jennifer Carroll. *Routing TCP/IP, Volume 1.* Cisco Press, 2005.

[35] Yannis Smaragdakis and Martin Bravenboer. Using datalog for fast and easy program analysis. In *Datalog Reloaded*, 2010.

[36] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. On abstraction refinement for program analyses in datalog. In *PLDI*, 2014.

[37] Magnus Madsen, Ming-Ho Yee, and Ondřej Lhoták. From datalog to flix: A declarative language for fixed points on lattices. In *PLDI*, 2016.

[38] Kryštof Hoder, Nikolaj Bjørner, and Leonardo De Moura. μZ: An Efficient Engine for Fixed Points with Constraints. In *CAV'11*.

[39] Ethan K. Jackson and Janos Sztipanovits. Towards a Formal Foundation for Domain Specific Modeling Languages. In *EMSOFT'06*.

[40] Ethan K. Jackson and Wolfram Schulte. Model Generation for Horn Logic with Stratified Negation. In *FORTE'08*.

[41] Ethan K. Jackson, Eunsuk Kang, Markus Dahlweid, Dirk Seifert, and Thomas Santen. Components, Platforms and Possibilities: Towards Generic Automation for MDA. In *EMSOFT'10*.

[42] Cristian Cadar and Koushik Sen. Symbolic Execution for Software Testing: Three Decades Later. *Commun. ACM*, 2013.

[43] Daniel Kroening and Michael Tautschnig. CBMC – C Bounded Model Checker. In *TACAS'14*. Springer.

[44] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *TACAS'04*. Springer.

[45] Armando Solar-Lezama, Liviu Tancau, Rastislav Bodik, Sanjit Seshia, and Vijay Saraswat. Combinatorial Sketching for Finite Programs. In *ASPLOS*, 2006.

[46] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitu Padhye, and David Walker. Don't Mind the Gap: Bridging Network-wide Objectives and Device-level Configurations. In *SIGCOMM'16*.

[47] Sanjai Narain, Gary Levin, Sharad Malik, and Vikram Kaul. Declarative Infrastructure Configuration Synthesis and Debugging. *J. Netw. Syst. Manage.*, 2008.