Synthesizing Formal Network Specifications from Input-Output Examples

Haoxian Chen* Chenyuan Wu* Andrew Zhao* Mukund Raghothaman† Mayur Naik* Boon Thau Loo*
*University of Pennsylvania †University of Southern California
{hxchen,wucy,anzhao,mhnaik,boonloo}@seas.upenn.edu raghotha@usc.edu

✦

Abstract—We propose NetSpec, a tool that synthesizes network specifications in a declarative logic programming language from input-output examples. NetSpec aims to accelerate the adoption of formal verification in networking practice, by reducing the effort and expertise required to specify network models or properties. NetSpec aims to be i) highly expressive, capable of synthesizing network specifications with complex semantics; ii) scalable, by virtue of using a novel best-first search algorithm to efficiently explore an unbounded solution space, and iii) robust, avoiding the need for exhaustive input-output examples by actively generating new examples. Our experiments demonstrate that NetSpec can synthesize a wide range of specifications used in network verification, analysis, and implementations. Furthermore, NetSpec improves upon existing approaches in terms of expressiveness, robustness to examples, and the quality of synthesized programs.

Index Terms—Network protocol, program synthesis.

1 INTRODUCTION

Formal specifications are vital for a wide range of networking tasks, including verification [20], [5], [6], [40], [41], analysis [4], [26], [9], and debugging [11], [43]. Network operators who seek to verify properties of their network need a formal specification of the network's protocols [20]. In cloud management, cluster administrators who wish to ascertain reachability of nodes must specify the desired behavior using declarative queries [4], [26]. In distributed systems, programmers who wish to verify certain system properties rely on formal specifications of a wide range of protocols, including inter-domain routing [41], [20], [6], consensus protocols [3], [37], and security protocols [12]. Furthermore, various domain specific languages [31], [25], [17], [7] rely on formal specifications expressed in logic as a basis for generating actual implementations, thereby bridging the specifications-implementation divide.

Despite their promising benefits, formal specifications have not yet gained mainstream adoption in practice. Today, it remains challenging for a network practitioner to write these formal specifications in the first place. It is even harder to ensure that the specifications capture all aspects of the network. Formal languages have steep learning curves, and it is difficult to find engineers who are simultaneously well-versed in network operations and formal methods. Consequently, despite the progress in tools for network verification and analysis, undertaking these tasks still necessitates a formal methods expert who can at least write the desired properties or model the network in formal specification languages.

In this paper, we present NetSpec, a *specification-by-example* (SBE) toolkit that aims to automatically synthesize formal specifications of network protocols in logic. NetSpec aims to make formal network analysis more accessible to network programmers, who do not necessarily have expertise in formal methods. In the SBE paradigm, programmers provide input-output examples of their protocol designs. These designs can be handwritten or derived from actual runtime communication traces. NetSpec then applies program synthesis techniques to automatically yield the logical specifications which are amenable to verification [40] or generation of distributed implementations [25].

Our choice of logic as a basis for NetSpec is motivated by the fact that many formal network models trace their roots to logical specifications. In particular, we target an extension of the declarative logic programming language Datalog [2], which is popular in the literature on network verification [20], [17], [5], [40], [41], analysis [31], [4], [26], debugging [43], [11], and implementation [31], [25], [3], [37]. Thus, our logical specifications can be seen as declarative programs in themselves: the input comprises facts about a network (e.g., topology, VM configurations, etc) or incoming messages (e.g., route requests), while the output comprises actual network state (e.g., the shortest path, the reachable VM pairs, etc) or outgoing messages (e.g., route updates).

We envision NetSpec being used in a variety of settings:

1) rapid prototyping of a protocol design by compiling the synthesized logical specifications into distributed implementations.
2) verifying network protocols at design time by providing input-output examples that can be proof-checked based on its synthesized logical specifications. When a design bug is revealed by a verifier, the user can correct the design by adding new examples.
3) taking a legacy program and deriving its logical specifications from runtime executions for subsequent verification or software analysis. When a verifier finds a counterexample, it can be used to test against the legacy program. If the legacy program exhibits undesired behavior, a real bug is caught. Otherwise, the logical specifications is inaccurate, and can be refined by adding the counter-

example to NetSpec.

To this end, NetSpec provides key features that advance upon state-of-the-art programming-by-example approaches and enable it to effectively address the above use-cases. We next elucidate each of these features:

- **Expressivity.** NetSpec supports expressive features necessitated by complex semantics involved in network specifications. These features include recursion, aggregation, and user-defined functions (UDFs). None of the existing techniques for synthesizing declarative programs support this combination of features, which precludes them from targeting many common network specifications, e.g., routing protocols and consensus protocols.
- **Scalability.** NetSpec uses a novel best-first search algorithm that incrementally proceeds from simple to complex programs, with the ability to rapidly backtrack, which enables to efficiently explore an unbounded search space and produce succinct specifications. In contrast, existing techniques either require the user to bound the search space [24], [33] (e.g., by providing the maximum number of operators), or suffer in terms of efficiency by exploring a large number of incorrect programs [28].
- **Robustness.** NetSpec is robust to the quality of input-output examples. Approaches based on programming-by-example rely on the user to craft a complete set of examples in order to learn the correct program. However, it is easy to miss corner cases when providing these examples manually. NetSpec proactively detects the incompleteness in the specified examples, and generates new input queries to the example provider—a network operator or a legacy implementation. These new inputs, together with the provider's answers as the outputs, improve the example quality and enable NetSpec to unambiguously learn a correct program.

We have developed a prototype of NetSpec and evaluate it on a suite of 26 benchmarks that encompass a wide range of network protocols in different sub-domains, including network analysis, software-defined networking (SDN), sensor networks, routing protocols, and consensus protocols. Our experiments demonstrate that NetSpec can faithfully synthesize most logical specifications in under a few seconds, with the most complex one in slightly more than 1 minute. In contrast, state-of-the-art tools GenSynth [28] and Scythe [42] cannot synthesize benchmarks requiring either aggregation or user-defined functions (10 out of 26), and benchmarks requiring recursion or user-defined functions (11 out of 26), respectively. Moreover, the specifications synthesized by NetSpec can be directly compiled into declarative networking [31], [25] for distributed implementations.

To validate NetSpec on actual implementations, we further demonstrate that NetSpec is able to synthesize logical specifications from actual program execution traces derived from popular open-source SDN controller implementations written in Floodlight [19] and POX [32], highlighting its ability to synthesize specifications for large-scale programs.

Contributions. To summarize, the key technical contributions of this paper are as follows:

- We propose a novel synthesis algorithm to efficiently synthesize highly expressive network specifications from input-output examples. The specifications, expressed in first-order relational logic, have a variety of uses including verifying, analysing, and generating implementations.

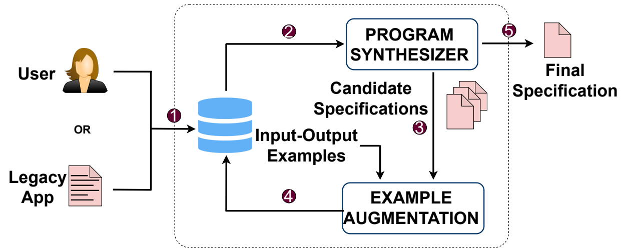


Fig. 1: Architecture of NetSpec.

- Since programming-by-example approaches are susceptible to missing examples, we develop a novel example generation algorithm to supplement synthesis. It queries the example provider for new examples that guide the synthesis algorithm to an unambiguous specification.
- We realize our approach in a tool NetSpec and evaluate it on diverse benchmarks and use-cases. NetSpec is able to correctly synthesize a wide-range of network protocols within seconds and is robust to missing examples. Moreover, we demonstrate that NetSpec outperforms state-of-the-art synthesis approaches in terms of its expressiveness, and in the quality of its synthesized programs.

2 ILLUSTRATIVE EXAMPLE

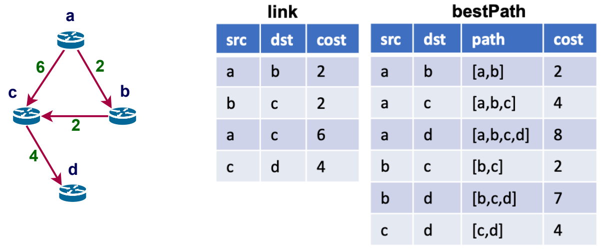
In this section, we illustrate the end-to-end operation of NetSpec using the shortest path routing protocol as an example. The overall architecture of NetSpec is depicted in Figure 1. In Sections 2.1, 2.2, and 2.3, we describe the input-output examples, the synthesis algorithm, and the example augmentation process respectively.

2.1 Problem Specification

NetSpec takes two kinds of input: (1) A set of input-output example pairs, where each pair is consist of a set of input tables, and a set of output tables. These tables are relational, where each row is interpreted as relational tuples in Datalog. (2) Optionally, a list of user-defined functions and aggregators that could appear in the output specification. And NetSpec returns a logical specification, in the syntax of Datalog, that is consistent with the input-output examples. In the remainder of this paper, we will use "specification" and "program" to refer to NetSpec's output interchangeably.

Figure 2a depicts such an example for our shortest path routing protocol. In this example, one input-output pair is provided. The input table is named `link`, describing the network topology as a weighted graph. And the output table is named `bestPath`, specifying an optimal path for each pair of source and destination nodes. Functions including list initialization ($l = [x, y]$), concatenation ($x :: l$), and membership checking ($x \text{ in } l$), and aggregators (min and max) are also provided.

From this data, NetSpec automatically synthesizes the declarative logical specification shown in Figure 2b. We have expressed the specification using the syntax for Datalog, which we briefly review in Section 3. The first two rules specify paths between pairs of nodes and their associated



link

src	dst	cost
a	b	2
b	c	2
a	c	6
c	d	4

bestPath

src	dst	path	cost
a	b	[a,b]	2
a	c	[a,b,c]	4
a	d	[a,b,c,d]	8
b	c	[b,c]	2
b	d	[b,c,d]	7
c	d	[c,d]	4

(a) A small network topology as a weighted directed graph (left), its relational representation (middle), and the expected output (right).

```
// compute available paths
r1: path(x, y, p, c) :- link(x, y, c), p=[x, y].
r2: path(x, y, x::p1 ,c1+c2) :- link(x, z, c1),
   path(z, y, p1, c2), !(x in p1).
// select the minimum cost path
r3: minCost(x, y, min<c>) :- path(x, y, _, c).
r4: bestPath(x, y, p, mc) :- path(x, y, p, mc),
   minCost(x, y, mc).
```

(b) Specification synthesized by NetSpec.

Fig. 2: Example of a shortest path routing protocol specification.

costs: rule `r1` specifies a network link as a one-hop path, and rule `r2` specifies the transitive case. In particular, `x::p1` prepends node `x` to the head of path `p1`, and `!(x in p1)` checks that `x` is not in path `p1`, to avoid generating loops and to enforce termination. Rules `r3` and `r4` select the path with the minimum cost as the output best path.

This specification provides a high-level abstraction for verifying route convergence properties [41] and explaining route derivations [45]. Similarly, logical specification of other routing protocols can also be used to reason about network connectivity under different network dynamics [26], [20].

Despite the simplicity of the final specification, several aspects of the synthesis problem make it challenging in practice. First, the search space is enormous. For example, the rule `r2` contains 13 variable occurrences, so that there are $13! \approx 10^9$ ways of filling in its variables even after the rest of the rule structure is fixed. Furthermore, interaction between the rules makes the problem non-compositional, and techniques which synthesize one rule at a time become inapplicable [30], [15]. Finally, because input-output examples often under-specify the target concept and because of the undecidability of program equivalence [2], it is difficult to determine whether the synthesized specification correctly captures the user's intent.

2.2 Synthesis by Optimization

We organize the synthesis algorithm as an optimization problem and illustrate the process in Figure 3. Each node in the figure represents a candidate program, and its outgoing edges indicate each of its possible offspring. We highlight critical steps that lead to the final program and defer details of the algorithm to the next two sections.

Conceptually, we consider three possible modifications to each candidate program: introducing rules, introducing literals within a rule, and introducing aggregation operators. In the rest of this section, we first describe the overall search strategy for applying the modifications, and then outline each one of the three modification steps.

Search strategies. The objective function of the optimization problem is based on two measures of success on a candidate program s:

$$\text{score}(s) = \text{precision}(s) \times \text{recall}(s) \tag{1}$$

In particular, given the set of expected output tuples O_{exp}, and a candidate specification s that produces set of output tuples O_{ret}, we calculate $\text{precision}(s) = |O_{exp} \cap O_{ret}|/|O_{ret}|$, which is the fraction of tuples produced which are expected, and $\text{recall}(s) = |O_{exp} \cap O_{ret}|/|O_{exp}|$, which is the fraction of expected tuples which are produced by the candidate specification. The $\text{score}(s)$ is discounted by a $\gamma(s)$ metric that is a fraction of columns whose column values are all known given s.

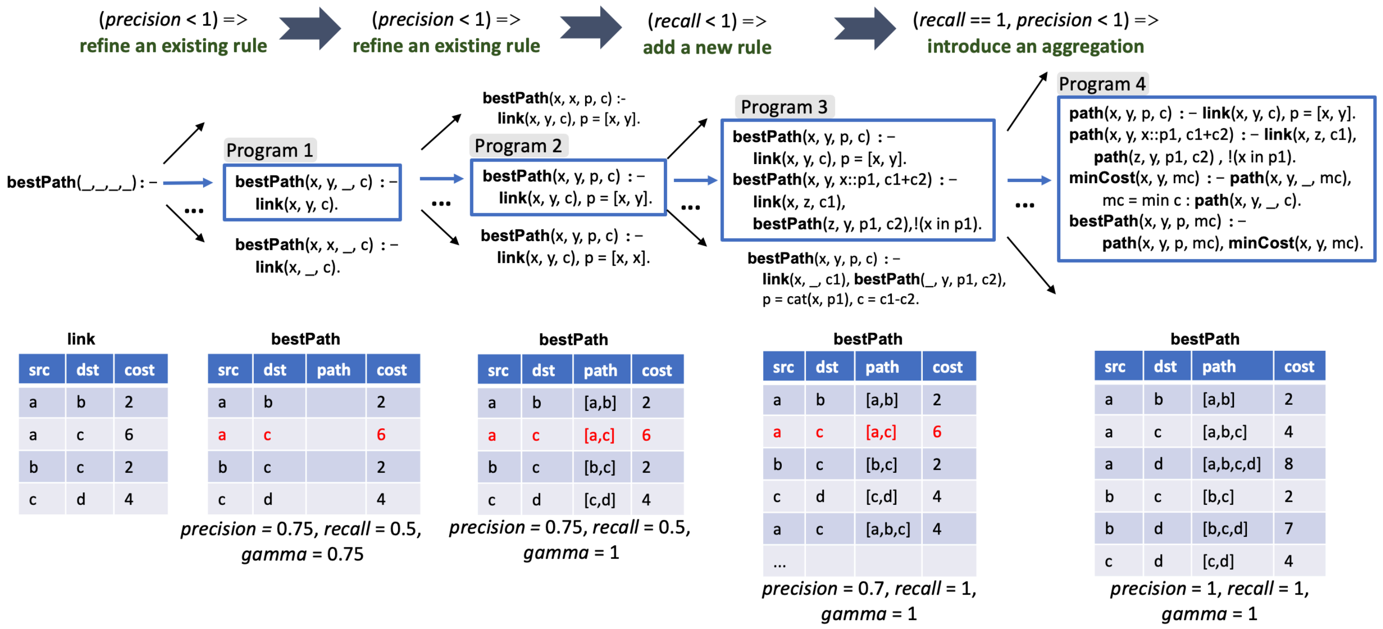
Starting with an empty program, with score 0, the synthesis algorithm repeatedly generates offspring programs by applying all mutation strategies, and adds these offspring into a set of candidate programs. The next program to mutate is sampled from offspring that have higher scores, or the whole set of candidate programs when no offspring has a higher score.

In Figure 3, the input relation `link` generates 3 of the 6 expected `bestPath` tuples in Programs 1 and 2. For instance, the rule `bestPath(x,y,p,c):- link(x,y,c), p=[x,y]` has a recall of 0.5 and a precision of 0.75, and has the highest score among all candidate programs. A red color `bestPath` tuple denoting the shortest path from `a` to `c` is incorrect and needs to be fixed. Moreover, some `bestPath` tuples are missing. In subsequent steps, the candidate program with the highest score is successively modified to include the transitive rule for paths, and the aggregation operation to select the optimum weight path. Eventually, the red tuple is corrected, the missing tuples generated, and we converge on the best paths that matches the given input-output examples.

We next describe the three modification steps that can be applied to the current best candidate program. Each modification is described by referencing the generated candidate programs (1–4) in Figure 3.

Modification 1: Introducing new rules. The algorithm begins by enumerating single rule programs which produce at least one expected output tuple. The same rule generation algorithm is also invoked when the intermediate program fails to produce a desired output tuple, i.e., it has imperfect recall (less than 1). Each synthesized rule is chosen so that it produces at least one desirable tuple which is currently missing. These rules are synthesized by repeatedly introducing literals (modification 2) to the set of minimal rules, which contain only one head literal and one body literal, until it produces at least one expected output tuple.

We illustrate this process for the running example in Figure 3. Midway through running the best-first search algorithm after two refinement steps, the precision and



```
bestPath(x, y, _, c) : –
   link(x, y, c).

bestPath(x, x, _, c) : –
   link(x, _, c).
```

```
bestPath(x, x, p, c) :-
   link(x, y, c), p = [x, y].

bestPath(x, y, p, c) : –
   link(x, y, c), p = [x, y].

bestPath(x, y, p, c) : –
   link(x, y, c), p = [x, x].
```

```
bestPath(x, y, p, c) : –
   link(x, y, c), p = [x, y].
bestPath(x, y, x::p1, c1+c2) : –
   link(x, z, c1),
   bestPath(z, y, p1, c2),!(x in p1).

bestPath(x, y, p, c) : –
   link(x, _, c1), bestPath(_, y, p1, c2),
   p = cat(x, p1), c = c1-c2.
```

```
path(x, y, p, c) : – link(x, y, c), p = [x, y].
path(x, y, x::p1, c1+c2) : – link(x, z, c1),
    path(z, y, p1, c2) , !(x in p1).
minCost(x, y, mc) : – path(x, y, _, mc),
    mc = min c : path(x, y, _, c).
bestPath(x, y, p, mc) : –
    path(x, y, p, mc), minCost(x, y, mc).
```

link

src	dst	cost
a	b	2
a	c	6
b	c	2
c	d	4

bestPath

src	dst	path	cost
a	b		2
a	c		6
b	c		2
c	d		4

precision = 0.75, *recall* = 0.5, *gamma* = 0.75

bestPath

src	dst	path	cost
a	b	[a,b]	2
a	c	[a,c]	6
b	c	[b,c]	2
c	d	[c,d]	4

precision = 0.75, *recall* = 0.5, *gamma* = 1

bestPath

src	dst	path	cost
a	b	[a,b]	2
a	c	[a,c]	6
b	c	[b,c]	2
c	d	[c,d]	4
a	c	[a,b,c]	4
...			

precision = 0.7, *recall* = 1, *gamma* = 1

bestPath

src	dst	path	cost
a	b	[a,b]	2
a	c	[a,b,c]	4
a	d	[a,b,c,d]	8
b	c	[b,c]	2
b	d	[b,c,d]	7
c	d	[c,d]	4

precision = 1, *recall* = 1, *gamma* = 1

Fig. 3: NetSpec synthesis procedure. On the left is the input table `link`. The highlighted blue boxes show the three intermediate programs leading up to the final solution. The tables underneath describe their respective outputs.

recall of the best candidate program are 0.75 and 0.5 respectively. At this point, the best candidate program is only able to generate one-hop best paths by virtue of the rule `bestPath(x,y,p,c):- link(x,y,c),p=[x,y]` (Program 2). New rules need to be added so that one can generate outputs for paths that are two-hops and beyond, and this is done by adding the recursive rule that contains $bestPath$ in the rule body. Upon adding this new rule, the recall of the resulting output (Program 3) is increased to 1 although the precision is still not yet 1 (pending one additional modification to introduce aggregates).

Modification 2: Rule refinement by introducing literals. If the precision of a candidate program is less than 1, the algorithm adds new constraints to its rules by introducing literals, or by augmenting them with aggregation operations. By adding new literals to its rules, the algorithm produces an offspring program s' which produces a subset of the output tuples produced by the original program s.

To provide some intuition on rule refinement, we consider the scenario shown in Figure 3 where the current best candidate is the partial rule `bestPath(x,y,_,c):- link(x,y,c)` (Program 1). Since the third column of the output relation has not yet been specified, the algorithm scores this partial rule by only comparing the remaining columns to the reference output. Intuitively, the partial rule mispredicts the cost of the `(a,c)` path, so that the program has a precision of 0.5 and a recall of 0.75. This score is additionally discounted by a factor of $\gamma = 0.75$ to account for incompleteness in output and bias the rule search towards faster rule completion. At this point, the rule refinement adds a literal $p = [x, y]$ where $[\,]$ is a path concatenation function which is one of the candidate user-defined functions provided to the synthesis algorithm. Interestingly, with this refinement, while $precision$ is unchanged in the resulting output (Program 2), γ increases to 1 and all column values are known.

```
path(x, y, p, c) : – link(x, y, c), p = [x, y].
path(x, y, x::p1, c1+c2) : – link(x, z, c1),
   path(z, y, p1, c2), !(x in p1).
minCost(x, y, mc) : – path(x, y, _, _),
   mc = min c : path(x, y,_, c).
bestPath(x, y, p, mc) : –
   path(x, y, p, mc),  minCost(x,y,mc).
```

```
path(x, y, p, c) : – link(x,y,c), p = [x, y].
path(x, y, x::p1, c2+c2) : – link(x, z, c1),
   path(z, y, p1, c2), !(x in p1).
minCost(x, y, mc) : – path(x, y, _, _),
   mc = min c : path(x, y, _, c).
bestPath(x, y, p, mc) : –
   path(x, y, p, mc), minCost(x, y, mc).
```

Fig. 4: Two solutions of the routing protocol specified with incomplete examples. The difference is highlighted.

Observe that this process of adding literals provides flexibility in supporting arbitrary functions because it makes no assumptions about the underlying semantics. It is also highly efficient because it only considers one literal at a time, instead of arbitrary combinations of literals.

Modification 3: Aggregation operators. The final way to modify a program output is to apply an aggregation operation to produce one of its output columns. Consider the rule `r3` which finds the length of the shortest path between `x` and `y`. Informally, the aggregation operator `min` first groups the output tuples by their source and destination nodes, `(x,y)`, and then aggregates over all possible values of `c` for which a path exists: `path(x,y,_,c)`. In Figure 3, after adding the `min` aggregate to a candidate (Program 3), the algorithm converges upon the final solution (Program 4).

2.3 The Example Augmentation Process

The synthesis algorithm discovers all programs which are consistent with the input-output examples up to a maximum depth. When the provided input-output examples only partially constrain the possible solutions, the algorithm may discover multiple solutions, all of which are consistent with the data. We show two possible solutions to the shortest path routing program in Figure 4, and highlight their difference in yellow. In general, dealing with under-

(input relation)	`I`		
(output relation)	`O`		
(function)	`F`		
(aggregation)	`A`	$\in$	{ `min`, `max`, `count` }
(variable)	x		
(body literal)	b	::=	`I`$(\bar{x})$ $\mid$ `!I`$(\bar{x})$ $\mid$ `O`$(\bar{x})$ $\mid$ `F`$(\bar{x})$
(head argument)	a	::=	x $\mid$ `A`(x)
(head literal)	h	::=	`O`$(\bar{a})$
(rule)	r	::=	h `:-` $b_1, ..., b_n$
(specification)	p	::=	$\{ r_1, ..., r_n \}$

Fig. 5: Abstract syntax of specifications in NetSpec. Input relation (I), output relation (O), user-defined functions (F), and aggregation (A) are application-specific.

constrained specifications is a major outstanding challenge in programming-by-example (PBE) systems, and solution disambiguation is an important contribution of this paper.

One reason for the difficulty of disambiguation is that the equivalence checking problem for Datalog programs is undecidable [2]. To address this, NetSpec employs the idea of differential testing from program analysis [27] to repeatedly run the two programs with randomly perturbed inputs. In our example, by modifying the link costs, one obtains an input which reveals the difference between the two programs. We can then request the user to provide the ground truth for this new example, which will in turn eliminate at least one of the candidate solutions. The process repeats until only one program remains, or NetSpec fails to generate a distinguishing input among the programs. In this latter case, NetSpec produces the simplest program as the final solution. Because the enumeration process is biased towards smaller programs, and because the tie-breaking routine favors the syntactically smallest solution, in practice NetSpec produces small programs that are also readily interpretable and resistant to over-fitting.

3 THE NETSPEC SPECIFICATION LANGUAGE

This section provides a more formal overview of the language of specifications synthesized by NetSpec. The design of the language is motivated by two key goals: the ability to express a wide range of network specifications, and the ability to leverage a variety of network verifiers, analyzers, and implementations.

Figure 5 presents the abstract syntax of specifications. We elucidate it using our running example of the shortest-path routing specification shown in Figure 2b. A specification is a program whose inputs and outputs are a set of relations. In our routing example, the input relations include `link`, which represents the network topology, as well as common predicates such as `in` (list membership). The output relations include `bestPath`, which represents the shortest path between every pair of nodes in the input network, as well as relations such as `path` and `minCost` which hold intermediate results needed to compute `bestPath`.

A specification comprises a set of rules that specify how to compute the output relations from the input relations. Our routing example comprises four rules denoted `r1` through `r4`. Each rule is a Horn clause of the form:

$$R_h(\bar{x_h}) \texttt{ :- } R_1(\bar{x_1}), ..., R_n(\bar{x_n})$$

where the $\bar{x_i}$'s are vectors of variables of appropriate arity. Each rule is read from right-to-left as a universally quantified implication: for all variable valuations $\bar{x}$, if each of tuples $R_1(\bar{x_1})$, ..., $R_n(\bar{x_n})$ are derivable, then so is $R_h(\bar{x_h})$.

For instance, rule `r4` in our routing example states that if $\texttt{path}(x, y, p, mc)$ and $\texttt{minCost}(x, y, mc)$ are derivable, then so is $\texttt{bestPath}(x, y, p, mc)$. This rule also depicts a basic logic operation: conjunction (i.e., join). On the other hand, disjunction (i.e., union) is expressed by means of different rules with the same head relation, as illustrated by rules `r1` and `r2` which denote the base case and inductive step, respectively, for computing the `path` relation. These two rules also illustrate recursion—an operation commonly needed in network specifications to specify reachability properties.

The features described thus far constitute the declarative logic programming language Datalog [2]. However, Datalog is inadequate to express real-world network specifications with rich functionality. The specification language of NetSpec therefore extends Datalog with three additional kinds of operations: negation (denoted `!`), aggregation (e.g., `min` and `count`), and user-defined functions, which include common utility functions such as `::` (list prepend) and `+` (integer addition). To ensure well-founded semantics (Datalog programs with negations should be stratified [2, Chapter 15]), NetSpec only applies negations to input relations or functions, e.g., to function `in` in rule `r2`. In addition, to keep the synthesis task tractable, NetSpec applies the following syntactic restrictions to each rules:

1) Each rule can have at most 2 literals of the same relation. For example, a rule $h(x, w)$:– $p(x, y), p(y, z), p(z, w)$ would not be generated by NetSpec because it has 3 literals of relation "p".
2) A negation literal can have at most 2 bound variables. For example, literal $!p(a, b, c)$ would not be added to rules, because it has 3 bound variables (a, b, c). But literal $!p(a, b, \_)$ could be added.
3) At most one aggregation is used in each program.
4) Aggregation can only be applied in the head of a rule. For instance, applying `min` in rule `r3` yields the minimum cost c over all paths between each pair of nodes x and y in the input network.
5) A user-defined function's result can only be used in the head of a rule, e.g., the result of + in rule `r2`.

In evaluation, we show that these syntactic restrictions have no impact on all declarative specifications from prior literature, except PAXOS, which has two layers of aggregations. We show how to synthesize such complex protocols by breaking it down into independent modules in Section 7.1.

Specifications are executable programs: execution begins with all output relations initialized to empty, and proceeds by repeatedly evaluating the rules until the output relations stop changing. The syntactic restrictions described above ensure a deterministic result regardless of rule evaluation order. However, note that the presence of recursion together with user-defined functions can lead to non-termination (e.g., by recursively applying integer addition). NetSpec thus supports a highly expressive class of specifications.

An important benefit of the specifications synthesized by NetSpec is their suitability for a variety of networking tasks. They can be verified using SDN verifiers such as Vericon [5] and FlowLog [31], and routing verifiers such as Batfish [20] and FSR [41]; They can be analyzed using network analysis tools such as NOD [26],Tiros [4] and ExSPAN [45]. Lastly, they can be compiled to distributed implementations in Network Datalog [25] and FlowLog [31].

4 SYNTHESIS ALGORITHM

Algorithm 1 $\mathrm{Synth}(I, O, F)$. Given a set of input tuples I, expected output tuples O, and a library of functions F, produces all consistent programs.

```
1) Initialize the set of solutions, $S := \emptyset$, the set of candidate programs, $Q := \{P_0\}$, and the current program $P := P_0$, where $P_0$ is the empty program.
2) While $Q \neq \emptyset$, do:
   a) Let $\mathrm{Offspring}(P) = \{P'_1, P'_2, \dots\}$ be the offspring of $P$, computed according to Equation 2.
   b) Update the set of solutions, and add all remaining programs for further enumeration:
        $S := S \cup \{P' \in \mathrm{Offspring}(P) \mid \mathrm{score}(P') = 1\}$, and
        $Q := (Q \setminus P) \cup \{P' \in \mathrm{Offspring}(P) \mid \mathrm{score}(P') > 0\}$.
   c) Sample the next program to explore:
        $\mathrm{HS} := \{P' \in \mathrm{Offspring}(P) \mid \mathrm{score}(P') > \mathrm{score}(P)\}$
        $\mathrm{HR} := \{P' \in \mathrm{Offspring}(P) \mid \mathrm{recall}(P') > \mathrm{recall}(P)\}$
        $P := \begin{cases} \mathrm{Sample}(\mathrm{HS}, P) & \text{if HS} \neq \emptyset \\ \mathrm{Sample}(\mathrm{HR}, P) & \text{else if HR} \neq \emptyset \\ \mathrm{Sample}(Q, P) & \text{otherwise.} \end{cases}$
3) Return $S$.
```

$$
\begin{aligned}
\mathrm{Offspring}(P) &= O_D(P) \cup O_C(P) \cup O_A(P), \text{ where} \qquad (2)\\
O_D(P) &= \begin{cases} \mathrm{AddRule}(P) & \text{if } \mathrm{recall}(P) < 1, \text{ and} \\ \emptyset & \text{otherwise,} \end{cases}\\
O_C(P) &= \begin{cases} \mathrm{ExtRule}(P) & \text{if } \mathrm{precision}(P) \leq 1, \text{ and} \\ \emptyset & \text{otherwise, and} \end{cases}\\
O_A(P) &= \begin{cases} \mathrm{MkAgg}(P) & \text{if } \mathrm{precision}(P) \leq 1 \text{ and} \\ & \quad \mathrm{recall}(P) = 1, \text{ and} \\ \emptyset & \text{otherwise.} \end{cases}
\end{aligned}
$$

We present the top-level synthesis procedure in Algorithm 1. It takes only a set of input tuples (I), and a set of output tuples (O), and we will explain how to support multiple instances of input-output example pairs in section 4.4. As described in Section 2, it models an optimization problem in the Datalog program space, where each state is a program. And the objective function $\mathrm{score}(p)$ is defined as the product of $\mathrm{precision}(p)$ and $\mathrm{recall}(p)$.

At each iteration, the algorithm explores the program space by mutating the current program P, which gives rise to several offspring (step 2a). $\mathrm{Offspring}(P)$ is defined in equation 2, where the D, C, and A subscripts indicate the generation of offspring by adding new rules (disjunctions), extending existing rules (conjunctions), and by applying aggregation operators, respectively. The conditions to apply each of these mutation strategy are based on the semantics of Datalog. Both adding clause in a conjunction rule, and aggregate the output of current program, monotonically decrease the size of program output (number of tuples), thus may improve precision, but may also lower recall at the same time. Thus they are only applied when the program has imperfect precision. Adding a rule, on the contrary, monotonically increase the size of program output, and could potentially improve recall, but lower precision at the same time. Therefore it is only applied when the program has imperfect recall. In addition, we assume the program space where only one aggregator is used, thus we wait until all necessary rules are added to reach perfect recall before applying aggregation.

In step 2b, offspring with score 1 are added to the solution set S. Offspring with score 0 implies that it produces no desired output ($P(I) \cap O = \emptyset$). Such offspring are discarded, based on the previous observation that, applying $\mathrm{ExtRule}$ or MkAgg to a program monotonically decreases the output size of the program. This means that further extending any rule of this program would not produce any desired output, except adding new rules. In addition, we assume that in all solution programs, every non-aggregate rule directly contributes to some output in O. Therefore, only rules with non-zero score ($P(I) \cap O \neq \emptyset$) are added into the set of candidate programs (Q) for further mutations.

Algorithm 2 $\mathrm{Sample}(Q, P)$. Given a set of candidate programs Q, the current program P, return a program $P' \in Q$.

```
For $k \in \{1, 2, ..., K_{max}\}$, do:
  1) Uniformly sample a program $P'$ from $Q$.
  2) Compute acceptance probability of $P'$:
       $s_0 := \mathrm{score}(P), s_1 := \mathrm{score}(P')$
       $T := 1 - \frac{k}{K_{max}}$
       $Pr[\text{accept } P'] := \begin{cases} 1 & \text{if } s_1 > s_0 \\ \exp(-\frac{s0-s1}{T}) & \text{otherwise} \end{cases}$   (3)
  3) If $Pr[\text{accept } P'] \geq \mathrm{random}(0, 1)$:
       • return $P'$
```

In step 2c, the next program to explore is sampled probabilistically. When there are offspring with higher score or higher recall, these offspring will always be chosen as the next program to explore. Otherwise, it samples from the whole set of candidate programs Q. The sub-routine $\mathrm{Sample}(Q, P)$ is described in algorithm 2. Borrowing the idea in simulated annealing, it iteratively samples a candidate $P' \in Q$ uniformly, and accept it with probability computed by equation 3. Intuitively, when a candidate program has higher score than current program, it is accepted with probability 1. Otherwise, it is accepted with probability between 0 to 1, depending on how worse its score compared to the current program.

The rest of this section describes each of these mutation strategies, and formal properties of the synthesis algorithm.

4.1 Adding and extending Rules

The $\text{AddRule}(P)$ procedure enumerates all minimal rules and generate offspring by adding one minimal rule to P. A minimal rule is a rule that has only one literal in the body, and the head only have one field bound to the body, with all remaining fields being empty place holders ("_"). In the short-path routing example, one of the minimal rules is:

```
r_0: bestPath(x,_,_,_) :- link(x,_,_).
```

Let MinimalRules be the set of all minimal rules obtained from the given input and output relations, $\text{AddRule}(P)$ is defined as:

$$\text{AddRule}(P) := \{(P \cup r) | r \in \text{MinimalRules}\} \tag{4}$$

Next we introduce "ExtRule(P)" procedure, which further contains two atomic operations on a rule, namely "AddLiteral(r)" and "AddBinding(r)". They are defined as:

$$\text{AddLiteral}(r) = \{r \wedge l | r \in P, l \in L\} \tag{5}$$

where L is the set of all literals whose relation is from the set of all input relations, output relations, and user-defined functions, and contains only empty place holders "_". $r \wedge l$ represents a new rule by adding literal l in conjunction with r's body. Continuing on the example on shortest-path routing, one of the new rules generated by "AddLiteral(r_0)" is:

```
r_1: bestPath(x,_,_,_) :- link(x,_,_),
     bestPath(_,_,_,_).
```

where $bestPath(\_,\_,\_,\_)$ is a literal instantiated from the output relation $bestPath$.

Next, "AddBinding(r)" is defined as follow:

$$\begin{aligned} \text{AddBinding}(r) =& \{r \wedge (v_1 = v_2) | v_1, v_2 \in r \\ & \wedge dom(v_1) = dom(v_2)\} \end{aligned} \tag{6}$$

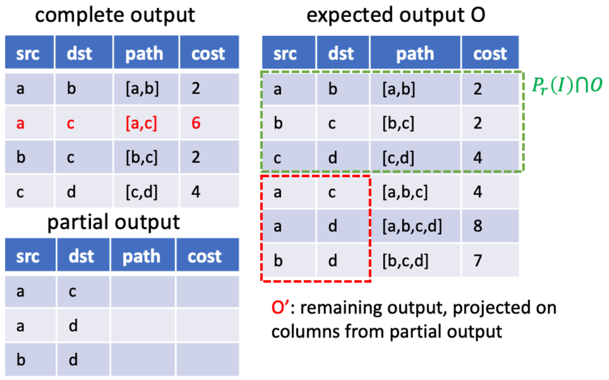
where $v_1, v_2 \in r$ means that variable v_1 and v_2 appear in the rule r, and $dom(v)$ is the domain of variable v, as specified in the schema of the literal where v appears. As an example, we show one of the rules generated by "AddBinding(r_1)":

```
r_2: bestPath(x,_,_,_) :- link(x,z,_),
     bestPath(z,_,_,_).
```

where the second variable in literal $link$ is bound with the first variable in literal $bestPath$ in the body. Note that instead of explicitly add a predicate that match two variables as: $bestPath(x,\_,\_,\_) \ :- link(x, v1, \_), bestPath(v2,\_,\_,\_), v1 = v2.$, we rename $v1$ and $v2$ to z for brevity.

Putting them together, "ExtRule(P)" generates all programs resulted from applying either "AddLiteral" or "AddBinding" to any one of the rules in program P. "ExtRule(P)" is defined as:

$$\begin{aligned} \text{ExtRule}(P) =& \{(P \setminus r) \cup r' | r \in P, \\ & r' \in (AddLiteral(r) \cup AddBinding(r))\} \end{aligned} \tag{7}$$



complete output

src	dst	path	cost
a	b	[a,b]	2
a	c	[a,c]	6
b	c	[b,c]	2
c	d	[c,d]	4

partial output

src	dst	path	cost
a	c		
a	d		
b	d		

expected output O

src	dst	path	cost
a	b	[a,b]	2
b	c	[b,c]	2
c	d	[c,d]	4
a	c	[a,b,c]	4
a	d	[a,b,c,d]	8
b	d	[b,c,d]	7

$P_r(I) \cap O$
O′: remaining output, projected on columns from partial output

TABLE 1: Example of scoring a partial program P_r. It contains a partial rule r, where only two fields in the head are determined, thus only two columns are generated by this rule. Precision is 0.86 because 6 out of the 7 output tuples are desired (in $O \cup \pi_c(O)$). Recall is composed of two parts, the complete tuples (3 in green box), and the partial tuples (3 in red box). The recall on the partially generated output is discounted by factor 0.5 because only 2 out of 4 columns are generated. Putting them together, the total recall is $\frac{3+3\times0.5}{6} = 0.75$.

4.2 Evaluating partial programs

When applying $\text{AddRule}(P)$ and $\text{ExtRule}(P)$, we will have partial rules in the program queue. By partial rule we mean rules that have empty place holders in the head. We further define partial programs as programs that contains at least one partial rule.

As an example, consider the four-place `bestPath(x, y, p, c)` relation, and a partial rule as the following:

```
r_p1: bestPath(x, y, _, _) :- link(x, z, _),
      bestPath(z, y, _, _).
```

Notice that this rule only produces the first two columns of the output relation, the source node and and the destination node, but does not produce the remaining two columns, the optimum path, and its length.

As a consequence, programs with these partial rules, such as $P \cup \{r_{p1}\}$, cannot be directly compared to the entire reference output O, and we are instead only able to compare its first two columns to $\pi_{\text{src,dest}}(O)$, borrowing the notation for projections from relational algebra.

This leads us to the following definition of precision and recall for partial programs P_r:

$$\text{precision}_d(P_r) = \frac{|P_r(I) \cap (O \cup \pi_c(O))|}{|P(I)|}, \tag{8}$$

$$O' = \{t \in (O \setminus P(I)) | \ \pi_c(t) \in P(I)\}$$

$$\text{recall}_d(P_r) = \frac{|P(I) \cap O| + \gamma |O'|}{|O|} \tag{9}$$

where π_c is the projection operator that project to the columns that have been bound on the partial rule's head, The set O' is the subset O that are not in $P(I)$, but whose projection on columns c appear in $P(I)$, we visualize this set computation in Table 1.

4.3 Introducing Aggregation Operations

Algorithm 3 $\mathrm{MkAgg}(P)$. Produces offspring of P by introducing aggregation operators.

```
1) Let $F$ be the family of aggregation operations, and Let $C$ be the set of all columns in output relation $R$.
2) For each operator $\mathrm{op} \in F$, for each subset $C_{\mathrm{agg}} \subseteq C$, and for each aggregation column $f \in C \backslash C_{\mathrm{agg}}$, construct the offspring program $P'$: First rename the output relation $R$ of $P$ to $R_{\mathrm{base}}$, and let $C_{rem}$ be the remaining columns $C \setminus C_{\mathrm{agg}} \setminus \{f\}$. Then add the following two rules.
       $R_{\mathrm{opt}}$($C_{\mathrm{agg}}$, $f_{\mathrm{opt}}$) :- $R_{\mathrm{base}}$($C_{\mathrm{agg}}$, $f_{\mathrm{opt}}$, _),
           $f_{\mathrm{opt}}$ = op $f$: $R_{\mathrm{base}}$($C_{\mathrm{agg}}$, $f$, _).
       $R$($C_{\mathrm{agg}}$, $f_{\mathrm{opt}}$, $C_{\mathrm{rem}}$) :- $R_{\mathrm{opt}}$($C_{\mathrm{agg}}$, $f_{\mathrm{opt}}$),
           $R_{\mathrm{base}}$($C_{\mathrm{agg}}$, $f_{\mathrm{opt}}$, $C_{\mathrm{rem}}$).
3) Return all offspring produced in Step 2.
```

Algorithm 3 describes MkAgg Procedure. Recall the third program in our running example of Figure 3:

```
r1: bestPath(x, y, p, c) :- link(x, y, c), p = [x, y].
r2: bestPath(x, y, x::p, c1 + c2) :- link(x, z, c1),
     bestPath(x, y, p, c2), !(x in p1).
```

This program correctly predicts the reachability relation, but it produces additional incorrect paths, i.e., it has perfect recall but imperfect precision. In this case, NetSpec attempts to remedy the situation by introducing aggregation operators. It introduces two new rules, which may be informally interpreted as follows: The first rule selects a subset of columns (in this case, the source node `x` and the destination node `y`), and performs an aggregation on another column (in this case, computing the minimum of all path weights which share the source and destination nodes). The second rule then selects the values of the remaining columns which lead to this maximization or minimization objective. Thus, after mutation, the following program is added to the queue:

```
r1: path(x, y, p, c) :- link(x, y, c), p = [x, y].
r2: path(x, y, x::p, c1 + c2) :- link(x, z, c1),
     bestPath(x, y, p, c2), !(x in p1).
r3: minPath(x, y, mc) :- path(x, y, _, mc),
     mc = min c: path(x, y, _, c).
r4: bestPath(x, y, p, mc) :- minPath(x, y, mc),
     path(x, y, p, mc).
```

4.4 Supporting multiple input-output example pairs

So far our algorithm description is based on one set of input tuples (I) and one set of output tuples (O). To support multiple instances of examples, NetSpec introduces an additional field 'InstanceID' to every tuple, which indicates the particular example instance that the tuple belongs to. Tuples across different instances are then combined into one set of input tuples (I), and one set of output tuples (O), respectively. During the rule search process, NetSpec only generates rules that bind all 'InstanceID' fields to one single variable. For example, a rule generated by NetSpec would look like the following:

```
h(v1,v2,i) :- p1(v1,v3,i), p2(v3,v2,i).
```

where all variables for 'InstanceId' field are bound to the same name i. Thus, the synthesis problem with multiple example instances is reduced to one with single instance, which is solved by Algorithm 1.

4.5 Soundness and completeness

Theorem 1 (Soundness). *Given a set of input tuples and a set of output tuples (I,O), when NetSpec terminates, its output S satisfies the following property:*

$$\forall p \in S, p(I) = O. \tag{10}$$

Proof sketch. In algorithm 1, a program p is added to solution set S if and only if $\mathrm{score}(p) = 1$ (step 2b). To prove Theorem 1 suffice to show that:

$$\mathrm{score}(p) = 1 \implies p(I) = O \tag{11}$$

where $\mathrm{score}(p)$ is defined in equation 1. By the definition, when $\mathrm{score}(p) = 1$, program output $p(I)$ has perfect precision and recall on the reference output O. This implies that $p(I) = O$.

Next, we state the completeness property. We first define the program space, using the following definitions:

Definition 1 (Empty program). *Program p is an empty program if and only if it consists of no rule.*

Definition 2 (Successor relation). *Let $\rightarrow$ be a binary relation on the set of Datalog programs:*

$$p \rightarrow q \iff q \in \mathit{Offspring}(p) \tag{12}$$

Let $\rightarrow^*$ be a binary relation on the set of Datalog programs:

$$p \rightarrow^* q \iff p \rightarrow p_1 \rightarrow \ldots \rightarrow p_n \rightarrow q \tag{13}$$

where $n \geq 0$.

Definition 3 (Output-contributing rule). *Given a set of input tuples and a set of output tuples (I,O), a rule r in a program p is an output-contributing rule if r's evaluation result on input I intersects with O.*

A special case is for programs with aggregations (either $argMax$ or $argMin$). If r's output is aggregated, then r's result is compared with renamed tuples in O, whose relations are renamed as r's output relation. In the shortest-path example (Figure 2b), $path$ relation is aggregated into $bestPath$, when determining if $r1$ is contributing to output, we rename relations of tuples in O from $bestPath$ to $path$, and then check intersection. If r is an aggregation rule, because NetSpec introduces an aggregation (min or max) rule and a selection rule simultaneously to achieve $argMin$ or $argMax$ semantics, r's output is interpreted as the derivation result of both aggregation and selection rules. If the shortest-path example, $r3$ and $r4$ are introduced simultaneously, and they are both considered output-contributing rules if the derivation results of $r4$ intersects with O.

All solutions of NetSpec contain only output-contributing rules. Because, during the rule extension phase in algorithm 1, a candidate program is discarded if the newly extended rule does not produce desired output.

Definition 4 (Program space). *Given a set of input tuples and a set of output tuples (I,O), and a set of user-defined functions,* *let P_c be the set of Datalog programs that contain only output-contributing rules (definition 3). The program space is defined as programs in P_c that are descendants of the empty program p_0:*

$$\{p \in P_c |\ p_0 \rightarrow^* p\} \tag{14}$$

Theorem 2 (Weak completeness)**.** *For all input-output tables (I,O), if there exists a program p in the program space (definition 4), such that p terminates on input I within a time bound T, with output O, then NetSpec always returns a solution set S, which contains at least one such program p. Otherwise, it returns an empty solution set $S = \emptyset$.*

This completeness property is "weak" because it assumes a smaller program space (only rules that derive output tuples, definition 4) than the full program space $\{p|p_0 \rightarrow^* p\}$. For example, in the routing protocol in Figure 2b, rule $r1$ generates one-hop paths. Given an input network where all best paths have more than one hop, $r1$'s output has no intersection with the desired output, and is thus discarded by Algorithm 1(step 2b), although it is part of the solution. Appendix A shows the proof sketch of Theorem 2.

5 HANDLING INCOMPLETE EXAMPLES

We now describe the example augmentation process. Given an initial set of input-output examples, when multiple satisfying programs are found, NetSpec searches for a new input example that can differentiate these candidate programs, and asks the user to specify the expected output for this new input example. By actively querying the user for feedback, this allows the system to robustly learn programs even from a set of initially under-specified examples.

Algorithm 4 Sample(). Samples new input tuples for disambiguation.

```
1) Initialize the set of input tuples $I := \emptyset$.
2) For each input relation $R$:
   a) Uniformly sample the number of tuples, $n \in \{1, 2, \ldots, n_{\max}\}$, where $n_{\max}$ is the upper bound on the size of the sampled tables.
   b) Sample $n$ tuples, $t_1$, $t_2$, ..., $t_n$, where each $t_i = (c_1, c_2, \ldots, c_k)$, all constants being uniformly sampled, and where $k$ is the arity of the relation $R$.
   c) Insert $t_1$, $t_2$, ..., $t_n$ into $I_R$.
3) Return $I$.
```

We describe the core example sampling process in Algorithm 4. In particular, in step 2a), n_{max} is the maximum of the table sizes in the initial example input I. In step 2b), constants are uniformly sampled from the set of all constants, that appear in the initial example input I.

Given a set of candidate programs P_1, P_2, ..., P_n which are consistent on the initial example input I (i.e., $P_1(I), P_2(I), ..., P_n(I)$ match the initial example output), we repeatedly run the sample procedure to obtain k new example inputs, I_1, I_2, ..., I_k. We then choose an example input $I_q \in \{I_1, I_2, \ldots, I_k\}$ for the user to label the corresponding example output, as follows:

$$I_q = \arg\max_{I_j}(-\sum_{O} p_O \log(p_O)), \tag{15}$$

where O ranges over the set of example outputs $\{P_1(I_j), P_2(I_j), \ldots, P_n(I_j)\}$, and p_O is the fraction of the candidate programs which produce O as output. By maximizing the entropy of the new example, we eliminate as many programs as possible after user feedback. We illustrate this using $n = 4$ candidate programs and $k = 3$ sampled examples $\{I_1, I_2, I_3\}$ [1] such that:

1) The programs are consistent on I_1. Then, O ranges over a singleton set of outputs and $p_O = 1$, so the score of I_1 is $-(1 \cdot 1 \cdot log(1)) = 0$.
2) The programs are 50-50 split on I_2. Then, O ranges over a set of two distinct outputs and $p_O = 0.5$, so the score of I_2 is $-(2 \cdot 0.5 \cdot log(0.5)) \sim 0.69$.
3) Each program produces a unique output on I_3. Then, O ranges over a set of four distinct outputs and $p_O = 0.25$, so the score of I_3 is $-(4 \cdot 0.25 \cdot log(0.25)) \sim 1.38$.

Thus, I_3 would be selected as the new example, which corresponds with the intuition that user feedback would eliminate the most (i.e. 3 out of 4) candidate programs.

We repeat this procedure until the remaining programs can no longer be distinguished by the sampled inputs. This approach is similar to the query-by-committee method [35] and enables NetSpec to rapidly converge to the final solution.

Theorem 3. *Assume NetSpec is always able to disambiguate candidate programs, and that the user always gives correct answers to NetSpec's queries, if there exists a solution p in the program space (Definition 4), then NetSpec always returns solutions that are logically equivalent to p after active-learning.*

Proof sketch. By theorem 2, p is always in the solution set S after every iteration of synthesis. By the assumption that NetSpec is always able to disambiguate candidate programs, a new queries will always be generated to differentiate p from other solutions, until all programs in S are logically equivalent. Therefore, when NetSpec terminates, all solutions are logically equivalent to p.

6 IMPLEMENTATION

NetSpec is implemented in Scala and comprises $\sim$ 3.5K lines of code. [2] It uses Souffle [38] as the backend Datalog interpreter to validate the candidate specifications. In this section, we discuss implementation details regarding how NetSpec handles non-terminating candidate specifications, and how it synthesizes specifications with constants.

Handling non-terminating specifications. During the synthesis process, NetSpec could encounter non-terminating specifications in the presence of recursion and user-defined functions. For example, consider the following candidate which NetSpec encounters in the process of synthesizing the routing example in Section 2.1:

```
// compute available paths
r1: path(x,y,p,c) :- link(x,y,c), p=[x,y].
r2: path(x,y,x::p1,c1+c2) :- link(x,z,c1),
                             path(z,y,p1,c2).
```

---

[1] Concrete examples: https://github.com/HaoxianChen/netspec/blob/master/docs/active-learning-example.md

[2] NetSpec is available at: https://github.com/HaoxianChen/netspec

Category	Specification	Features			# Relations		Examples		Synthesis time (s)			Output size (#rules)		
		recur.	aggr.	UDF	In	Out	#Inst.	#Rows	NS.	Fa.	GS.	NS.	Fa.	GS.
Network analysis	reachable	✓			1	1	1	42	17	1	11	2	2	2
	path	✓		✓	1	1	1	15	23			2		
	path-cost	✓		✓	1	1	3	17	92			2		
	publicIP				4	1	1	17	5	5	12	1	1	1
	sshTunnel	✓			1	1	1	25	11	1	8	2	2	3
	locality				3	1	5	21	2	x	14	1		1
SDN	learning-switch				2	3	4	15	6	x	x	3		
	stateless-firewall				3	2	2	9	2	1	66	3	3	3
	firewall-l3				6	3	14	76	12	TO	199	5		6
Consensus protocol	paxos-acceptor				3	3	5	14	7			4		
	paxos-quorum		✓	✓	2	1	2	19	36			1		
	paxos-maxballot		✓		2	1	1	8	3			3		
	paxos-decide				2	1	2	5	1			1		
Routing protocol	shortest-path	✓	✓	✓	1	1	1	10	14			4		
	least-congestion	✓	✓	✓	1	1	1	15	24			4		
	ospf		✓	✓	2	1	1	14	8			3		
	bgp		✓	✓	2	1	1	11	4			3		
	rip		✓	✓	3	1	1	18	180			4		
Sensor network	evidence				2	1	1	5	1	1	4	1	1	1
	store				2	1	1	7	1	1	6	1	1	1
Wireless routing	dsr-rrep			✓	2	1	2	5	2			2		
	dsr-rreq			✓	2	2	2	6	1			1		
	dsr-rerr			✓	3	1	1	6	1			1		

TABLE 2: Synthesis results for benchmarks where the original examples are sufficient. For expressiveness, specifications that use the features of recursion, aggregation, and UDFs are highlighted in the "Features" columns. The "#Relations" shows the number of input and output relations for each specification. The effort of specifying examples is described by the number of input-output instances and the total number of rows in all instances. Column "Time" shows the synthesis time of each tool, measured in seconds. Column "Output size" shows output size of each tool, measured in lines of Datalog rules. Both synthesis time and output sizes are average across 10 runs. NS., Fa., and GS. stand for NetSpec, Facon and GenSynth respectively. In the "Time" column, × means the tool terminates and finds no solution, and TO means the tool times out after 20 minutes. For benchmarks where the tool is inapplicable, the time and size entries are left empty.

Category	Specification	Features			# Relations		# Examples		# Queries		Time (s)	Output size (#rules)
		recur.	aggr.	UDF	In	Out	#Inst.	#Rows	med.	max		
Network analysis	subnet				4	1	7	27	3	6	43	1
	sshTunnel	✓			1	1	1	25	2	4	243	2
	protection				3	1	2	19	3	11	45	1
	locality				3	1	5	21	9.5	16	154	1
SDN	learning-switch				2	3	4	15	6.5	10	65	3
	l2-pairs				2	3	6	23	7.5	10	90	4
	stateful-firewall				5	3	15	78	18.5	26	367	5
	firewall-l3-stateful				5	3	13	68	12.5	15	189	4
Consensus protocol	2pc		✓		5	2	8	129	30	48	TO	2
	acceptor				3	3	5	14	19.5	26	430	4
	proposer				4	1	7	26	31.5	35	723	2
Routing protocol	ospf		✓	✓	2	1	1	14	5.5	9	2,736	3
	bgp		✓	✓	2	1	1	11	6	10	2,945	3
	tree		✓	✓	1	1	1	15	2	3	1,841	4
	min-admin		✓	✓	2	1	1	8	6	8	1,591	3
	rip		✓	✓	3	1	1	18	4	8	TO	4
Wireless	dsdv			✓	4	1	6	23	17.5	28	TO	4
Sensor	temperature-report				6	2	10	34	42.5	52	474	2

TABLE 3: Active learning results for benchmarks that needs example augmentation. NetSpec runs active learning to augment the input-output examples and finds the validated solutions. In the "#Queries" column, "med." and "max" stand for median and maximum. Column "Time" shows the average end-to-end time. "TO" means timing out after 1 hour. Column "Output size" reports the size of the synthesized specification, measured in the number of Datalog rules.

This specification does not terminate when the input network topology represented by the `link` relation contains a cycle, since both `::` (list prepend) and `+` (integer addition) used in the recursive rule `r2` generate new values every time the rule is evaluated. NetSpec handles such specifications by halting the specification interpreter after a timeout period, and considers their output to be empty.

Generating constants in specifications. Many network specifications in practice require constants. For example, in an SDN firewall specification, the controller application monitors and responds only to a certain port. Such a specification cannot be synthesized without the use of constants. On the other hand, naively adding constants into the specification can lead to over-fitting it to the provided input-output examples.

To distinguish specifications where constants are fundamentally needed from those which can be realized symbolically, NetSpec employs a fail-over mechanism: it embarks by only searching for symbolic specifications; when it exhausts the candidate program queue and fails to find a solution, it switches to use constants from the input-output examples. In experiments where NetSpec learns specifications from execution traces (Section 7.3), all firewall applications monitor certain ports on a dedicated switch. On their traces, the fail-over mechanism is triggered and NetSpec synthesizes specifications with constants on the switch and port field.

7 EVALUATION

Our evaluation aims to answer the following four questions:

1) **Expressivity.** Is NetSpec able to synthesize a wide range of network specifications correctly, and how does its coverage compare to state-of-the-art synthesis tools?
2) **Efficiency.** Can NetSpec synthesize a network specification in reasonable time (on the order of seconds)?
3) **Robustness.** Is NetSpec robust to input-output example quality, in particular, can it handle incomplete examples?
4) **Scalability.** Can NetSpec learn specifications from examples derived from a large volume of execution traces? Note that this question goes beyond performance to also capture NetSpec's ability to debloat legacy applications.

Benchmarks. [3] We survey the use of declarative specifications from literature, and organize them into five categories: network analysis, SDN, sensor networks, consensus protocols, and routing protocols. Network analysis refers to prior work on formalizing reachability and other correctness properties in networks [4], [26], [31]. SDN specifications are from works on verifying correctness of controller programs [5], [31]. Sensor network specifications are based on a declarative sensor network system [14]. Consensus protocols [3] and distributed routing are based on declarative specifications targeted for distributed execution [25], [14], and verification [20], [17], [41].

Input-Output example generation. To provide examples free of bias to any synthesizer, we manually read through the documentations for each benchmark protocol, and come up with input-output examples that cover all the use scenarios described in the documentations. The example size is measured by the number of input-output example instances (i.e. groups of input-output tables), and the number of total tuples (i.e., number of rows in relational tables) in all instances, as shown in the "#Examples" column in Table 2.

Result validation. A synthesis result is correct if it is identical to the reference specification after two modifications: (1) variable renaming, and (2) removing redundant predicates and rules (if any). We manually validate all experiment results. Reference [1] illustrates how each benchmark is validated, and has synthesis results of all experiments. Modification (1) dominates the validation process and a few results require modification (2). In the remainder of this section, we refer to such results as *validated solutions*.

The rest of the section are structured as follows. In Section 7.1, we evaluate NetSpec's expressivity and efficiency by comparing with state-of-the-art program synthesis tools. In Section 7.2, we evaluate NetSpec's robustness to input-output example quality, on benchmarks with insufficient examples. In Section 7.3, we evaluate NetSpec's scalability on execution traces that consist of thousands of examples.

7.1 Synthesis Expressivity and Efficiency

We first evaluate expressivity and synthesis efficiency given sufficient examples. We compare NetSpec to two state-of-the-art tools, Facon [13] and GenSynth [28].

Applicable benchmarks. Like NetSpec, both Facon and GenSynth operate on relational input-output data. However, they are less expressive: neither tools support UDFs and aggregation. Therefore, we only run these tools on benchmarks that they apply to. In addition, some of the original benchmark specifications may have insufficient examples, i.e. missing corner cases and resulting in incorrect specifications. To evaluate synthesis efficiency and compare with baselines that do not augment examples, this section focuses on benchmarks with sufficient examples for this experiment (Table 2), where NetSpec returns validated solutions for at least 8 out of 10 repeated runs. We will revisit applications with insufficient examples in Section 7.2.

In addition, GenSynth does not support multiple instances of examples. We therefore combine the multiple instances into a single instance by unioning tuples that belong to the same relation into the same table. We avoid introducing spurious correlations across the original instances by renaming constants appropriately. Note that this process is challenging to automate since certain constants (e.g. port numbers) are global and must not be renamed. This highlights the benefit of supporting multiple example instances as well as constants in NetSpec.

Performance metric. We measure NetSpec's expressivity in terms of coverage of different network specifications from the areas of network analysis, SDN, sensor networks, consensus protocols, and routing protocols. To evaluate synthesis efficiency, we measure the end-to-end synthesis time, on a server with 32 2.6GHz cores and 125GB memory. Both NetSpec and Facon run in single thread. GenSynth, however, often runs indefinitely long in single thread, due to its high degree of nondeterminism. Therefore, we run GenSynth in 8-thread mode in order to obtain results within 20 minutes each.

Results. Table 2 summarizes our overall results. Focusing on the first two criteria of expressivity and efficiency, our main takeaways are as follows:

Expressivity. NetSpec successfully synthesizes all 23 benchmarks in Table 2, spanning different types of network protocols. On the other hand, due to limited language feature support, competing solutions such as Facon [13] and GenSynth [28] support only 9 benchmarks. In addition, Facon fails to synthesize the locality benchmark because it lies outside of Facon's program search space, which only contains Datalog rules where each relation appears at most once. Both Facon and GenSynth fails to synthesize learning-switch benchmark due to the lack of support for negation.

Efficiency. NetSpec is highly efficient. NetSpec finishes most benchmarks within one minute, with the exception of path-cost and the RIP protocols, which takes 92 seconds and 3 minutes respectively. On the other hand, Facon is only able to synthesize 8 out of 10 applicable benchmarks, and in

---

[3] The full list of benchmarks: https://github.com/HaoxianChen/netspec/tree/master/benchmarks

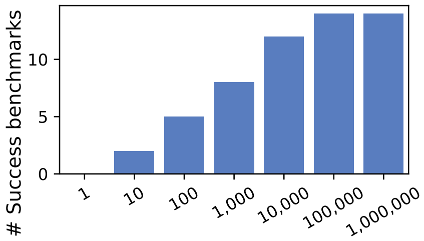


Fig. 6: NetSpec solves more benchmarks as the number of random samples in active learning increases.

fact, two benchmarks timed out after 20 minutes. Compared to GenSynth, NetSpec consistently finishes faster on all applicable benchmarks, except $reachable$ and $sshTunnel$, where NetSpec takes 6 seconds and 3 seconds longer respectively. This is impressive since NetSpec runs in single thread whereas GenSynth in 8.

Benefits of component-based synthesis. Our benchmarks also showcase the benefits of synthesis in a component-based fashion. We describe case studies based on two protocols: PAXOS (paxos-* in the Table 2) and DSR (dsr-*) in Table 2. The original specification of PAXOS lies contains two layers of aggregations (first count the votes to determine which ballot has a quorum, and then decide a value by choosing the one with the maximum ballot value). In normal circumstances, this is beyond NetSpec's search capabilities since it can only synthesize programs with at most one layer of aggregations. However, by breaking PAXOS into different components, synthesis is not only possible but done efficiently.

DSR, on the other hand, can be synthesized as one monolithic protocol. Yet, by breaking its synthesis into component modules, it significantly reduce the number of examples to sufficiently specify the protocol. DSR handles three different kinds of input messages independently, thus it gives the opportunity to break down the synthesis task into independent modules. For example, when synthesizing a rule to process route request message, the synthesizer does not need to consider any input value of a route error message. On the other hand, if examples for all types of message handling are combined together, although NetSpec can still efficiently find a solution, but it will generate a lot of invalid solutions (consistent with input-output examples but not equivalent to the reference solution) due to the larger program space. We note that component-based synthesis strategy for DSR is not only complete, but also highly efficient.

7.2 Robustness to Insufficient Examples

We evaluate NetSpec's robustness to insufficient examples on two kinds of benchmarks: (1) benchmarks with insufficient examples (Table 3); (2) benchmarks with sufficient original examples, but some of the examples are randomly dropped to test NetSpec's limit (Figure 7).

Handling insufficient examples. For each benchmark in Table 3, we run NetSpec with active learning, which iteratively queries the user with extra input examples, until it finds no ambiguities in the examples.

To determine the number of random samples in active learning phase (Section 5), we gradually increase the sample number from one to a million, and measure the number of benchmarks solved by NetSpec. Due to the randomness of the active learning algorithm, a benchmark is determined successful if NetSpec returns validated solutions in all 10 repeated experiment runs.

Figure 6 shows the results, where NetSpec's performance saturates at 100K samples, with 15 out of 18 benchmarks succeeding. The remaining three benchmarks involve the most complex specifications. They timed out and return incorrect specifications (consistent with input-output examples but different from the reference). The time bound is introduced because NetSpec is designed for interactive use. Recall that the active-learning phase involves multiple iterations of specification synthesis and new input example generation, whose output is annotated by protocol designers. Since increasing the sampling parameter beyond 100K does not reduce the end-to-end time (i.e. does not helping to solve more benchmarks within the time budget), we use 100K as the number of random samples for our remaining experiments, and the default value for this parameter. Users could also determine this parameter for their problem domains using the same experiment procedure.

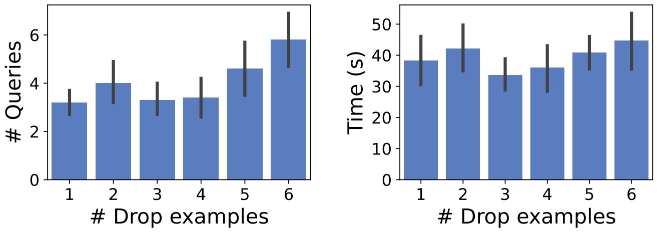
Table 3 shows the detailed statistics of the active learning experiments. The number of queries varies across different benchmarks, with the median ranging from 2 to 42.5. Similarly, the end-to-end time ranges from 43 seconds to 2,945 seconds across benchmarks.

For the three benchmarks (2pc, rip, and dsdv) that timed out, their relations compose a much larger program space (rules with many predicates and aggregators), and thus more examples are needed to unambiguously specify a program. This results in too many iterations in active learning, which is a limitation of input-output example based interface. Synthesizing correct specifications for them require either reducing the number of queries or improving synthesis efficiency, which remains an interesting avenue of future work.

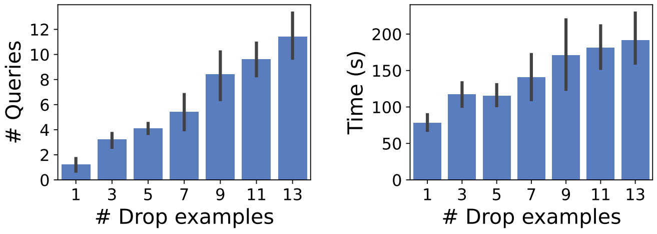
Randomly omitted examples. We further stress test NetSpec by randomly dropping examples. Three benchmarks with at least seven examples are chosen for this experiment. For each of them, examples are dropped incrementally until reaching the most extreme case, where every output relation appears in only one example instance. Otherwise, an output relation is missed from all examples, and NetSpec would skip synthesizing rules for the relation, thus returning an incomplete program.

Figure 7 presents the distributions of the number of queries and the end-to-end active learning times for each benchmark across ten repeated runs. The number of queries shows positive correlation with the number of dropped examples with one exception. Benchmark "temperature-report" shows weaker correlation because each active learning run takes more queries (40 ± 5) than the original example set size (9). Hence, the impact of dropping examples is weaker than benchmarks where the overall number of queries are smaller.

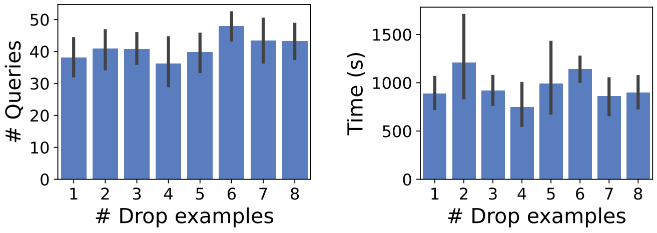
For relationship between end-to-end time and the number of dropped examples, "firewall-l3" shows strong positive correlation. The "temperature-report" shows no such



(a) subnet queries (b) subnet time



(c) firewall-l3 queries (d) firewall-l3 time



(e) temperature report queries (f) temperature report time

Fig. 7: Randomly drop examples.

correlation, which is expected since its query numbers is not correlated with the number of dropped examples (Figure 7e). On the other hand, for the "subnet" benchmark, although the number of dropped examples has strong correlation with the number of queries (Figure 7a), the correlation with time (Figure 7b) is weaker. This is because the end-to-end time is dominated by the synthesis time of the final few runs (where examples are almost sufficient). The earlier runs are fast because, with just a few examples left, NetSpec can quickly find superficial solutions and query new examples. When examples are sufficient, the solution becomes much more complex (more literals in a rule). This complexity, coupled with the randomness of the synthesis algorithm, leads to larger variation in synthesis time.

In the "subnet" and "temperate-report" benchmarks, NetSpec recovers validated specifications consistently (100%), even under extreme cases where only 1 is example is left. "Firewall-l3", on the other hand, fails when examples for particular message types are all dropped. For instance, if no example responds to ARP packets, then all candidate programs would just ignore ARP packets, because NetSpec discard rules that derive no reference output, although the reference program actually handles ARP packets. In practice, however, it is rare that a protocol designer provides no examples for a typical type of output at all.

Overall, NetSpec's active learning mechanism is effective in improving example quality and finding validated solutions. General differential testing of programs is a hard problem. However, by separating protocol logic from implementation details, declarative specifications drastically reduce the search space to differentiate alternative specifications. By exploiting the simplicity of declarative specifications, NetSpec's simple random testing mechanism is able to effectively disambiguate alternative protocol specifications.

Program	LOC	#T	#R	#Queries		Time
				med.	max	(s)
Floodlight stateless firewall	216	895	5	8	9	188
Floodlight stateful firewall	233	121	7	23	25	468
POX l3 stateless firewall	97	185	5	10	12	79
POX l3 stateful firewall	107	4,591	6	22.5	26	505
POX learning-switch	98	334	4	6	10	2,295

TABLE 4: Learning specifications from program communication traces. Column "#T" measures the number of input output messages in the execution trace, column "#R" measures the number of rules of the reference specification, column "#Queries" measure the median and maximum number of queries posted by NetSpec, across 10 repeated experiments, and column "Time" shows the average end-to-end active learning time.

7.3 Learning from Program Traces

In our final experiment, we explore NetSpec's ability to directly synthesize from actual execution traces as input-output examples. The benefits of this approach are two-fold. First, for code refactoring or program analysis, the generated specifications expose the essential logic of the program, and can serve as a formal model for further analysis. Second, the logic specifications can be compiled into a more compact and less bloated program for execution.

Trace collection. We collect program communication traces from two popular SDN platforms, POX [32] and Floodlight [19], on which we run controller programs and collect its communication traces with the switches in the network. We select SDN platforms as a basis for this experiment due to readily available open-source implementations that match our benchmarks.

To generate input-output examples, we generate representative traffic loads that we inject into each SDN controller program. Based on the inputs, we capture the outputs by observing the SDN programs. For instance, for learning switches, all hosts send probe packets to establish full connectivity in the network. For firewalls, we divide the network into two zones, one protected by the firewall, and the other serving as the external network. We then randomly pick hosts from either side of the network to establish TCP sessions. We validate the functionality of the firewall by checking that only sessions initiated from internal hosts are successfully established.

Trace collection is done by running the controller programs in both POX and Floodlight on the Mininet [29] emulator. All Mininet topologies are setup on a 16-node, 8-switch, tree topology network. For each run, we collect the controller's input-output traces as it interacts with Mininet switches (via incoming/outgoing packets and flow modifications). In all our experiments, we observe that this setup suffices to collect enough examples for NetSpec to learn a validated specification, with additional queries to user.

We implemented a trace collector on both POX and Floodlight that collects the controller's input and output messages at run-time and the state changes to the program. The state monitor works as follows. Within each application's input packet handler, we inspect all the accessible global variables. We then record any changes to such global variables. We exclude the known global constructs that are irrelevant to each application's execution logic, like loggers.

Table 4 summarizes the results. We make the following observations. First, NetSpec is able to correctly synthesize the intended specifications for all applications. Second, even though traces contain up to 4,500 communication messages, additional queries are needed to augment the examples. This observation shows the practicability of NetSpec's example augmentation mechanism in helping user uncover corner cases. Third, even for non-trivial SDN applications with hundreds of lines of code, NetSpec is able to generate compact specifications with less than seven rules. Finally, the synthesis times are in the order of hundreds of seconds, except "learning-switch" at 2,295 seconds, despite the need to analyze actual communication traces with up to 4591 examples, and generate multiple queries, indicating the efficiency and scalability of our approach.

8 RELATED WORK

Programming by example. NetEgg [44] enables programming SDN policies by example timing diagrams. NetEgg demonstrates via actual user studies that a programming-by-example paradigm can result in higher programming productivity and fewer errors. The key distinction is that NetSpec synthesizes the actual control plane program in the target DSL, which generates the data plane configurations, whereas NetEgg directly generates the data plane configurations. This target DSL can be used to verify and check for errors in the control plane program, whereas NetEgg can only provide counter-examples to indicate that the input examples are incorrect. NetSpec mitigates one inherent weakness of NetEgg in its reliance on the user to provide all possible examples that meet the scenarios. Facon [13] is a programming-by-example tool for synthesizing SDN programs. NetSpec employs a more scalable synthesis strategy, targets a more general logical model, and can handle more complex protocols and incomplete examples.

Network configuration synthesis. Taking high-level routing policies as input, NetComplete [17] synthesizes BGP configurations that comply with these policies, and Genesis [39] synthesizes forwarding tables in multi-tenant networks. Avenir [10] synthesizes SDN data plane operations from high-level forwarding specifications. Propane [7] compiles high-level routing policies into distributed router BGP configurations. In contrast, NetSpec uses input-output examples, or execution traces from legacy programs, as input, and generate executable protocol specifications that are consistent with given input-output examples.

Config2Spec [8] takes network configurations and a failure model as input, and generates network policies that should hold for all possible concrete data planes derived from the given configurations and failure model. On the other hand, NetSpec generates executable specifications (analysis rules), instead of the static policies (facts derived from analysis rules). NetSpec can complement Config2Spec when analysis tools for the interested policies are not available. NetSpec takes the concrete data plane and the set of satisfied policies as input, and generates data plane analysis rules that can be applied to all concrete data planes. For example, in section 7.1, we show that NetSpec can generate reachability analysis rules similar to what is used by Config2Spec when inferring reachability policies.

Datalog and logic program synthesis. A large body of work has proposed techniques to synthesize logic programs [15] from input-output examples. With the exception of GenSynth, existing techniques require the user to syntactically constrain the search space by means of specifications such as mode declarations (e.g. ILASP [24]), meta-rules (e.g. Metagol [16]), candidate rules (e.g. ALPS [36] and ProSynth [33]), or templates (e.g. NTP [34] and δILP [18]). NetSpec does not require the user to provide any such specifications, but with the trade-off to require more input-output examples to fully specify an intended program. However, with the help of active-learning, as our evaluation demonstrates, NetSpec synthesizes more general programs than GenSynth, and is more efficient and robust.

Network verification and domain-specific languages. There is significant prior work on network verification [22], [20], [6], [40], [41] and DSLs for networking [31], [21], [23], [5]. NetSpec generates a logical network specification that can be verified using existing techniques. Hence, verification should be viewed as a complementary technology to NetSpec. The same benefits of having a restricted language, such as scalable synthesis and automated example augmentation, apply to other DSLs as well.

9 CONCLUSION

NetSpec addresses a long-standing problem in network verification: the widening gap between formal models and actual implementations. As a step towards closing the gap, we have proposed a new *specification by example* (SBE) toolkit where users can build formal models of their network protocols from input-output examples either supplied by the network designer or extracted from a legacy implementation. Our synthesized models are declarative logic programs which are amenable to formal verification and even generation of distributed implementations.

Our initial forays and experimental results are promising. The SBE approach can efficiently synthesize a wide range of network protocols, and is robust to missing examples. NetSpec should be viewed as a first step towards understanding the SBE paradigm and its application in different domains of networking, with limitations in the size of synthesized specifications and complexities. In the future, we plan to explore how to synthesize more complex specifications, methods for parallelizing the synthesis algorithms to handle larger specifications, and how SBE can interact with different formal verification techniques.

10 ACKNOWLEDGEMENT

We thank the anonymous reviewers for their insightful feedback. This work is supported by NSF under Grant CCF-2107429, CCF-2107261, CCF-2124431, CNS-2104882, CNS-2107147, and Office of Naval Research grant on Security for resource limited networked Cyber-physical systems.

REFERENCES

[1] Netspec synthesis result validation. https://github.com/HaoxianChen/netspec/blob/master/synthesis-validation.

[2] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases: The Logical Level*. Pearson, 1st edition, 1994.

[3] Peter Alvaro, Tyson Condie, Neil Conway, Joseph M Hellerstein, and Russell Sears. I do declare: Consensus in a logic language. *ACM SIGOPS Operating Systems Review*, 2010.

[4] John Backes, Sam Bayless, Byron Cook, Catherine Dodge, Andrew Gacek, Alan J Hu, Temesghen Kahsai, Bill Kocik, Evgenii Kotelnikov, Jure Kukovec, et al. Reachability analysis for aws-based networks. In *International Conference on Computer Aided Verification*, 2019.

[5] 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.

[6] Ryan Beckett, Aarti Gupta, Ratul Mahajan, and David Walker. A general approach to network configuration verification. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, 2017.

[7] Ryan Beckett, Ratul Mahajan, Todd Millstein, Jitendra Padhye, and David Walker. Don't mind the gap: Bridging network-wide objectives and device-level configurations. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[8] Rüdiger Birkner, Dana Drachsler-Cohen, Laurent Vanbever, and Martin Vechev. Config2spec: Mining network specifications from network configurations. In *17th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 20)*, 2020.

[9] Nikolaj Bjørner and Karthick Jayaraman. Checking cloud contracts in microsoft azure. In *International Conference on Distributed Computing and Internet Technology*, 2015.

[10] Eric Hayden Campbell, William T Hallahan, Priya Srikumar, Carmelo Cascone, Jed Liu, Vignesh Ramamurthy, Hossein Hojjat, Ruzica Piskac, Robert Soulé, and Nate Foster. Avenir: Managing data plane diversity with control plane synthesis. In *NSDI*, 2021.

[11] Ang Chen, Yang Wu, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. The good, the bad, and the differences: Better network diagnostics with differential provenance. In *Proceedings of the 2016 ACM SIGCOMM Conference*, 2016.

[12] Chen Chen, Limin Jia, Hao Xu, Cheng Luo, Wenchao Zhou, and Boon Thau Loo. A program logic for verifying secure routing protocols. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, 2014.

[13] Haoxian Chen, Anduo Wang, and Boon Thau Loo. Towards example-guided network synthesis. In *Proceedings of the 2nd Asia-Pacific Workshop on Networking*, 2018.

[14] David Chu, Lucian Popa, Arsalan Tavakoli, Joseph M Hellerstein, Philip Levis, Scott Shenker, and Ion Stoica. The design and implementation of a declarative sensor network system. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, 2007.

[15] Andrew Cropper, Sebastijan Dumancic, and Stephen H. Muggleton. Turning 30: New ideas in inductive logic programming. In *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, 2020.

[16] Andrew Cropper and Stephen H. Muggleton. Metagol system. https://github.com/metagol/metagol, 2016.

[17] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. Netcomplete: Practical network-wide configuration synthesis with autocompletion. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*, 2018.

[18] Richard Evans and Edward Grefenstette. Learning explanatory rules from noisy data. *Journal of Artificial Intelligence Research*, 61, 2018.

[19] Floodlight. 2020. http://www.projectfloodlight.org/floodlight/.

[20] Ari Fogel, Stanley Fung, Luis Pedrosa, Meg Walraed-Sullivan, Ramesh Govindan, Ratul Mahajan, and Todd Millstein. A general approach to network configuration analysis. In *12th {USENIX} symposium on networked systems design and implementation ({NSDI} 15)*, 2015.

[21] Nate Foster, Rob Harrison, Michael J Freedman, Christopher Monsanto, Jennifer Rexford, Alec Story, and David Walker. Frenetic: A network programming language. *ACM Sigplan Notices*, (9), 2011.

[22] Ahmed Khurshid, Xuan Zou, Wenxuan Zhou, Matthew Caesar, and P Brighten Godfrey. Veriflow: Verifying network-wide invariants in real time. In *10th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 13)*, 2013.

[23] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russ Clark. Kinetic: Verifiable dynamic network control. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015.

[24] Mark Law, Alessandra Russo, and Krysia Broda. The ilasp system for inductive learning of answer set programs. *arXiv preprint arXiv:2005.00904*, 2020.

[25] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E Gay, Joseph M Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. Declarative networking. *Communications of the ACM*, 2009.

[26] Nuno P Lopes, Nikolaj Bjørner, Patrice Godefroid, Karthick Jayaraman, and George Varghese. Checking beliefs in dynamic networks. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, 2015.

[27] William M McKeeman. Differential testing for software. *Digital Technical Journal*, 1998.

[28] Jonathan Mendelson, Aaditya Naik, Mukund Ragothaman, and Mayur Naik. Gensynth: Synthesizing datalog programs without language bias. *Proceedings of the Conference on Artificial Intelligence (AAAI)*, 2021.

[29] Mininet. http://mininet.org/.

[30] Stephen Muggleton and Luc De Raedt. Inductive logic programming: Theory and methods. *The Journal of Logic Programming*, 19, 1994.

[31] Tim Nelson, Andrew D Ferguson, Michael JG Scheer, and Shriram Krishnamurthi. Tierless programming and reasoning for software-defined networks. In *11th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 14)*, 2014.

[32] POX. 2020. https://github.com/noxrepo/pox.

[33] Mukund Raghothaman, Jonathan Mendelson, David Zhao, Mayur Naik, and Bernhard Scholz. Provenance-guided synthesis of datalog programs. *Proceedings of the ACM on Programming Languages*, 2020.

[34] Tim Rocktäschel and Sebastian Riedel. End-to-end differentiable proving. In *Proceedings of the Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[35] H. S. Seung, M. Opper, and H. Sompolinsky. Query by committee. In *Proceedings of the 5th Annual Workshop on Computational Learning Theory (COLT'92)*, 1992.

[36] Xujie Si, Woosuk Lee, Richard Zhang, Aws Albarghouthi, Paraschos Koutris, and Mayur Naik. Syntax-guided synthesis of Datalog programs. In *Proceedings of the Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018.

[37] Atul Singh, Tathagata Das, Petros Maniatis, Peter Druschel, and Timothy Roscoe. Bft protocols under fire. In *NSDI*, 2008.

[38] Souffle. 2020. https://souffle-lang.github.io/index.html.

[39] Kausik Subramanian, Loris D'Antoni, and Aditya Akella. Genesis: Synthesizing forwarding tables in multi-tenant networks. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, 2017.

[40] Anduo Wang, Prithwish Basu, Boon Thau Loo, and Oleg Sokolsky. Declarative network verification. In *International Symposium on Practical Aspects of Declarative Languages*. Springer.

[41] Anduo Wang, Limin Jia, Wenchao Zhou, Yiqing Ren, Boon Thau Loo, Jennifer Rexford, Vivek Nigam, Andre Scedrov, and Carolyn Talcott. Fsr: Formal analysis and implementation toolkit for safe interdomain routing. *IEEE/ACM Transactions on Networking*, 2012.

[42] Chenglong Wang, Alvin Cheung, and Rastislav Bodik. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2017.

[43] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. Automated network repair with meta provenance. In *Proceedings of the 14th ACM Workshop on Hot Topics in Networks*, 2015.

[44] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. Scenario-based programming for sdn policies. In *Proceedings of the 11th ACM Conference on Emerging Networking Experiments and Technologies*, 2015.

[45] Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao. Efficient querying and maintenance of network provenance at internet-scale. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 2010.

APPENDIX

We first present the following lemmas, and their proof sketches. With these lemmas, we then prove the completeness property, defined in Theorem 2.

Lemma 1. *Given a set of input tuples and a set of output tuples (I, O), if all rules in a program p are output contributing rules (Definition 3), then* $\mathrm{Score}(p) > 0$.

Let $\mathrm{OutCtrb}(p)$ *denotes that all rules in program p are output-contributing rules, the lemma is defined as:*

$$\mathrm{OutCtrb}(p) \implies \mathrm{Score}(p) > 0 \tag{16}$$

Proof sketch. By the semantics of Datalog, a program's output is the union of all rules' output. Thus $p(I) \cap O \neq \emptyset$. By the definition of Score, we have that $\mathrm{Score}(p) > 0$.

Lemma 2. *Given a set of input tuples and a set of output tuples (I, O), if all rules in a program p are output-contributing rules, then for every p's predecessors, all rules are also output-contributing rules:*

$$\forall q \to p, \mathrm{OutCtrb}(p) \implies \mathrm{OutCtrb}(q) \tag{17}$$

Proof. We enumerate all ways to generate offspring ($q \to p$):

1) If $p \in \mathrm{AddRule}(q)$ (equation 4), and let r_0 be the minimal rule added in p, we have $p = q \cup r_0$. Given $\mathrm{OutCtrb}(p)$, that is, all rules in p produces some desired output in O, and $p = q \cup r_0$, we have $\mathrm{OutCtrb}(q)$.
2) If $p \in \mathrm{ExtRule}(q)$ (equation 7), let r be the rule in q that have been extended as r' in p. Let $r(I)$ and $r'(I)$ denote the direct derivation output of r and r' on input I, respectively.
   a) Given $\mathrm{OutCtrb}(p)$, and that $r' \in p$, we have that $r'(I) \cap O \neq \emptyset$.
   b) By the semantics of Datalog, adding a predicate to a rule monotonically reduces the output of the rule. Thus we have $r'(I) \subseteq r(I)$.
   c) Given that $r'(I) \cap O \neq \emptyset$, and that $r'(I) \subseteq r(I)$, we have that $r(I) \cap O \neq \emptyset$.
   d) Given that all other rules in q are also in p, we have that $\mathrm{OutCtrb}(q)$.
3) If $p \in \mathrm{MkAgg(q)}$, given that NetSpec only introduces $argMax$ and $argMin$ aggregations, $p(I) \subseteq q(I)$.
   a) For rules $r \in q$ whose output are aggregated and renamed as $r' \in p$, because NetSpec introduces only $argMax$ and $argMin$ aggregations, $r'(I) \subseteq r(I)$.
   b) Following the same reasoning from 2c) to 2d), we have that $\mathrm{OutCtrb}(q)$.

□

Lemma 3. *Given a set of input tuples and a set of output tuples (I, O), if all rules in a program p are output-contributing rules, then there exists a lineage of programs $p_0 \to p_1 \to p_2 \to \ldots \to p_n \to p$, such that p_0 is the empty programs, and $p_1, \ldots, p_n$ contain only output-contributing rules:*

$$\begin{aligned}\mathrm{Lineage}(p) \coloneqq \mathrm{OutCtrb}(p) \implies& \\ \exists p_1, \ldots, p_n,\ &[(p_0 \to p_1 \to \ldots \to p_n \to p) \\ &\wedge \forall i \in \{1, \ldots, n\}, \mathrm{OutCtrb}(p_i)]\end{aligned}$$

$$\forall p, \mathrm{Lineage}(p) \tag{18}$$

Proof sketch. We prove by well-founded induction on the successor relation $\to$ on the program space (Definition 2). $\to$ is a well-founded relation on program space, because rules or literals cannot be taken away from a program indefinitely. To prove $\forall p, \mathrm{Lineage}(p)$ it is suffice to show that:

$$\forall p, [\forall q \to p, \mathrm{Lineage}(q)] \implies \mathrm{Lineage}(p) \tag{19}$$

If $\mathrm{OutCtrb(p)}$ is true, then by Lemma 2, we have that $\forall q \to p, \mathrm{OutCtrb}(q)$. By the antecedent of the induction hypothesis (equation 19), we have that there exists a lineage of programs $p_0 \to p_1 \to \ldots \to q$, and $\forall i \in 1, \ldots, n, \mathrm{OutCtrb}(p_i) \wedge \mathrm{Score}(p_i)$. Let $p_{n+1} = q$, and given that $q \to p$, we have that $\mathrm{Lineage}(p)$.

By well-founded induction, we have that $\forall p, \mathrm{Lineage}(p)$.

Lemma 4 (Termination)**.** *For all finite set of input tuples and output tuples (I,O), NetSpec always terminates.*

Proof sketch. We first show that the search space of NetSpec is finite. First, suppose there are N_R input relations, then according to the syntax constraints in Section 3, each rule contains at most $2N_R$ literals. Second, in offspring generation, we only add a rule to a candidate program if it has imperfect recall. In the worst case, each rule generates a tuple in O. Therefore, a program contains at most $|O|$ rules.

Let E_n be the set of all programs that have been popped from Q_n at the beginning of iteration n. Let P be the search space of NetSpec. We construct a function on iteration number n: $f(n) = |P| - |E_n|$. We then show that $f(n) >= 0$ and $f(n+1) < f(n)$. By the principle of well-founded induction, NetSpec terminates.

Proof sketch for weak completeness (Theorem 2). We first prove the case where valid solution exists. We prove by induction on iterations in Algorithm 1. We use subscript n to denote the state variable values at the beginning of the nth iteration, e.g., Q_n is the set of candidate programs at the beginning of iteration n.

Given a solution p, and by Lemma 3, we have that there exists a lineage of programs: $p_0 \to p_1 \to p_2 \to \ldots \to p_k \to p$, such that p_0 is the empty programs, and $p_1, \ldots, p_k$ contain only output-contributing rules.

Induction hypothesis: In every iteration, either p is in the solution set S_n, or one of p's ancestors in the lineage from p_0 to p is in the set of candidate programs Q_n:

$$\forall n, p \in S_n \vee (\exists i \in \{0, 1, 2, \ldots, k\}, p_i \in Q_n) \tag{20}$$

Base case: In iteration 0, by algorithm 1 step 1, Q is initialized with only the empty program p_0. Thus induction hypothesis holds.

Induction: Suppose in the nth iteration, the induction hypothesis holds, which implies either of the following:

1) If $p \in S_n$, by algorithm 1 step 2b, we have $S_n \subseteq S_{n+1}$. This implies that $p \in S_{n+1}$. Thus induction hypothesis holds in iteration $n + 1$.
2) Otherwise, $\exists i \in \{0, 1, 2, \ldots, k\}, p_i \in Q_n$. We discuss by two cases on the value of the current program P_n:
   a) If $P_n \neq p_i$, by step 2b, every program in Q_n is copied into Q_{n+1} except P, thus p_i remains in Q_{n+1}. Induction hypothesis holds in iteration n+1.
   b) Otherwise, $P_n = p_i$.

i) In step 2a, all offspring of p_i are generated. By the definition of the successor relation $\rightarrow$ (Definition 2), $p_{i+1} \in \mathrm{Offspring}(p_i)$.
ii) By Lemma 1 and Lemma 3, $\mathrm{Score}(p_{i+1}) > 0$.
iii) In step 2b, all offspring with score greater than 0 is added to Q_{n+1}. Given $\mathrm{Score}(p_{i+1}) > 0$, we have that $p_{i+1} \in Q_{n+1}$. Induction hypothesis holds in iteration n+1.

This induction hypothesis, in conjunction with the termination condition that $Q = \emptyset$, implies that $p \in S$ when NetSpec terminates.

For the second case, when no valid solution exists, by Theorem 1 (soundness) and Lemma 4 (termination), we have that NetSpec will terminate with $S = \emptyset$.