Towards Example-Guided Network Synthesis

Haoxian Chen
University of Pennsylvania
hxchen@seas.upenn.edu

Anduo Wang
Temple University
anduoadw@temple.edu

Boon Thau Loo
University of Pennsylvania
boonloo@seas.upenn.edu

Abstract

In recent years, there has been a proliferation in network domain-specific languages (DSL). These languages enable us to exploit the programmability of these networks, while still providing correctness guarantees through verification and analysis of DSLs. However, none of these DSLs have received widespread adoption. First these new languages require a learning curve among operators who may not be trained programmers. Second, these new SDN applications sometimes rely on functionality in legacy networks that cannot be easily migrated or analyzed.

To address these challenges, we propose Facon, a new tool that enables us to automatically generate programs in arbitrary DSLs, based on input/output examples. Since input/output examples applies to any network protocols, this approach can be generalized, hence enabling us to migrate legacy networks to new DSLs, or to transform one DSL to another. As an initial feasibility study, we apply Facon to a family of logic-based network DSLs based on declarative networking. Facon takes as examples input/output examples either provided by an operator or observed from a legacy system. Facon then automatically generates a declarative networking program faithful to these examples. We propose an efficient search algorithm that exploits syntactic constraints in declarative networking to prune the search space, and semantics as heuristics to guide the search direction. Our initial results are promising. Facon successfully synthesizes declarative networking programs at a scale beyond previous logic program synthesis tools can handle.

CCS Concepts

• **Networks → Programming interfaces**; • **Computing methodologies → Inductive logic learning**;

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
APNet '18, August 2–3, 2018, Beijing, China
© 2018 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-6395-2/18/08...$15.00
https://doi.org/10.1145/3232565.3234462

Keywords

Network Programming, Program Synthesis, Inductive Logic Programming

ACM Reference Format:
Haoxian Chen, Anduo Wang, and Boon Thau Loo. 2018. Towards Example-Guided Network Synthesis. In *APNet '18: 2nd Asia-Pacific Workshop on Networking, August 2–3, 2018, Beijing, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3232565.3234462

1.Introduction

The emergence of software-defined networking (SDN) has open up an exciting era for innovation in networking. To fully leverage these SDN platforms, many high-level domain-specific languages (DSL) have been proposed. These include logic-based languages such as declarative networking [11], FlowLog [15], functional languages such as Frenetic [14] and Pyretic [14], and state-machine based languages Kinetic [9]. By leveraging new language constructs such as functions [14], state machine [9], and graphs chains [16], those DSLs enabled many new capabilities not found in traditional networks ranging from automatic verification [4], composition [14], debugging [3], to consistent updates [18].

Despite these advantages, DSLs have not gained widespread adoption in practice. There are many potential reasons (e.g., performance concerns, limited expressiveness, etc.), but one major hurdle is the learning curve associated with new programming paradigm (syntax and semantics). Particularly, declarative DSLs, like FlowLog [15] and NDLog [11], would require significant changes in programmers' reasoning process, due to the semantic differences between imperative and declarative programming.

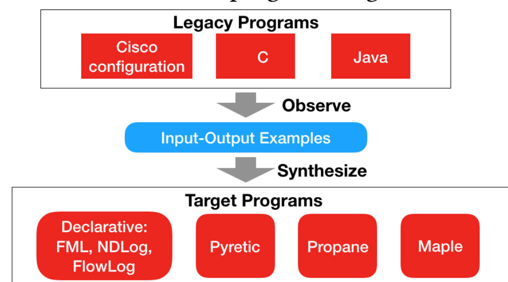

Figure 1: Migrate to new DSLs via program synthesis

To address this particular limitation, we propose a new approach to ease programming effort for network operators. Facon is a network DSL program synthesis tool that takes input-output examples, and generates a program that

satisfies these examples. Figure 1 highlights our approach. Rather than learning new DSLs and rewrite programs when migrating to a new platform, we automatically generate DSL programs by learning from input-output examples. These examples can come from either an operator or the I/O trace of an equivalent legacy system. Facon would then follow the syntax of the DSLs and generate a program that satisfies all given input-output examples.

Input-output examples are intuitive to human as programming interface. And they already exist for many applications: test cases. Test-driven development paradigm is well-adopted in software development practice, which provides a smooth transition path for network operators to adopt Facon.

Facon significantly improves upon NetEgg [23], a prior programming-by-example toolkit. Users draw timing diagrams of network scenarios, then NetEgg generates switch configurations that are consistent with the timing diagrams. While the user interfaces of Facon and NetEgg are similar, Facon generates the **program**, whose function is to generate data plane configurations, not the configurations themselves, as NetEgg does.

Generating the actual program is a superior approach in many ways. First, a program is portable. It can be moved from one network to another, even when the networks differ in the underlying execution model (e.g., moving from OpenFlow to a traditional network). Moreover, a synthesized program can be composed with other programs to implement more complex features. Second, our approach is more amenable to detecting errors in network configurations. While one can use tools like VeriFlow [8] to detect errors in network configurations, they do not fix the root cause, which is typically a bug in the program that generated the configurations. Finally, in enabling code generation from examples, Facon unlocks the possibility of debloating legacy codes, and transforming one DSL to another. These opportunities are not possible with an approach that only generates network configurations.

As the first step towards realizing our eventual vision for Facon, we perform a feasibility study of Facon, through a case study of Network Datalog (NDLog) [11], a representative declarative networking language. Facon is by no means limited to one DSL. We pick declarative networking for two reasons. First, it is a well-known DSL based on logic programming, with a rich set of advanced tools such as verification and debugging. Second, the synthesized logic model can be easily transferred to other DSLs within logic-based DSL family, e.g., FLog [7], FlowLog [15], and NLog [10]. Once the logic has been figured out correctly, the syntax translations among these declarative languages are straightforward.

In addition, declarative networking programs are known to be highly concise (e.g., Chord protocol implemented in 47 lines [12]). While declarative networking programs are not always easy for a human operator to understand, it makes it easier for program synthesizer to scale, because the compactness of the target program reduces the search space.

We summarize our key contributions as follows: (1) we propose a new approach to lower the bar for adopting new DSLs, using program synthesis, (2) we present a scalable algorithm to synthesize declarative networking programs using input-output examples, and finally, (3) we have implemented a prototype of Facon, we show that Facon can produce declarative networking programs within affordable time budgets, e.g., a stateful firewall program with 4 relations and 11 variables are generated within 72 seconds, whereas an SMT-solver on the same scale input would end-up timed-out.

Moving forward, we envision that our example-guided synthesis techniques can be generalized to other DSLs within the class of logic-based programming languages. By adding the DSL syntax to the logic models, and generating a final program that is consistent with given syntax. Facon is a step towards our long-term vision of synthesizing any network DSLs from input-output examples.

2.Overview

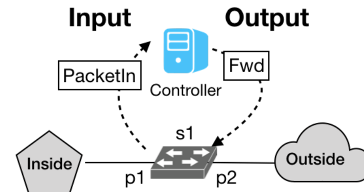

Figure 2: Example network with Stateful Firewall at edge

We provide an overview of Facon by an end-to-end example: synthesizing an NDLog program from input-output examples. Figure 2 depicts an SDN network where a controller manages a switch $s1$ sitting between the internal network and the external global Internet. Our target is to synthesize a **controller program** that implements a stateful firewall.

An SDN switch sends a request to the controller if any incoming packet fails to match its local forwarding table. Our target controller program, upon receiving the request should make a forwarding decision and push the corresponding forwarding rules back to the switch via a reply message. The stateful firewall should block external traffic unless it is returning traffic answering some requests originated from inside the network earlier. In Figure 2's example, traffic from port $p1$ to port $p2$ is allowed. But traffic from $p2$ to $p1$ is blocked, unless it's some returning traffic.

2.1 Input-Output Examples

The input to our target program is the request message from $s1$ (*PacketIn* in Figure 2), and the output is a replying message (*Fwd*). $PacketIn(s1, srcIp, dstIp, p1)$ means a switch $s1$ receives a packet from port $p1$, and the packet is sent from $srcIp$ to $dstIp$. $Fwd(s1, dstIp, p2)$ means a forwarding rule

$(dstIp, p2)$ is sent to switch $s1$, which instructs $s1$ to send a packet through $p2$ if its destination is $dstIp$. In practice, most SDN network follows standard protocols, e.g., OpenFlow [13]. We can obtain the message formats from these protocol specifications. Note that we simplify the message formats for ease of exposition.

Input-Output Examples are given as a set of tuples, each of which is the input or output messages for the controller program. In figure 2, upon receiving a packet from port $p1$ via $PacketIn(s1, ip1, ip2, p1)$. Controller then installs forwarding rules for traffic (both direction) by replying to $s1$ with message $Fwd(s1, ip2, p2)$ and $Fwd(s1, ip1, p1)$. Thus here the example would be:

$$Input : \{PacketIn(s1, ip1, ip2, p1)\}$$
$$Output : \{Fwd(s1, ip2, p2), Fwd(s1, ip1, p1)\}$$

2.2 Background on NDLog

In this paper, we focus on synthesizing NDLog programs, which is a dialect of the Datalog language, commonly used in the database community for querying graphs. The language is based on logic, and we provide some preliminaries on logic programming below.

Clause is a disjunction of literals (e.g., $a \vee b \vee c$).

Horn Clause is a clause that has at most one positive literal. It can be written in implication form, which is often used to specify logical rules in logic programming. For example, $\neg a \vee \neg b \vee c \equiv c \leftarrow a \wedge b$.

Relation is the property that applies to either literals or variables. For clarity, in the following we use capital letters to denote variables, and non-capital letters to denote literals. For example, a network link between switch a and b can be interpreted as $link(a, b)$.

Ground Relation is a relation whose subjects are literals.

Given the above definitions, we now define the semantics of an NDLog program. An NDLog program consists of a set of NDLog rules. On input, as a set of ground relations, each rule within the program is evaluated independently. Therefore, the output of a NDLog program P on input I, is the union of the output of each rule $R \in P$ on I: $O = \cup_{R \in P}(R \wedge I)$. Program execution can be written in a simpler form as $P \wedge I \vDash O$.

A NDLog rule is a Horn Clause whose literals are all relations, and these relations' subjects are all variables. For example, $path(X, Y) : -link(X, Y)$ is an NDLog rule, where the literals are relation $path$ and $link$, and they apply to variables X and Y. We call the relation on the left ($path$) as *rule head*, and the rest of the relations as a whole, *rule body*.

Semantics of an NDLog rule: the rule head relation is derived, if there exists both a ground relation, and an assignment to the variables, such that the rule body is true. For example, given ground relation $link(a, b)$, rule $path(X, Y) : -link(X, Y)$ derives $path(a, b)$. Because the assignment $\{X = a, Y = b\}$ makes the rule body evaluate to true.

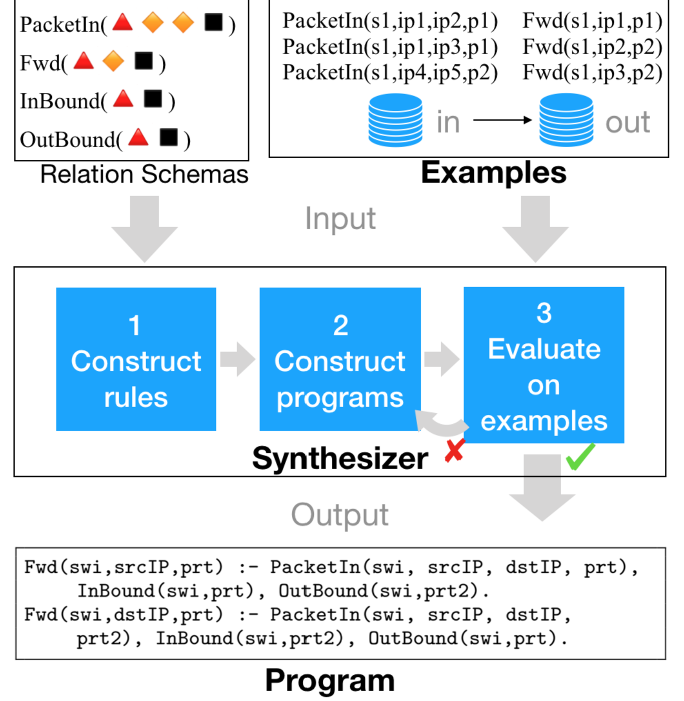

Figure 3: Facon workflow

2.3 Synthesis Workflow

Facon takes two kinds of inputs: (1) input-output examples; (2) NDLog relation schemas. Then it produces an NDLog program that satisfies all input-output examples. Figure 3 illustrates our synthesis workflow in three steps. Note that this "firewall" example is simplified: only the rule installation logic is shown. We omit the rule removal and update logics for the ease of illustration.

Step 1: Construct Rules. Facon first constructs all valid NDLog rules, using relations and variables as the basic building blocks. Top left corner of Figure 3 visualizes how these building blocks are presented as relation schemas: the name of relation comes first, followed by a list of variable types in the parenthesis (e.g., red triangle represents *switch*).

Apart from the previously mentioned input ($PacketIn$) and output (Fwd) relations, in this example Facon also assumes inputs include $InBound$ and $OutBound$ that represent ground facts of the network topology: $InBound(s1, p1)$ says the port $p1$ of switch $s1$ is connected to the inside of the network. Though they do not belong to the standard controller-switch communication protocol, thus cannot be directly observed from legacy program I/O traces, we will discuss how to automatically collect these relations in Section 7.

Concretely, Facon puts all relation schemas together and creates rule skeletons, e.g., a skeleton could look like this:

$$fwd(\triangle, \diamond, \square) : -PacketIn(\triangle, \diamond, \diamond, \square), InBound(\diamond, \square).$$

Facon's job is to find all valid variable name assignments for the skeleton rule. Section 4.2 describes the technical details on how do we efficiently enumerate all valid rules.

Step 2 and 3: Construct Programs and evaluate on examples. Facon iterates between these two steps: first constructs a batch of candidate programs (step 2), then verifies the candidates against all examples (step 3). It returns a target program if it satisfies all examples (the green check mark in Figure 3), otherwise (the red cross mark) goes back to step 2 to construct another candidate set. Section 4 elaborates the search algorithm.

The bottom of Figure 3 shows the synthesized program, for our simplified stateful firewall example. It satisfies all given input-output examples.

Generating input-output examples correctly is an important but orthogonal problem. These examples can either be generated by a network operator during the design phase, or gathered from execution traces from an equivalent legacy implementation.

Gathering relation schemas automatically is another essential task. In practical networks where controller and switches communicate via standard protocol, Facon can infer such information from the protocol specifications, e.g., $PacketIn$ message format from OpenFlow specifications. Other relations that are not part of the protocol interfaces, e.g., $InBound$, can be inferred from network configuration files.

3.Problem Formulation

The goal of example-guided program synthesis is to generate a program that is consistent with given examples.

Definition 1. *Given example inputs I, positive outputs O_+, negative outputs O_-, an example-guided program synthesis task is to find a program P, such that:*

- $\forall o \in O_+, P \bigwedge I \vDash o$
- $\forall o \in O_-, P \bigwedge I \nvDash o$

The $\vDash$ is the logical derivation symbol. $P \bigwedge I \vDash O$ means program P and input I produces output O.

Examples are represented as ground relations, classified into three sets: the inputs I, the positive outputs O_+, and the negative outputs O_-. A program P is consistent with examples if on input I, it produces all positive output in O_+, and none negative outputs O_-.

4.Synthesis Algorithm

Given the schemas of the input relations, the synthesis process consists of three phases:

Phase 1. Refine each relation schema by replacing the variable type with names. For example, suppose we have a relation schema $link(switch_t, switch_t)$, a refinement would be $link(X, Y)$.

Phase 2. Put these refined relation together to construct an NDLog rule. For example, $path(X, Y) : -link(X, Y)$.

Phase 3. Find the minimal set of rules that makes a correct program, as defined in section 3.

We describe the basic approach (Step 1-3), called "sequential covering" in Section 4.1. In Section 4.2 we dive into more details in Step 1 and 2.

4.1 Sequential Covering

One of the problems that make program synthesis intractable is the variability of program length. Previous approach [1] to synthesize Datalog programs assumes upper bounds on both the number of rules, and the number of relations within each rule. It then joins them into a large logical formula, and feeds it in an SMT solver for a solution. This approach has limited scalability because the input length to SMT solver is polynomial to the product of these two upper bound values. And yet constraint solving problem is NP-COMPLETE [21].

We leverage the "divide-and-conquer" principle to tackle this challenge. The basic idea is to synthesize one rule (r) at a time, then remove the output of r from the positive output set O_+. Keep iterating until all example outputs are "covered" (removed). Putting all synthesized rules together gives the final program. (This principle is also called "sequential covering" in inductive logic programming tasks [17] [24]).

The reason why "divide-and-conquer" applies to NDLog synthesis is that each rule is evaluated independently in Datalog programs (section 2.2). And the program output is the union of each rule's output. Therefore, combining rules that cover disjoint subsets of the output, gives the final program that covers all the outputs.

Algorithm 1 Basic example-guided synthesis procedure

```
1:  P ← {}
2:  while O+ ≠ ∅ do
3:      for r in make_new_rules() do
4:          O' ← r ∧ I
5:          if O' ∩ O+ ≠ ∅ and O' ∩ O− == ∅ then
6:              satisfied ← True
7:              break
8:          end if
9:      end for
10:     if satisfied then
11:         P ← P ∪ {r}
12:         I ← I ∪ O'
13:         O+ ← O+ − O'
14:     else
15:         return Failure
16:     end if
17: end while
18: return P
```

Algorithm 1 depicts the basic procedure of "sequential covering". P is the program, which is a set of NDLog rules. I are example inputs. O_+ are positive example outputs, and O_-

are negative outputs. O' are outputs produced by candidate rule r with regard to I.

The outer loop adds a new rule to the program. The inner loop finds the next rule to add, by iterating over syntactically correct rules and evaluating it against the input-output examples. A rule r is added if it produces some positive outputs, and no negative output (line 5). Once r is added to P, its outputs are removed from the positive example output set O_+ (line 13). P is returned when O_+ becomes empty.

Recursive rules. In recursive rules, the head relation also appears in the rule body. This is useful for computing graph properties, such as reachability in the network. For example, in the rule $reach(X, Y) : -edge(X, Z), reach(Z, Y)$, $reach$ appear in both rule head and rule body. However, the basic sequential covering algorithm cannot synthesize recursive rules: without head relation instances in the inputs, recursive rules would not be triggered. To handle recursive rules, we add the outputs produced by a newly found rule to the input set for future rule evaluations (line 12).

4.2 Candidate Rules Enumeration

We now dive into step 1 and 2: refining relations from schemas, and using these refined relations to construct candidate rules. These are done by the $make_new_rules()$ subroutine in algorithm 1.

The challenge is to avoid generating semantically equivalent rules. In any logic program, variables are local to each individual rule. Therefore, for two rules that share the same set of relations, what distinguishes them is how the variables are matched to each other. For example, $r1 : p1(X, Y) : -p2(X, Y)$, and $r2 : p1(X, Z) : -p2(X, Z)$ are equivalent in semantics. If we assign each variable independently (i.e. picking an element from the set $\{X, Y, Z\}$ to put each position), we will generate both $r1$ and $r2$.

To solve this problem, we enumerate on different partitions of $\{p11, p12, p21, p22\}$ instead, where p_{ij} denotes the j_{th} variable in relation p_i. Partitioning a set means grouping set elements into non-empty subsets. We assign elements within the same subset the same name. Since variables of different types cannot share the same name, grouping one type at a time will not exclude any valid candidate rule.

Algorithm 2 *make_new_rules* subroutine

```
1: P ← {}
2: for t in types do
3:     V_t ← {all variables of type t}
4:     P_t ← partitions(V_t)
5:     P ← P × P_t
6: end for
7: for p in P do
8:     rule ← assign_names(p)
9: end for
```

Algorithm 2 depicts the $make_new_rules()$ subroutine. The first for loop finds all possible ways to subgroup variables. We handle one variable type at a time: for each variable type t, go through the relation schemas, and collect the variables of type t into a set V_t (line 3). We then find all partitions of the set V_t, and keep a Cartesian product of these partitions in P. The second for loop enumerates all partitions in P. For each partition scheme p, assign the variables in the same subset with the same name (line 8).

In summary, two insights reduce rule space: (1) the use of variable types to factor the candidate rule space, and (2) for each variable type, we enumerate different groupings of variables, rather than assigning name to them independently.

4.3 Soundness and Completeness

Based on the correctness notion in definition 1, algorithm 1 is both sound (every output is correct), and complete (always outputs a correct solution if there exists one).

Algorithm 1 is sound. This is trivially true because our algorithm only returns a solution (program) after verifying its correctness.

Algorithm 1 is complete. By the analysis in section 4.2, all syntactically valid rules are enumerated, and verified for correctness. If there exists a correct solution, that is, a set of rules that produces the desired outputs, algorithm 1 always returns one. If there does not exist a correct solution, algorithm 1 always terminates with a failure. Therefore, algorithm 1 is complete.

5.Evaluation

Prototype Implementation. We implement a proof-of-concept prototype, Facon, using 814 lines of python codes and Z3 library [5]. Z3 (fixedpoint engine) is used to evaluate candidate programs.

Results. We validate the usability of Facon by synthesizing four NDLog programs: (1) A recursive reachability program that computes the reachability information among nodes within the network; (2) Learning switch that automatically learns the MAC address and switch port mapping information; (3) Stateful firewall, as described in section 2, blocks outside traffic unless it's replying to earlier requests sent from inside, and (4) Application-based forwarding, that forward packets based on what application it's visiting (TCP port number).

We run Facon on the above benchmark synthesis inputs, on MAC OS X 10.13 with 2.7 GHz Intel Core i5, and 8 GB RAM. The results are shown in Table 1.

The second major column in table 1 describes the basic facts of the programs, where L denotes the number of rules, $|R|$ and $|V|$ denotes the number of relations and variables. The third major column |S| is the search space for an SMT solver. The fourth column shows the performance of our

Program	|L|	|R|	|V|	|S|	|E|	#iters	Time(s)
Reachability	2	2	6	10^5	2	226	0.4
Learning Switch	1	2	7	10^6	1	11	0.02
Stateful Firewall	2	4	11	10^{11}	4	13497	72.0
App-based Forwarding	2	5	13	10^{14}	3	28829	149.0

Table 1: Facon's performance. Search space |S| shows the time complexity of the equivalent constraint solving problem. #iters shows how much fewer iterations Facon takes.

synthesizer, in terms of number of examples ($|E|$), number of iterations, and synthesis time.

The main takeaways are: (1) with large input size, like stateful firewall, and Application-based forwarding, Facon drastically reduces the number of iterations to search for the satisfying program, within a huge program space. (2) Facon synthesizes NDLog programs within reasonable time budget. Note that the timings in table 1 are collected on a commodity laptop. Considering that the candidate program evaluation stage consumes most of the time, and that this stage is highly parallelizable: candidate programs are evaluated independently, we are confident that we can reduce the timing within seconds on high-performance servers.

More complicated programs. We use the above simple programs to evaluate the feasibility and performance of Facon. In future work, we consider these practical scenarios which can be benefited by Facon: (1) Automating program refinement in test-driven development. Test-driven development is a well-established paradigm of software engineering, where a set of tests, in the form of input-output examples, are used to guide the iterative software refinement process. Facon can further automate this process. (2) Handling corner cases. A correct robust network program needs to deal with all corner cases that can be easily missed by a human operator (e.g., potential looping problem in OSPF misconfiguration). Moreover, the correct handling of the corner cases often require revisits of the rest of the code base. As a result, manually managing all corner cases is often a challenging task. In contrast, the operator can leverage Facon to automatically handle the corner cases with correctness guarantee.

6.Related Work

Network Configuration synthesis. NetComplete [6] and Propane [2] both compile high-level routing policies into distributed router configurations. However, they require formal specifications of the network policies as input. Facon uses examples to specify programmer intentions instead.

Datalog Program synthesis. Zaatar [1] synthesizes general Datalog programs using input-output examples. It translates syntax and examples into constraints, and uses SMT solver to find the target program. While Facon uses search-based algorithm and scales better. QuickFOIL [24] also synthesizes first-order logical rules. But it focuses on learning relations among database objects, while Facon handles both relational database schema, and arbitrary functions and data structures at the same time.

Automatic program debugging. Meta-Provenance [22] is an automated tool that uses execution trace information to fix minor bugs in programs. Meta-Provenance is capable of fixing localized errors, e.g., a cut-and-paste error or errors isolated to a line of code. However, Facon's scope is significantly bolder and more general, in that we plan to generate entirely new programs, not fix isolated bugs in programs.

7.Ongoing Work

Scale synthesis algorithm. Although the algorithm presented in this work (section 4) reduces the search space by orders of magnitudes than previous constraint-based approaches, its exhaustive enumeration nature still hinders it from scaling beyond a few tens of free variables. We are exploring admissible heuristics to guide the program searching direction. This can speed up searching with high probability, while conserving soundness and completeness.

Automatic example generation. We will explore automatic ways to gather examples for synthesis, by observing the I/O traces of legacy running programs. A promising technique is active learning [20], which samples a small representative set of examples from a much larger example pool.

Richer DSL support. Our current work focuses on synthesizing logic-based declarative DSLs. In the future, we plan to general to other DSLs such as Pyretic [14] and Kinetic [9].

Specification-based synthesis Verification tools such as VeriFlow [8] and Cocoon [19] provides high-level specifications. These specifications can serve as alternative ways to describe user intentions than input-output examples. Specifically, Cocoon decomposes high-level specifications into smaller concrete ones. Facon can be applied to synthesize the program implementations for each concrete specifications.

Acknowledgments

We thank Mayur Naik, Xujie Si, and Woosuk Lee for pointing us towards Datalog synthesis. We thank our shepherd Marco Canini, Yiwen Jin, and the anonymous reviewers for their helpful feedbacks. This material is based upon work supported in part by the the Defense Advanced Research Projects Agency (DARPA) under Contract No. N00014-18-1-2021, NSF grant CNS-1513679, and NSF grant #1657285. Any opinions, findings and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of DARPA or NSF.

References

[1] Aws Albarghouthi, Paraschos Koutris, Mayur Naik, and Calvin Smith. 2017. Constraint-Based Synthesis of Datalog Programs. In *International Conference on Principles and Practice of Constraint Programming*.

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

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

[4] Chen Chen, Lay Kuan Loh, Limin Jia, Wenchao Zhou, and Boon Thau Loo. [n. d.]. Automated verification of safety properties of declarative networking programs. In *Proceedings of the 17th International Symposium on Principles and Practice of Declarative Programming*.

[5] Leonardo De Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer.

[6] Ahmed El-Hassany, Petar Tsankov, Laurent Vanbever, and Martin Vechev. 2018. NetComplete: Practical Network-Wide Configuration Synthesis with Autocompletion. In *NSDI*. USENIX Association. https://www.usenix.org/conference/nsdi18/presentation/el-hassany

[7] Naga Praveen Katta, Jennifer Rexford, and David Walker. 2012. Logic programming for software-defined networks. In *Workshop on Cross-Model Design and Validation (XLDI)*.

[8] Ahmed Khurshid, Wenxuan Zhou, Matthew Caesar, and P Godfrey. 2012. Veriflow: Verifying network-wide invariants in real time. In *HotNets*.

[9] Hyojoon Kim, Joshua Reich, Arpit Gupta, Muhammad Shahbaz, Nick Feamster, and Russell J Clark. 2015. Kinetic: Verifiable Dynamic Network Control.. In *NSDI*.

[10] Teemu Koponen, Keith Amidon, Peter Balland, Martín Casado, Anupam Chanda, Bryan Fulton, Igor Ganichev, Jesse Gross, Paul Ingram, Ethan J Jackson, et al. 2014. Network Virtualization in Multi-tenant Datacenters.. In *NSDI*.

[11] Boon Thau Loo, Tyson Condie, Minos Garofalakis, David E Gay, Joseph M Hellerstein, Petros Maniatis, Raghu Ramakrishnan, Timothy Roscoe, and Ion Stoica. 2009. Declarative networking. *Commun. ACM* (2009).

[12] Boon Thau Loo, Tyson Condie, Joseph M Hellerstein, Petros Maniatis, Timothy Roscoe, and Ion Stoica. 2005. Implementing declarative overlays. In *ACM SIGOPS Operating Systems Review*. ACM.

[13] Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. 2008. OpenFlow: enabling innovation in campus networks. *ACM SIGCOMM Computer Communication Review* (2008).

[14] Christopher Monsanto, Joshua Reich, Nate Foster, Jennifer Rexford, David Walker, et al. 2013. Composing Software Defined Networks.. In *NSDI*.

[15] Tim Nelson, Andrew D Ferguson, Michael JG Scheer, and Shriram Krishnamurthi. 2014. Tierless Programming and Reasoning for Software-Defined Networks.. In *NSDI*.

[16] Chaithan Prakash, Jeongkeun Lee, Yoshio Turner, Joon-Myung Kang, Aditya Akella, Sujata Banerjee, Charles Clark, Yadi Ma, Puneet Sharma, and Ying Zhang. 2015. Pga: Using graphs to express and automatically reconcile network policies. In *ACM SIGCOMM Computer Communication Review*. ACM.

[17] J. Ross Quinlan. 1990. Learning logical definitions from relations. *Machine learning* 5, 3 (1990), 239–266.

[18] Mark Reitblatt, Nate Foster, Jennifer Rexford, and David Walker. 2011. Consistent updates for software-defined networks: Change you can believe in!. In *HotNets*.

[19] Leonid Ryzhyk, Nikolaj Bjørner, Marco Canini, Jean-Baptiste Jeannin, Cole Schlesinger, Douglas B Terry, and George Varghese. 2017. Correct by Construction Networks Using Stepwise Refinement.. In *NSDI*.

[20] Burr Settles. 2012. Active learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning* (2012).

[21] Michael Sipser. 2006. *Introduction to the Theory of Computation*. Vol. 2. Thomson Course Technology Boston.

[22] Yang Wu, Ang Chen, Andreas Haeberlen, Wenchao Zhou, and Boon Thau Loo. 2015. Automated network repair with meta provenance. In *HotNets*.

[23] Yifei Yuan, Dong Lin, Rajeev Alur, and Boon Thau Loo. 2015. Scenario-based programming for SDN policies. In *ACM CoNEXT*.

[24] Qiang Zeng, Jignesh M Patel, and David Page. 2014. Quickfoil: Scalable inductive logic programming. *Proceedings of the VLDB Endowment* (2014).